15-418/516 Final Project: Parallel-In-Time Integration

Alexius Wadell (awadell)

May 5, 2022

1 Summary

For my project, I implemented the Multigrid Reduction in Time (MGRIT) and Parareal algorithms for solving ordinary differential equations using the threading primitives provided by the Julia Programming Language. I then benchmarked both algorithms on both my 2020 Macbook Pro (M1 chip), and my group's cluster (AMD EPYC 7713 64-Core Processor). Overall, the overhead of Julia's threading primitives limited the speedup on small problems, but for a sufficiently expensive differential, I was able to achieve notable speedups over a serial implementation on both systems.

Throughout this project, I learned a lot about applying the concepts from the course to debugging and understanding the performance limitations of Julia's threading primitives. While the tools presented in class (OpenMP, CUDA, AVX) were extremely transparent, and thus complex to use, the primitives provided by Julia were both simple and opaque. Unfortunately, this made it difficult to deceiver what exactly was happening under the hood.

2 Background

Solving ordinary differential equations (ODE) numerically is generally done using by incremental stepping forwards in time. For example, given a differential equation (Eq. 1) and initial conditions $u(0) = u_0$, we can step forward in using Euler's method (Eq. 2).

$$\frac{du}{dt} = f(u, t) \tag{1}$$

$$u(t + dt) = u(t) + \delta t \cdot f(u(t), t) \tag{2}$$

More generally, a set of discrete time points $t = i\delta t$ for $i = 0, 1, 2, \ldots N$, and some serial integrator $\Phi(u_i) = u_{i+1}$, we want to find u_i for $i \in [1, N]$. One of the benefits of the MGRIT algorithm is that we can use any serial integrator for Φ. For example, using Euler's method, $\Phi(u_i) = u_i + \delta t f(u_i, t)$. By leveraging, DifferentialEquations.jl [1], I was able to easily take advantage of a wide range of serial integrators. For the sake of this project, I have used the `Tsit5` integrator, a fourth-order Runge-Kutta method developed by Tsitouras.

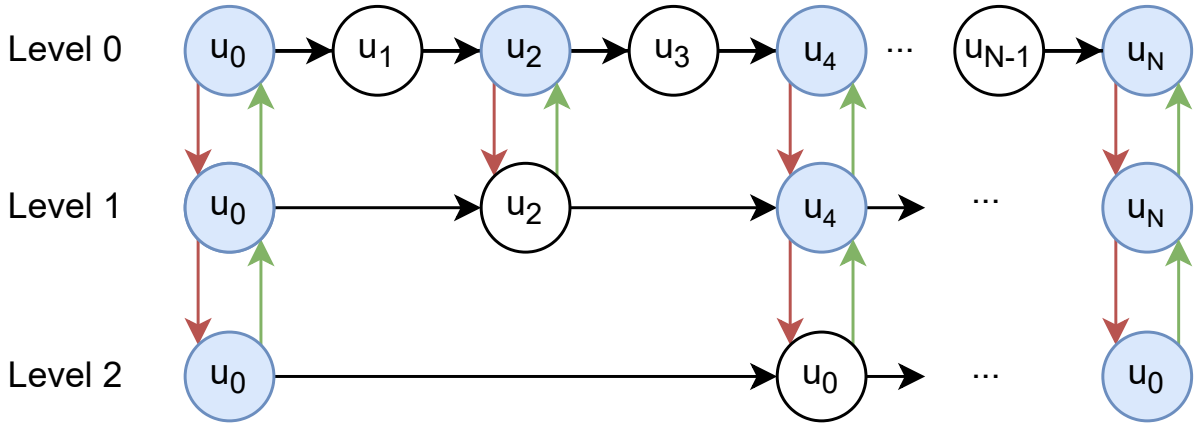


Figure 1: MGRIT Algorithm

Typically, solving an ODE is done by iteratively applying a serial integrator to set forward in time, resulting in a cost that is linear in the number of time steps. As each step is dependent on the last, there is no obvious method for paralleling over time. Instead, most work has focused on parallelization within the differential f, with most intergrator suites lacking any parallel-in-time capabilities [3].

In this project, I have implemented two parallel-in-time algorithms for solving ODEs: Parareal[4] and MGRIT[5]. The Parareal Algorithm, iteratively applies a "coarse" integrator to predict the state at some coarse time step($t\delta$), and a "fine" integrator to refine the prediction between coarse time steps in parallel. This is repeated until convergence is achieved, which for N_c coarse time steps will occur in at most N_c iterations [4]. As after N_c iterations, we have effectively applied the fine integrators in serial. Of course, in order to achieve speedup we must converge in less than N_c iterations, ideally significantly less. As such, the usefulness Parareal algorithm is limited to problems that require very high accuracy, but can also be well approximated using a coarse integrator. As such, it's implementation here was mostly as a stepping stone to MGRIT.

The Multigrid Reduction in Time (MGRIT) algorithm reframes solving an ODE as solving a large diagonal system of linear equations, and is currently being developed into the XBraid package by Lawrence Livermore National Laboratory [6]. Specifically, MGRIT is trying to solve the linear system of equations $Au = g$ given by Eq. 3. Where $g_0 = u_0$, and Φ is the linear realization of the serial integrator (i.e. $\Phi(u) \equiv \Phi \cdot u$). From here, the MGRIT algorithm now treats "time" as an additional spatial dimension, and proceeds to solve it using existing multigrid reduction methods. That is, instead of solving the entire system, we solve segments of the solution on a fine grid, and use the coarse grid to rectify the various sub-segments.

$$Au = g \rightarrow \begin{bmatrix} I & & & \\ -\Phi & I & & \\ & \ddots & \ddots & \\ & & -\Phi & I \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{bmatrix} = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_N \end{bmatrix} \tag{3}$$

Practically, this is done using the algorithm outlined in Algorithm 1 and Figure 1. But a high level

it, we partition the time domain $t = i\delta t$ into C-points $i = jm$ (Blue) and F-points (White), where m is some coarsening factor, and is typically set to be 2. We then update each F-point using Φ and stepping forward from the previous C-point, this is done in parallel and is called an F-relaxation. Next we update each C-point using Φ and stepping forward from the proceeding F-point, this is also done in parallel and is called an C-relaxation. An additional F-relaxation is then performed completing the FCF-cycle. The C-points of the current level are then used to update the next level's point (Red Arrows), and we recurse into the next level. After updating the coarser levels, we then update the current level's C-Points based on the next level (Green Arrows) and perform a F-relaxation to update the F-points. This is repeated until convergence is achieved.

As noted by Friedhoff et al., the Parareal algorithm is MGRIT restricted to only 2 levels, and using a single F-relaxation, instead of the FCF-cycle used in MGRIT.

Algorithm 1 The MGRIT Algorithm

if Current level l is the Coarsest Level **then**
  Solve the system serially: $A_L u_L = g_L$
else
  Update the current level using an FCF relaxation
  for $j = 0, 1, \ldots N_l/m$ **do** ▷ In-Parallel F-Relaxation
    Update the F-points: $u_{i+1} = \Phi(u_i) + g_i$
  end for
  for $j = 1, \ldots N_l/m$ **do** ▷ In-Parallel C-Relaxation
    Update the C-Points: $u_{mj} = \Phi(u_{mj-1}) + g_{mj-1}$
  end for
  for $j = 0, 1, \ldots N_l/m$ **do** ▷ In-Parallel F-Relaxation
    Update the F-points: $u_{i+1} = \Phi(u_i) + g_i$
  end for
  Inject the residuals of the current level into the next level: $\vec{g}_{l+1} = R_I(\vec{g}_l - A_l\vec{u}_l)$
  Solve the next coarse level: $MGRIT(l+1)$
  Refine this level using the coarse solution: $\vec{u}_l = \vec{u}_l + P_\Phi \vec{u}_{l+1})$
end if

3 Approach

For this project, I used the Julia Programming Language[7] to implement the MGRIT algorithm on a Shared Memory machine using Julia's built-in threading primitives. I choose to implement this project using Julia largely due the strength of it's ecosystem for Differential Equations [1]. Hardware-wise, I decided to benchmark my implementation on my 2020 MacBook Pro (M1 Chip) and on the AMD EPYC 7713 64-Core Processor available via the RM nodes on the Bridges2 [8] or Arjuna [9] computing clusters. Ultimately, this project was guide by my research interest (Modeling Dynamical Systems) and the types of hardware and software I typically use.

3.1 Data Structures

I used preallocated vectors of vectors (`Vector{Vector{uType}}`) for storing the solution u, residuals g, and a scratch space for computing the updates to each level. Additionally, as some integrators Φ require

additional storage, I preallocated a separate integrator for each thread. From there it was largely a matter of looping over F/C-points, performing the F/C-relaxations, and then recursing into the next level.

3.2 Thread Overhead

One issue, that was unexpected, was the large overhead of the threading primitives. For example, the `residual(integrator)` function in `src/mgrit_algo.jl` performs the following operations:

for **do** $i = 1, \ldots, N$
  $u' = |u_i|$ ▷ Compute the norm of the solution
  $g' = |g_i|$ ▷ Compute the norm of the residual
  $e = \frac{g'}{\alpha + \beta u'}$ ▷ Scale based on an absolute α and relative β tolerance
  $r = \max(r, e)$ ▷ Reduce to the max e over all i
end for

N	Serial	`Threads.@threads`	`@batch minbatch=64`
10	10.0	4000.	9.9
100	8.8	110.	8.7
1000	9.0	51.	9.1
10000	9.0	35.	8.9

Table 1: Median time per element in nanoseconds to compute the residual. Measured on a 2020 MacBook Pro (M1) using 4 threads for the @threads and @batch examples. Overall, benchmarking suggests that the overhead of threading is significant (10s of microseconds) and the lack of a reduction primitive effectively serializes the work.

I benchmarked several different methods of parallelization using BenchmarkTools.jl [10]. The exact benchmarks can be found in `benchmark/bench_residual.jl`. The Serial results were benchmarked using from a julia process with `-O3` enabled an no-threads, While `@batch` and `Threads.@threads` were benchmarked using from a julia process with `-O3` and 4 threads enabled. All benchmarks were run on a 2020 MacBook Pro.

From the results tabulated in Table 1, we can see that the overhead of launching 4 threads, is negligible compared to the potential speedup provided by the additional threads. This isn't shocking, with a serial cost of $\tilde{1}0$ ns or 14–19 instructions per element given the 1.4 1.9 GHz clock rate of a single core. Further, community benchmarks suggest that the overhead of `Threads.@threads` vs. a serial saxpy implementation requires an input side of $\approx 10^4$ elements for an Intel Core i7[11]. While not an apples to apples comparison, this does confirm our observation here `Threads.@threads` has an overhead on the order of tens of microseconds.

This can be somewhat mitigated by using the `@batch` macro provided by Polyester.jl [12]. This macro avoids most of the overhead of launching tasks by batching working assigned to a single thread. As benchmarked with a `minbatch = 64` this results in 1 task for the $N = 10$ case, and 2 tasks for the $N = 100$ case. As we can see, this does eliminated the overhead of launching 4 threads, but still provided virtual no speedup. Benchmarking with and without the final reduction step, $r = \max(r, e)$, reduces the time per element to $\tilde{2}$ ns or roughly a 4x speedup. Strongly suggesting that the final reduction step is effectively serializing the work. Unfortunately, julia lack a threaded reduction primitive, and attempts to

batch the reduction step were unfruitful.

3.3 Runtime Allocations Usage

4 Results

To benchmark my implementation of MGRIT, I examined the one-dimensional diffusion equation (Eq. 4) in-line with analysis presented by Friedhoff et al. [5]. This was then discretized into a one-dimensional mesh of with N_s points and spacing of $\Delta x = \frac{1}{N_s+1}$ using the method of lines and central finite differences.

$$\begin{aligned} \frac{\partial T}{\partial t} &= \frac{\partial^2 T}{\partial x^2} \\ T(x,0) &= \sin(2\pi x) \\ T(0,t) &= 0 \end{aligned} \tag{4}$$

As the resulting system is only stable for $\Delta t < \frac{1}{2}\Delta x^2$, I set the time step to be at least 8 times smaller then required. This allows MGRIT to have 3 levels of temporal refinement while remaining stable at all levels. In order to keep the total amount of work constant, regardless of N_s, I integrated the system from $t = 0$ to $256\Delta t$. Overall, this boils down to performing a sparse matrix-vector product, with a compute cost $\mathcal{O}(N_s)$ per evaluation of the differential. I then swept over both the problem size N_s, and the number of threads used N_t, and recorded the median runtime over 5 runs for the MGRIT, Parareal and serial (Tsit5) integrators. Due to problem size, these benchmarks were only performed using on the AMD EPYC 7713 64-Core Processor system provided by the Arjuna Cluster.

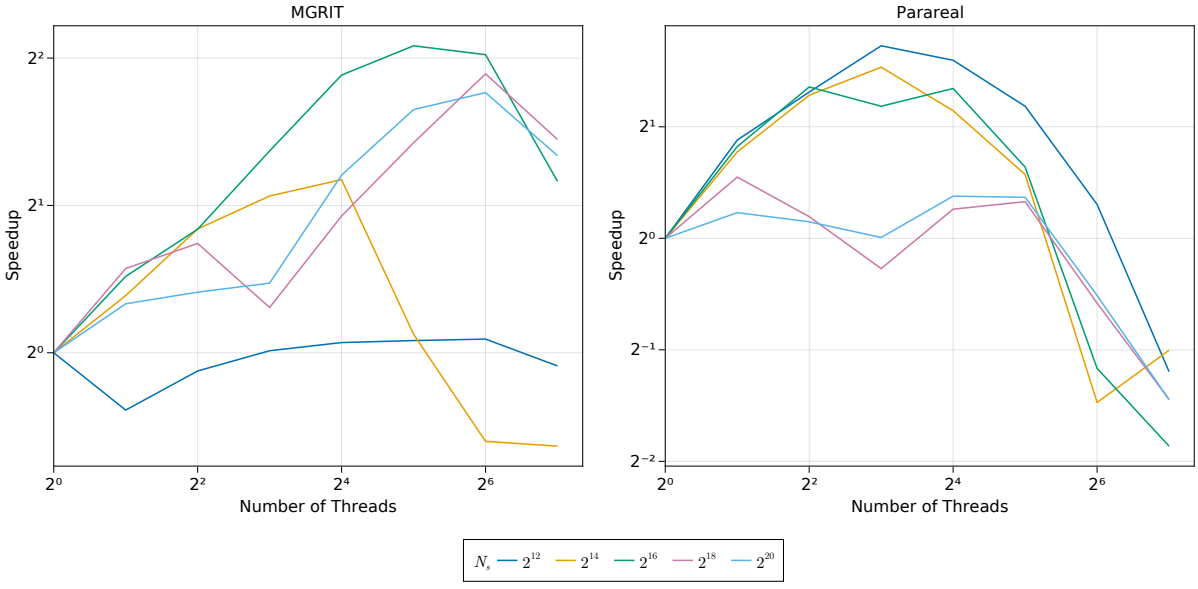


Figure 2: Speedup of the MGRIT and Parareal Integrators relative to their performance with a single thread

As shown by Figure 2, both the MGRIT and Parareal integrators do show moderate speedups as the number of threads increases. Due to the high cost of launching tasks using Julia's primitives (Section 3.2),

our speedup is highly dependent on problem size N_s. This is expected as the work per differential must be sufficiently large to cover the cost of launching tasks. For MGRIT, it appears that $N_s \geq 2^{16}$ is sufficient to support 64 threads as speedup continues to increase with the number of threads. However for smaller problems $N_s = 2^{14}$, above $\tilde{1}6$ threads, the overhead of the additional threads outweighs the additional computing power. Unfortunately, for small problems ($N_s \leq 2^{12}$, the overhead of the threads virtually eliminates any speedup.

The hump shaped speedup curve of the Parareal integrator is a function of it's limitations as an integrator. For low thread counts, the coarse integrator is able to quickly estimate a solution that is then refined by the fine integrator. Resulting ina solution after only a few iterations. However, as the thread count increases both the communication costs and task overhead increase linearly. This results in the sudden drop in speedup with increasing thread count. It is effectively similar to the drop in speedup observed for the MGRIT integrator, but occurs sooner, as the algorithmic advantage of more threads is less.

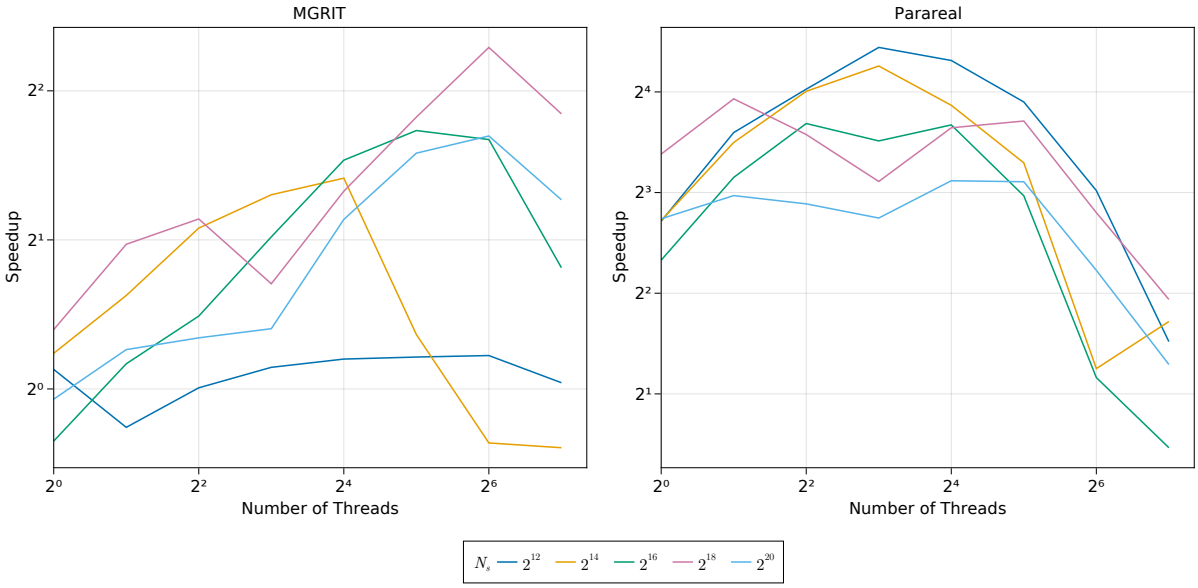


Figure 3: Speedup of the MGRIT and Parareal Integrators relative to the performance of the Tsit5 integrator

Both the MGRIT and Parareal algorithms were able to beat out the performance of the Tsit5 provided by DifferentialEquations.jl (Figure 3). This is a pleasant surprise, as the Tsit5 algorithm is DifferentialEquation.jl's recommended integrator for non-stiff problems[1]. Further, in benchmarks against other packages Tsit5 and DifferentialEquations.jl tend to beat out offerings [14]. That is, matching, let alone beating, the performance of the Tsit5 algorithm suggest that the parallel-in-time integration provided here provides a not insignificant value-add.

It should be noted that the performance of the Parareal over Tsit5 is likely in part due to the coarse integrator being allowed to make adaptive steps. In effect this drastically reduces the cost of the coarse integrator, while retaining all of the benefits of the fine integrator. Additionally, the Tsit5 integrator does

[1]For the MATLAB Fans, while not exactly ode45, it is the recommended equivalent[13]

save significantly more information (For example, it provides an accurate interpolant of the results), as well as suffering from run-time memory allocations as it does not pre-allocate space for the results.

4.1 Profiling

The next few pages show results from profiling the MGRIT algorithm on the 1D diffusion equation. Profile was conducted using Julia's builtin statistical profiler, and visualized as a flamegraph using PProf.jl [15]. Overall, similar trends to the benchmark results are observed, with larger problem sizes reducing the marginal cost of launching tasks.

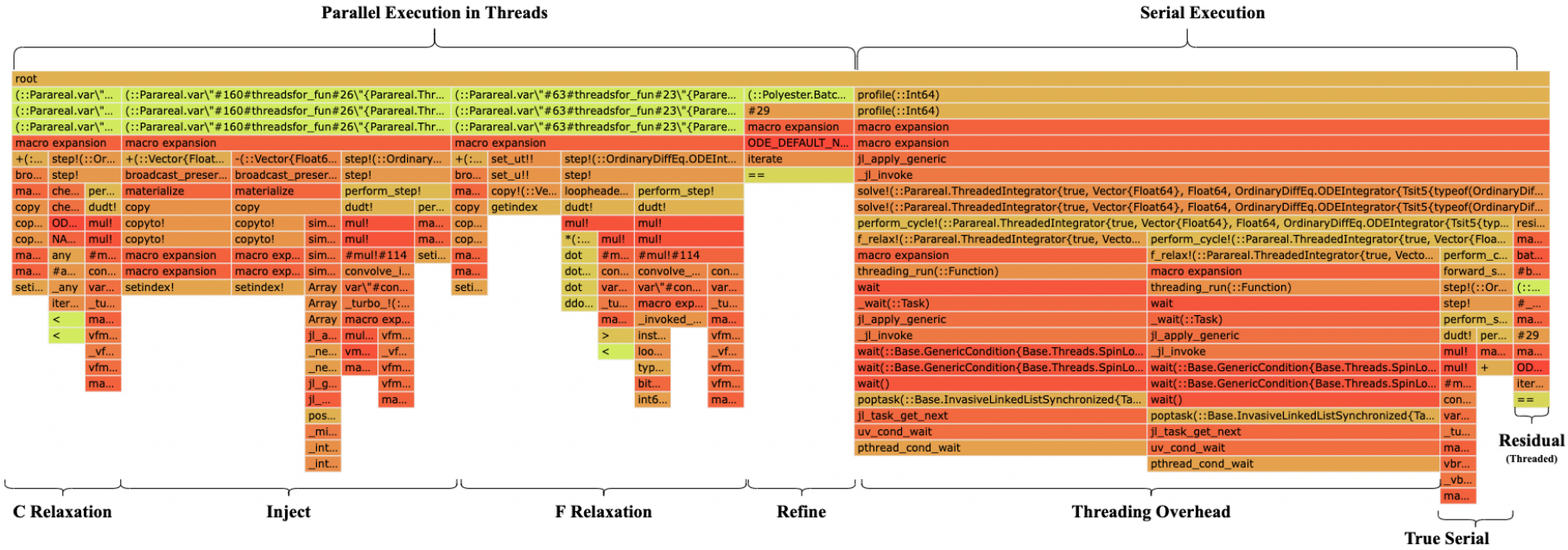


Figure 4: Annotated Flamegraph of MGRIT for a 1D Diffusion Equation with $N_s = 2^{12}$ and $N_t = 64$. As shown, threading overhead is significant and dominates the overall cost for small problem sizes. The Flamegraphs shown in Table 2 follow a similar trend, but with different proportions of work. Profiling occurred on an AMD EPYC 7713 64-Core Processor.

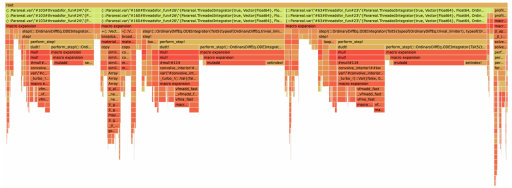
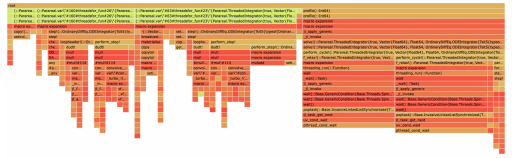
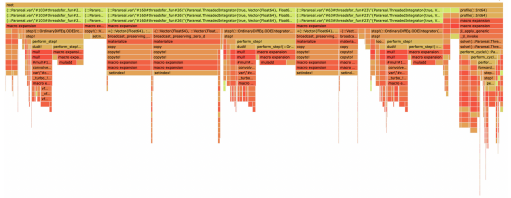
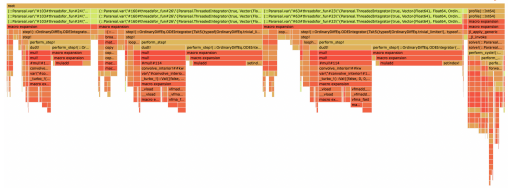
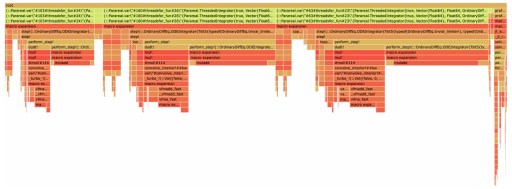
N_s:	2^{12}	2^{16}	2^{20}
$N_t = 4$			
$N_t = 16$			
$N_t = 64$			

Table 2: Flamegraphs from profiling MGRIT on a 1D Diffusion Equation, while varying the work size N_s and number of threads N_t. Profiling occurred on an AMD EPYC 7713 64-Core Processor. Overall, as work size increases, the overhead of launching threads becomes negligible, shrinking the serial execution region (See Figure 4 for a labeled flamegraph). As expected from the speedup plots (Figure 2), larger thread counts required a larger work size to effectively overcome the overhead of launching threads.

4.2 Working Set Size

As shown in Figure 2, the $N_s = 2^{20}$ performs worse than smaller work sizes, despite having a generally higher level of parallelism (Table 2). While I was unable to confirm this via measurement with `pref` due to noise from Julia's JIT compilation, I believe this is due to the cache size of the AMD EPYC 7713 processor. With a state of 2^n 64 bit floating point numbers, the total memory cost per time step is $(1 + 1/2 + 1/4 + 1/8) \times 8 \times 2^n \to 15 \times 2^n$ bytes. Accounting for the 256 timesteps and 3x duplication of the state (u, g and scratch space), this gives a total working set size of 11520×2^n bytes. The maximum working set size the can fit into the L1, L2 or L3 caches of the AMD EPYC 7713 processor has be tabulated in Table 3.

Cache	Size [MiB]	Max N_s for single step	Max N_s for Entire Solution
L1	4	$2^{16.5}$	$2^{8.5}$
L2	32	$2^{19.5}$	$2^{11.5}$
L3	256	$2^{22.5}$	$2^{14.5}$

Table 3: Cache Sizes for the AMD EPYC 7713 Processor and maximum work sizes for a 1D Diffusion Equation. Cache sizes provided by WikiChip [17].

As the working set size for a single time step at $N_s = 2^{20}$ is larger than the L3 cache size, each core needs to access main memory to compute even a single timestep. This results in the $N_s = 2^{20}$ case having the lowest speedup with increasing thread counts due to increased contention for the main memory, and higher cache capacity misses. This is further confirmed by the fact that the $N_s = 2^{18}$ case saw the largest speedup overall (Figure 2). As the working set for a single step can fit within the L2 cache, while the entire solution can fits within the L3 cache, when split across 64 cores.

5 Future Work

Given the results presented (Figure 23), a Julia implementation of the MGRIT algorithm appears to be a viable contender for adding parallelism to the differential equation solver. However, as noted above (Section 3.2), the overhead of launching tasks is significant. My choice to use Julia's threading primitives instead of it's distributed computing ones or an external package (i.e. MPI.jl) was largely motived by my use case: Accelerating solves on my personal laptop. However, given the overhead of threads, and the tendency of MGRIT to require immense parallelism [5], a message passing approach is likely to provide improved performance for sufficiently numerous computing resources. This would also be in line with the approach taken by other implementations of the MGRIT algorithm [6, 16].

Finally, as noted above (Section 3.3), my current implementation suffers from runtime memory allocations when resetting the serial integrators. As the serial integrators provided by DifferentialEquations.jl are designed for serial solving of ODEs, they are not optimized for repeatedly being reset to a new initial condition. Specifically, both the `Euler` and `Tsit5` integrators appear to re-allocate their internal caches when reset. Further, performance could be achieved by eliminating these dynamic memory allocations.

Finally, while not pursued here, implementing the MGRIT algorithm on the GPU could provide access to the numerous execution units typically required [5]. This concept was initially floated during my

milestone update report, as I was not seeing any notable performance gains at the time. However, by chasing down memory allocations and switching after implementing MGRIT, I was able to see a speedup a lower thread counts, thus eliminating the need for the massive ($> 2^{12}$ processor counts suggested by Friedhoff et al., was no longer required.

6 Work Distribution

As this was a solo group project, all work was completed by Alex Wadell.

References

[1] Christopher Rackauckas and Qing Nie. "DifferentialEquations.Jl – a Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". In: *The Journal of Open Research Software* 5.1 (2017). DOI: 10.5334/jors.151.

[2] Ch Tsitouras. "Runge-Kutta Pairs of Order 5 (4) Satisfying Only the First Column Simplifying Assumption". In: *Computers & Mathematics with Applications* 62.2 (2011), pp. 770–775.

[3] *A Comparison Between Differential Equation Solver Suites In MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran*. Stochastic Lifestyle. Sept. 26, 2017. URL: https://www.stochasticlifestyle.com/comparison-differential-equation-solver-suites-matlab-r-julia-python-c-fortran/ (visited on 05/03/2022).

[4] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. "Résolution d'EDP par un schéma en temps «pararéel »". In: *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics* 332.7 (Apr. 2001), pp. 661–668. ISSN: 07644442. DOI: 10.1016/S0764-4442(00)01793-6.

[5] S Friedhoff et al. "A MULTIGRID-IN-TIME ALGORITHM FOR SOLVING EVOLUTION EQUATIONS IN PARALLEL". In: (), p. 12.

[6] *XBraid: Parallel Multigrid in Time*. URL: http://llnl.gov/casc/xbraid.

[7] Jeff Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671.

[8] *Bridges-2*. PSC. URL: https://www.psc.edu/resources/bridges-2/ (visited on 05/03/2022).

[9] *Arjuna Computing Cluster*. Arjuna. URL: https://arjunacluster.github.io/ArjunaUsers/ (visited on 05/03/2022).

[10] Jiahao Chen and Jarrett Revels. "Robust Benchmarking in Noisy Environments". In: *arXiv e-prints*, arXiv:1608.04295 (Aug. 2016). arXiv: 1608.04295 [cs.PF].

[11] *Gcc vs Threads.@threads vs Threads.@spawn for Large Loops - Specific Domains / Julia at Scale*. JuliaLang. Feb. 6, 2020. URL: https://discourse.julialang.org/t/gcc-vs-threads-threads-vs-threads-spawn-for-large-loops/34273 (visited on 05/03/2022).

[12] *Polyester*. JuliaSIMD, Apr. 10, 2022. URL: https://github.com/JuliaSIMD/Polyester.jl (visited on 05/03/2022).

[13] *ODE Solvers · DifferentialEquations.Jl*. URL: https://diffeq.sciml.ai/stable/solvers/ode_solve/#Translations-from-MATLAB/Python/R (visited on 05/04/2022).

[14] *ODE Solver Multi-Language Wrapper Package Work-Precision Benchmarks (MATLAB, SciPy, Julia, deSolve (R))*. URL: https://benchmarks.sciml.ai/html/MultiLanguage/wrapper_packages.html (visited on 05/04/2022).

[15] *PProf.Jl*. JuliaPerf, Apr. 20, 2022. URL: https://github.com/JuliaPerf/PProf.jl (visited on 05/05/2022).

[16] Jens Hahne, Stephanie Friedhoff, and Matthias Bolten. "PyMGRIT: A Python Package for the Parallel-in-Time Method MGRIT". Aug. 12, 2020. arXiv: 2008.05172 [cs]. URL: http://arxiv.org/abs/2008.05172 (visited on 04/28/2022).

[17] *EPYC 7713 - AMD - WikiChip*. URL: https://en.wikichip.org/wiki/amd/epyc/7713 (visited on 05/05/2022).