Parallel N-Body Performance Comparison: Julia, Rust, and More

Mark C. Lewis[1][0000−0002−7451−5176], Clarissa Garcia[1], Audrey Tollett[1], Seven Aguirre[1], Henry Hafner[1], John McMahon[1], and Amanda A. Sickafoose[2][0000−0002−9468−7477]

[1] Trinity University, San Antonio, TX 78212, USA
https://www.trinity.edu
[2] Planetary Science Institute, Tucson, AZ 85719, USA
https://psi.edu

Abstract. This paper explores parallelism performance for C, C++, Go, Java, Julia, and Rust on N-body simulations. We begin with a basic $O(N^2)$ simulation for each language based on the n-body benchmark in the Benchmark Game. The original benchmark is adjusted to include a larger number of particles and run in parallel. We also add parallelism to the force calculations using a kD-tree. This work builds on previous work by including parallelism and adding the Julia programming language to our survey. We find that for straight number-crunching, all of these languages provide similar performance, and all have sufficient support for parallelism that runtimes scale well with thread counts. On the other hand, when a spatial data structure, such as the kD-tree, is introduced, the runtimes vary dramatically between languages. In that situation, Julia's performance looks more like Python, taking over 100 times as long as Rust/C/C++ to finish. Rust comes out on top with an impressive 50% lead over C and C++.

Keywords: N-body, simulation, performance, Rust, Julia

1 Introduction

This work builds on our previous work [15] comparing language performance for N-body simulations using a variety of programming languages. Part of the goal of that paper was to see if Rust was a suitable alternative to C/C++ for larger N-body simulations that use a spatial tree to provide $O(N \log N)$ performance. In this paper, we are interested to see how well various languages work with the inclusion of shared-memory parallelism. We also add Julia to the mix of languages we consider.

We look at two separate benchmarks. The first is a basic $O(N^2)$ approach used in the n-body benchmark in the Computer Language Benchmark Game [2]. We submitted the original versions of that benchmark to the site roughly 20 years ago. They maintain a suite of benchmarks across multiple languages. They are all based on small programs, but they stress the computer in various ways and

highlight the strengths and weaknesses of different languages. In recent years, the Benchmark Game has been used as the foundation for energy efficiency studies as well [6, 10, 11].

This paper includes a second set of benchmarks using the kD-tree implementations from [15] and parallelizing the force calculations. The $O(N^2)$ code is effectively a test of raw number crunching. Particles are stored in flat arrays, and no interesting data structures are used. These simulations are also limited in how many particles can be considered because of the scaling. This leads to a smaller memory footprint in all languages. The kD-tree tests the languages in different ways. There is still a large amount of number crunching, but the building and navigation of the tree is a different type of workload. In addition, better scaling means that the simulations can be extended to over a million particles, significantly increasing the memory footprint.

For both of these styles of simulation, once the code has been adjusted properly, the actual calculation of the forces is an embarrassingly parallel problem. This makes it easy to add parallelism to the force calculations in many languages. Note that the construction of the kD-tree in parallel is non-trivial. We do not undertake that task here.

1.1 Language Selection

We began with the same set of languages used in [15]: Rust, C, C++, Java, Golang, TypeScript, and Python. For this study, we decided to replace JavaScript / TypeScript and Python with Julia. The decision to drop the JavaScript environment and Python was mainly because of the focus on parallelism. While Node.js does have support for worker threads, it is not a strength of the platform. The GIL in Python means multithreading generally provides no performance boost for CPU-bound workloads. We replaced them with Julia [3], a scripting language that is aimed in many ways at replacing Python, R, and Matlab for numerical computations. Unlike Python, R, and Matlab, Julia was designed with performance in mind, especially numerical performance. In the n-body benchmark testing [4], Julia is only topped by Chapel and Rust. That implies that Julia can beat the fastest safe implementations in C and C++ for that particular benchmark. Julia also has built-in support for parallel processing, making it an ideal candidate for this work.

As many readers might be unfamiliar with Julia, we'll provide a bit more information on it here. As was mentioned, Julia is a scripting language that aims to provide a programming interface similar to Python or R, but with a focus on performance. Julia has a fully functional REPL, and scripts can be written with low syntactic overhead. The language itself isn't object-oriented. It uses `struct`s that hold data, but they don't have methods attached. The language can be written as a dynamically typed language, but its real power is achieved when the user specifies types, which enables Julia's most unique language feature: multiple dispatch. Multiple dispatch can be reasonably well described as function overloading with dynamic binding on all the arguments. Function overloading only makes sense when the types of arguments are specified

to differentiate the overloaded versions. In our own testing, we also found that specifying types also allows Julia's JIT to produce faster code. For example, if Julia knows that it has an array of doubles, it will generate faster code to sort that array than if given an array of an unspecified type.

The motivation for having a fast scripting language is given in [7] and [12], where they describe the two language problem in scientific computing. This is when algorithms are implemented first in a language like Python where people feel they can construct them more quickly[3], but then they have to be translated into other languages when they are used on full-scale problems that require more speed. Several papers, such as [17], look at Julia's performance in different scenarios. This work aims to add to that list.

2 O(N^2) Approach

The benchmark that is part of [2] is a test of raw number crunching. It uses a first-order, symplectic integrator to integrate the orbits of the four giant planets around the Sun. This is a standard test of the accuracy of an integrator for planetary science, as the orbits should evolve in predictable ways over long timescales [13]. Unfortunately, that form of the benchmark does not work well for testing parallelism as there are only five bodies, and each time step must be calculated sequentially. To get around this, we use a setup that resembles a variable number of asteroids in order around a star.

The approach for performing the integrations also had to be updated slightly. The algorithm runs through all pairs of particles, calculates the force of gravity between the pair, and then adds that force to either an aggregate acceleration or the velocities of the particles. When done sequentially, this is done with an outer loop that runs through all the particles and an inner loop that goes through all the particles after the first one. The distance is calculated once for each pair, and that distance is used to calculate the force on each particle.

This approach minimizes the number of distance calculations, but if run in parallel, it will introduce race conditions as different iterations of the outer loop will mutate the same aggregators in the inner loop. To make the code work better in parallel, the inner loop also runs through all the particles, skipping the one the outer loop is on, and it only updates the aggregator for the outer loop particle. This results in twice as many distance calculations, but each iteration through the outer loop mutates separate memory, so there are no race conditions if the outer loop is turned into a parallel loop.

For these tests, we started with versions of the code copied directly from the Benchmark Game site. We picked the fastest version from each language that was amenable to parallelization. We then changed the loop structure, as described above, and changed the outer loop to execute in parallel. The details of how

[3] The authors of this paper don't necessarily agree with this analysis. In our experience, creating correct/working software in dynamically typed scripting languages often takes longer than in statically typed languages because of the lack of static syntax/type checking.

the outer loop was parallelized varied by language, as each language supports different parallelization mechanisms.

The code in the Benchmark Game also includes energy calculations. In that context, they are primarily used to verify that the algorithm works. These calculations are also $O(N^2)$, but with only five particles, the computational cost is minimal. As we are boosting the number of particles and taking fewer time steps, the relative cost of the energy calculation grows. This calculation can also be done in parallel with a map-reduce operation. Unfortunately, Julia and Golang do not have good support for parallel map-reduce, so the energy calculations were removed for the timing results to keep the comparisons even.

2.1 Implementations

The full version of all the $O(N^2)$ implementations can be found at `https://github.com/MarkCLewis/ParallelNBodyPerformance`. This also includes the scripts used to run and collect data from the benchmarks. Table 1 summarizes the languages, what version of the code from the Benchmark Game was used to build our solution, and what approach/library was used to add the parallelism.

Table 1. Implementation Details

Language	Benchmark Game	Parallelism
C (GCC 9.4.0/clang 10.0.0)	#6	OpenMP
C++ (GCC 9.4.0/clang 10.0.0)	#9	OpenMP
Go (1.22.3)	#3	go-parallel
Java (GraalVM-java17-22.2.0)	#5	Parallel Streams
Julia (1.10.3)	#4 and #8	Thread macro
Rust (1.77.0-nightly)	#3	Rayon

OpenMP is the standard for multithreaded HPC work in C, C++, and Fortran. Java's parallel streams were added in Java 8 and provide a simple mechanism for both parallel loops and doing functional transformations in parallel. Julia has built-in support for multithreading in the form of the `@Threads` macro. This macro provides parallel loops but doesn't include strong support for much else. While Rust doesn't currently have support for parallel loops or other similar operations in the standard library, the Rayon library has gained significant traction, and it seems to be the de facto standard for this type of work in Rust. Lastly, while multithreading and parallelism are part of the selling points of the Go programming language, this support comes in the form of goroutines and channels. The language does not have built-in support for parallel loops. Unlike Rust, there doesn't seem to be a clear leader in this space for Go. We chose to use [8].

The fastest version of n-body in both Julia and Rust wasn't an acceptable starting point as it uses an approach that stores all the distance pairs. There are only 10 of these when there are only five bodies, but we want to consider simulations with 10,000 and 100,000 bodies, which makes this approach unacceptably memory intensive. It also doesn't lend itself well to parallelization.

We did draw some inspiration from the fastest version of the Julia code. That code uses immutable `structs`. We wanted to see if the immutability allowed the Julia JIT to do extra optimizations, so we implemented two versions of this code in Julia, one with mutable `structs` and one with immutable `structs`. While these two versions had a very different structure to the code, their benchmark results were equal within the error bars. Therefore, we only present the timing results of the mutable version. Both versions can be found in the GitHub repository.

One other element worth noting is that while the safety features of Rust can make it a little more challenging to create working code, they do truly help with preventing errors. There was a period of time when we had created race conditions in the implementations in a few languages by merging the loop that updates positions into the loop that aggregates accelerations. This can cause some forces to be calculated with new positions instead of old ones. This creates a subtle bug as the differences between positions before and after a time step are generally small. But it breaks the symplectic nature of the integrator and would cause the system's overall energy to drift for long integrations. Attempting to create this bug in Rust leads to an error message from the borrow checker, as it requires having an immutable borrow simultaneously as a mutable one [9].

2.2 Results

The benchmarks were run on a Linux workstation with two Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz processors and 64 GB of RAM. To get consistent timing values across all the languages involved, we used the Linux `time` command instead of timing commands built into any of the languages. This potentially provides a small penalty to Java and Julia as they use a JIT instead of compiling to a native machine executable. This penalty would be most obvious in the shortest-running benchmarks.

We ran simulations with 10,000 and 100,000 particles for ten time steps. This was done seven times for each language using 2, 4, 6, 8, 12, 24, and 48 threads. We present the mean and the standard deviation of those seven runs. The timing results are shown in Fig. 1.

For both the smaller and larger simulations, the performance scales very close to linearly up to 12 threads for all the languages. This makes sense, given that the machine used has two 12-core processors. Going from 12 to 24 threads, performance gains were still shown. These gains were close to linear for the larger simulations but distinctly sub-linear for the smaller ones. That jump moves the simulation off of a single processor, making the cache locality worse. That impact is more negative when there isn't as much data. One interesting result is that all the languages see performance gains going from 24 to 48 threads.

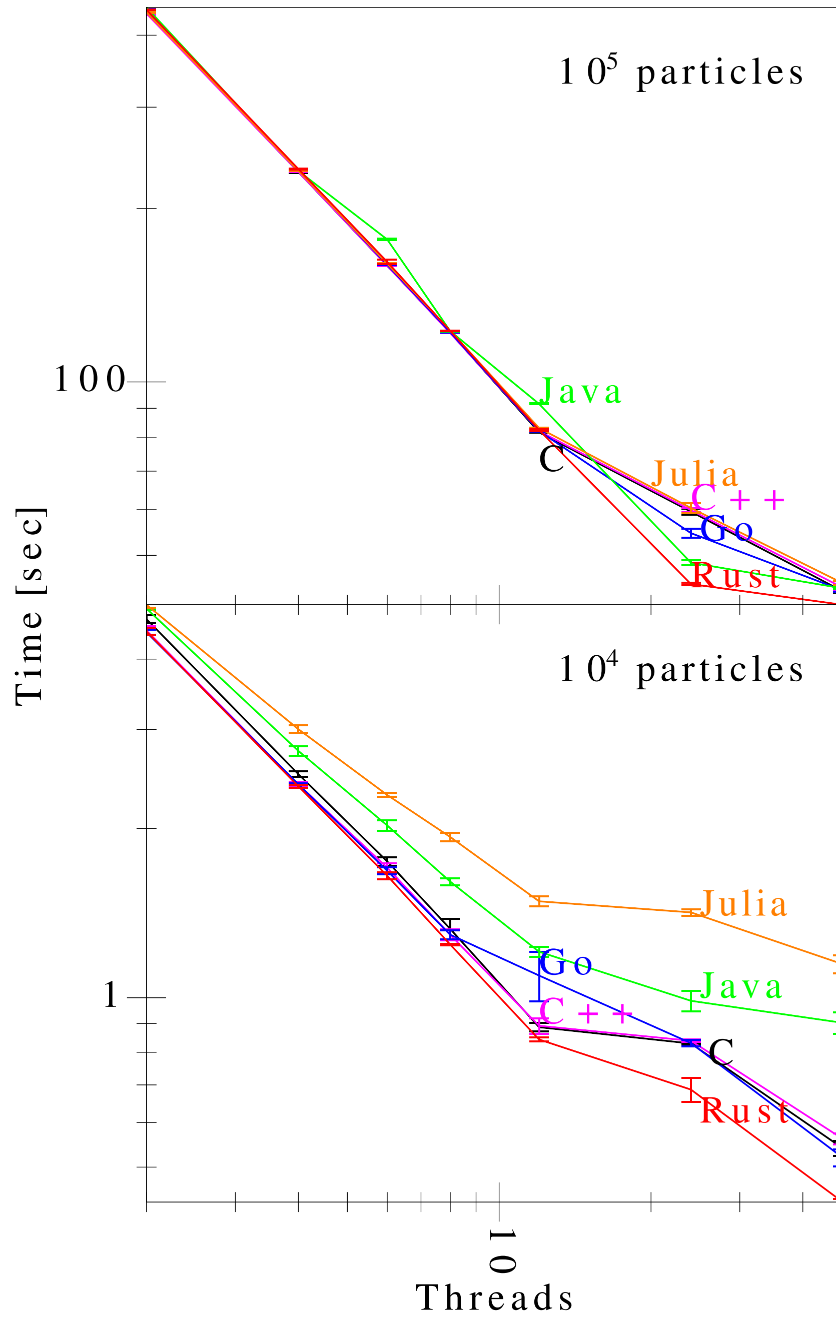

Fig. 1. Run times for ten time steps using the O(N^2) approach. One standard deviation error bars are shown.

This is surprising as "hyperthreading" often doesn't provide speed benefits for numerically intensive applications.

When comparing languages, a few patterns jump out. One is that Rust is the performance winner across the board, especially at the higher thread counts, where it matters most for actual research work. In the smaller simulations, it is somewhat surprising that Julia is the slowest by a fair margin. From there, the trends are more what one might expect. Julia is followed by Java, with Go, C, and C++ having similar performance and Rust being the fastest. The slow performance for Java and Julia at high thread counts could be largely the result of the short total times and the fact that they have JITs that give them longer startup times. However, the patterns are far less clear for the larger simulations, where startup time will matter less. Two main things stand out in the timing results of the larger simulations. First, the difference between the performance of the languages is remarkably small at every thread count except 24 threads. Second, the JVM/Java performance is erratic. It is the slowest language at 6 and 12 threads but the second fastest at 24.

The key takeaway from the results is that for raw number crunching across multiple threads, all these languages have remarkably similar performance as long as there is enough work to be done, enough memory being used, and the JIT startup time isn't a significant fraction of the workload. While Rust is the fastest, it is generally by less than 10%, even compared to Java and Julia with 10^5 particles.

3 kD-Tree Approach

Most research simulations need significantly more than 100,000 particles. The first approach's $O(N^2)$ scaling makes that infeasible, even with the introduction of parallelism. This motivated using a kD-tree in [15]. Tree-based structures have a long history in numerical simulations going back to Barnes and Hut [5]. The original work used an octree. The idea is that gravitational interactions with distant groups of particles can be modeled as a force from the collection of particles instead of having to calculate the forces from each of the individual particles. This gives $O(N \log N)$ scaling behavior. This approach has been applied generally to astrophysical simulations ranging from cosmology [18, 19] to planetary rings [14, 16] and allows for simulations involving many millions to even billions of particles.

The kD-tree approach is also more interesting with the inclusion of Julia. The Benchmark Game doesn't include a kD-tree or any other spatial data structures used for number-crunching benchmarks. However, the kD-tree is, in many ways, a fancy binary tree, and the Benchmark Game does include a binary tree benchmark [1]. It is worth noting that Julia does quite poorly in this benchmark. While Julia is one of the fastest languages in the n-body benchmark, it is more than 14 times slower than the top languages in the binary-tree benchmark. Note that Python is more than 34 times slower than the fastest languages in the binary-tree benchmarks, so Julia still does much better than Python, but

the best Java implementation is only 2.5 times slower than C++ and Rust, so Julia looks more like a slow, scripting language here. This begs the question of whether a kD-tree integrator will perform more like the pure number crunching code or the data structure code.

3.1 Implementations

The C, C++, Go, Java, and Rust implementations began with the versions used in [15]. The Rust version was slightly updated to use an `enum` for the tree nodes. Then, all the loops over the particles in the main function were updated to work in parallel. The parallelization techniques were the same as described in section 2.1.

The Julia version of the code was built as a conversion from the Java version with a few modifications. The tree nodes in Julia were implemented as an abstract type with two subtypes for the leaves and the internal nodes. The functions to traverse the tree use Julia's multiple dispatch. As described above, we have seen that Julia is able to provide more optimization when it knows more about the types. Hence, we annotated all functions with types for both inputs and outputs.

Section 1 mentioned that thanks to the borrow checker, the Rust implementation of multithreading could catch bugs accidentally added in other languages. Unfortunately, something of the opposite is true in Julia. The creators of Julia decided to follow in the footsteps of Fortran instead of C and use 1-indexed arrays. While there are certain arguments that this makes sense for a language aimed at scientists and others who aren't professional programmers, it also leads to subtle bugs when converting code from languages that are 0-indexed, which is the case for all the other languages we looked at.

As was mentioned earlier, the code to construct the kD-tree was not parallelized. This is another $O(N \log N)$ operation that happens once each time step and is completely sequential in the current implementations. The full implementations for all the languages can be found at https://github.com/MarkCLewis/MultiLanguageKDTree in the `Parallel` subdirectory.

3.2 Results

These benchmarks used the same machine as described above. The same thread counts were used, but the particle counts were bumped up by a factor of 10 to 100,000 and 1,000,000 particles for the two simulation sizes. The timing results are shown in Fig. 2. Note that the results for Julia are not complete for 1 million particles.

Unsurprisingly, these results roughly mirror what we found with the sequential kD-trees in [15] for the languages that had been part of the original study. As before, we see roughly linear scaling up to 12 threads. Performance continues to improve all the way up to 48 threads, though it is distinctly sub-linear. We see that Rust is the fastest, with C and C++ tied and taking nearly 50% longer

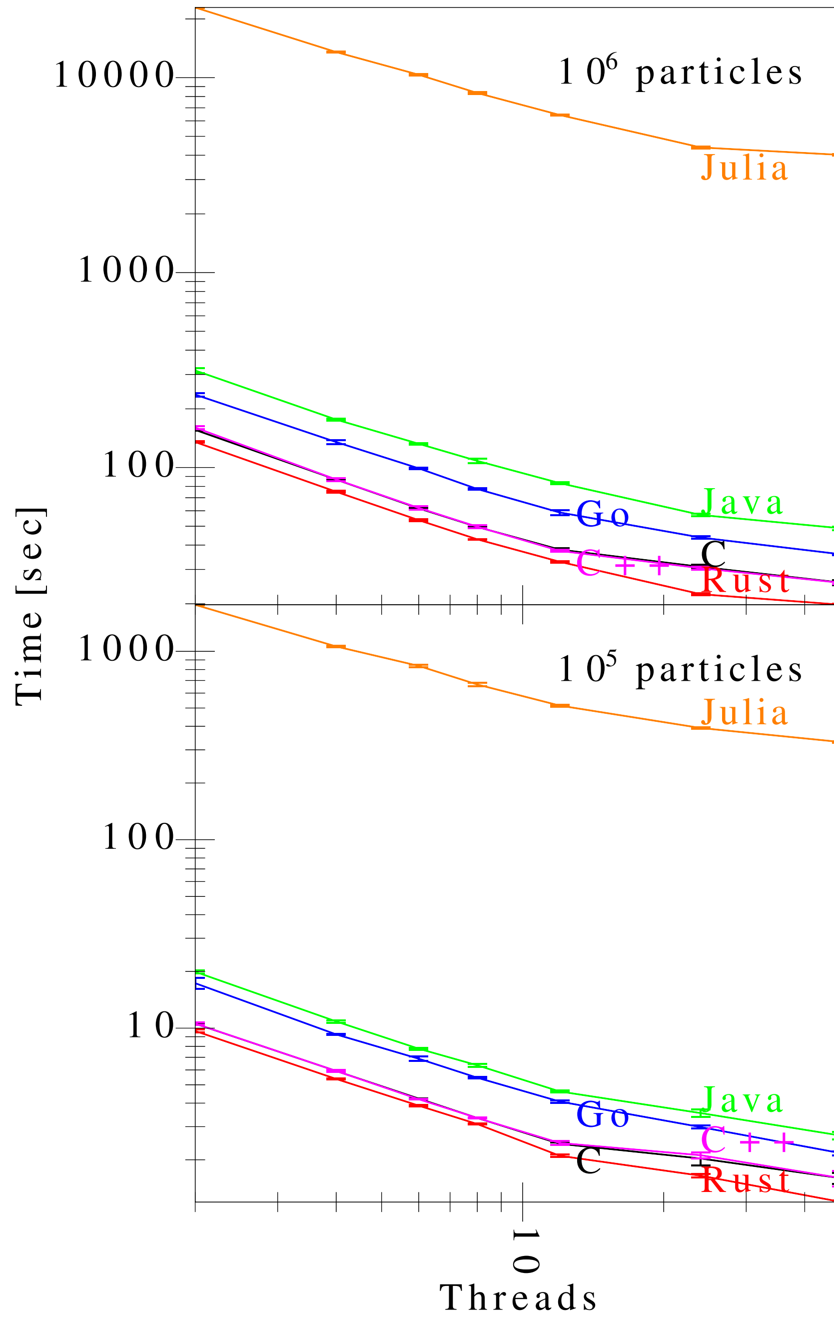

Fig. 2. Run times for ten time steps using the kD-tree versions. One standard deviation error bars are shown.

than Rust. Go is slightly faster than Java, with both taking nearly twice as long as Rust to complete the simulations.

The surprising result here is Julia's remarkably poor performance. In these tests, Julia is as far behind the Rust implementation as Python was in the sequential testing. Indeed, Julia is so slow we did additional testing to ensure there wasn't a bug. We added code to the Julia and Java versions to count how many times various operations were performed. The counts between the two matched. So, the Julia code does the same number of gravity calculations as the other versions. It is simply much slower at doing them.

Another factor that could be playing a role is that the multithreading in Julia appears to have issues with load balancing in this code. Watching the load bar on the machine as the benchmarks are running, we see it go up and down many times during a time step. Depending on the implementation details, each time step has two or three parallel loops. With the other languages, the load bar is observed to jump to roughly the number of threads used in that run and stay close to that level until the end of the time step when it hits a period of sequential processing for the tree construction.

This also shows up in the output of `time`, which includes lines for `real` and `user`. The `@Threads` macro in Julia is not doing as good a job of evenly distributing the work across threads as other implementations.

A second surprising result is how well Rust with Rayon performs compared to C and C++ with OpenMP. OpenMP is a well-established standard used across industry and in most HPC workloads. Despite this, Rust opens its biggest performance lead over C and C++ when the thread count is high. This trend holds for both simulation sizes we considered.

The main message from these results is somewhat different from that of the first round of benchmarks. When we include a data structure and the number of particles scales up, the choice of language becomes much more significant. Instead of having results within 10% of each other, here we see a solid factor of two difference in the performance between languages. Even the difference between Rust and C/C++ is large enough it should make people consider using Rust for large, expensive workloads.

4 Conclusions and Future Work

If all you are doing is large amounts of double arithmetic, with little to no data structures required, all of the languages looked at in this paper provide similar performance, and they all scale well with the addition of more threads. However, if your work involves more complicated data structures, there are clear benefits to using systems languages.

In particular, Rust stands out as the best overall choice for performance in these benchmarks, even compared to C and C++. This isn't too unexpected, but what is truly impressive is that Rust, with Rayon, can beat out C and C++ using OpenMP. Indeed, Rayon seems to scale slightly better when the thread count on our machine is maxed out.

The most disappointing result is how poorly Julia performs when we are using a kD-tree. While Julia is highly competitive with the other languages for pure number crunching on larger systems, its performance with the kD-tree was so slow that it would be on par with Python. At least with Julia, some performance can be gained by multithreading, which would provide no benefit in current versions of Python. However, even running with 48 threads, the Julia version with the kD-tree was still slower than Java would be with a single thread.

The main element missing from this work, which remains a task for future work, is to parallelize the building of the kD-tree. The force calculations were low-hanging fruit as they are embarrassingly parallel. However, parallelizing the kD-tree construction is much more challenging. It is a recursive process. In the lower parts of the tree, the parallelization can be done by distributing each subtree to a different thread. However, each tree level requires $O(N)$ work to partition the particles, so multithreading the upper levels is just as important as multithreading the lower levels. We are still working on finding efficient ways to do this, especially with mechanisms that don't have significant memory overhead and that work well across multiple languages.

Acknowledgment

This work has been supported by NSF Award ID 2206306, "Unraveling the mysteries of small-body ring systems through numerical modeling". We'd like to thank Zachary Taylor for suggesting including Julia in this work.

References

1. Anon: binary-trees - which programs are fastest (benchmarks game) (2024), https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/binarytrees.html
2. Anon: The computer language 24.04 benchmarks game (2024), https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
3. Anon: The julia programming language (2024), https://julialang.org/
4. Anon: n-body - which programs are fastest (benchmarks game) (2024), https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/nbody.html
5. Barnes, J., Hut, P.: A hierarchical o (n log n) force-calculation algorithm. nature **324**(6096), 446–449 (1986)
6. Couto, M., Pereira, R., Ribeiro, F., Rua, R., Saraiva, J.: Towards a green ranking for programming languages. In: Proceedings of the 21st Brazilian Symposium on Programming Languages. pp. 1–8 (2017)
7. Engheim, E.: Julia as a Second Language: General purpose programming with a taste of data science. Simon and Schuster (2023)
8. Graves, D.: dgracesq/go-parallel - go parallel construct inspired by openmp (2022), https://github.com/dgravesa/go-parallel
9. Matthews, B.: Code Like a Pro in Rust. Simon and Schuster (2024)

10. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: Energy efficiency across programming languages: how do energy, time, and memory relate? In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. pp. 256–267 (2017)
11. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: Ranking programming languages by energy efficiency. Science of Computer Programming **205**, 102609 (2021)
12. Perkel, J.M., et al.: Julia: come for the syntax, stay for the speed. Nature **572**(7767), 141–142 (2019)
13. Perminov, A., Kuznetsov, E.: The orbital evolution of the sun–jupiter–saturn–uranus–neptune system on long time scales. Astrophysics and Space Science **365**, 1–21 (2020)
14. Richardson, D.C., Quinn, T., Stadel, J., Lake, G.: Direct large-scale n-body simulations of planetesimal dynamics. Icarus **143**(1), 45–59 (2000)
15. Rotter, J., Lewis, M.C.: N-body performance with a kd-tree: Comparing rust to other languages. In: 2022 International Conference on Computational Science and Computational Intelligence. pp. 458–463. https://american-cse.org/csci2022-ieee/pdfs/CSCI2022-2lPzsUSRQukMlxf8K2x89I . . . (2022)
16. Salo, H., Ohtsuki, K., Lewis, M.C.: Planetary Ring Systems, chap. Computer simulations of planetary rings, pp. 434–493. Cambridge University Press (2018)
17. Sells, R.: Julia programming language benchmark using a flight simulation. In: 2020 IEEE Aerospace Conference. pp. 1–8. IEEE (2020)
18. Stadel, J., Wadsley, J., Richardson, D.C.: High performance computational astrophysics with pkdgrav/gasoline. In: High Performance Computing Systems and Applications. pp. 501–523. Springer (2002)
19. Stadel, J.G.: Cosmological N-body simulations and their analysis. University of Washington (2001)