Beyond Classical Parallel Programming Frameworks: Chapel vs Julia

Rok Novosel
Faculty of Computer and Information Science, University of Ljubljana
Večna pot 113, 1000 Ljubljana, Slovenia
novosel.rok@gmail.com

Boštjan Slivnik[1]
Faculty of Computer and Information Science, University of Ljubljana
Večna pot 113, 1000 Ljubljana, Slovenia
bostjan.slivnik@fri.uni-lj.si

Abstract

Although parallel programming languages have existed for decades, (scientific) parallel programming is still dominated by Fortran and C/C++ augmented with parallel programming frameworks, e.g., MPI, OpenMP, OpenCL and CUDA. This paper contains a comparative study of Chapel and Julia, two languages quite different from one another as well as from Fortran and C, in regard to parallel programming on distributed and shared memory computers. The study is carried out using test cases that expose the need for different approaches to parallel programming. Test cases are implemented in Chapel and Julia, and in C augmented with MPI and OpenMP. It is shown that both languages, Chapel and Julia, represent a viable alternative to Fortran and C/C++ augmented with parallel programming frameworks: the programmer's efficiency is considerably improved while the speed of programs is not significantly affected.

2012 ACM Subject Classification Computing methodologies → Parallel programming languages

Keywords and phrases parallel programming languages, Chapel, Julia

Digital Object Identifier 10.4230/OASIcs.SLATE.2019.12

Supplement Material The sources are at `https://github.com/novoselrok/parallel-algorithms`.

Acknowledgements The authors want to thank Janez Pintar for valuable discussions at the beginning of this work.

1 Introduction

To implement a parallel algorithm or to write a parallel application, most programmers would use Fortran or C/C++ and a parallel programming framework that best suits the target parallel computer's architecture. Hence, MPI and OpenMP would be used for programs designed to run on distributed and shared memory computers, respectively, or OpenCL/CUDA if the computation must run on a GPU. In 2019 this remains de facto approach to parallel programming even though parallel programming languages have been around for decades.

Among parallel programming languages, Fortran stands out as in its 2018 version it includes a wide range of constructs supporting data parallelism and concurrency. Otherwise, many languages died away, e.g., SISAL, ZPL and Fortress, or faded into obscurity, e.g., X10. Nevertheless, it has always been claimed that languages supporting parallel programming will one day boost parallel application development [8], and were therefore studied and analyzed [6, 16].

[1] The corresponing author.

In this paper writing parallel programs in two programming languages, namely Chapel and Julia, is considered. Chapel has been developed at Cray, a traditional supercomputer manufacturer, and has been intended for increasing supercomputer productivity and parallel programming from the start. With its explicit support for declaring a topology of processor unit the program is to be run on it is a border case of a domain-specific language. Julia is rather different. Designed for programming high-performance computational science, it is definitely a general programming language. It is a dynamic language with garbage collection and uses just-in-time compilation. Hence, a list of characteristics usually not associated with the fastest possible execution.

Chapel and Julia are compared one against the other and against the combination of C and a selected parallel programming framework. The comparison is performed by implementing the algorithms for using three selected problems (Section 2) and evaluating the programming process and the final result (Section 3). Although this approach is far from new [12, 4], it has been called for more studies like this as no better methodology for comparing programming languages is available today [26].

2 Test Cases

2.1 *k*-Means Clustering

The k-means algorithm [21], also known as Lloyd's algorithm, is a method of finding clusters in a dataset. Its aim is to organize n points in d-dimensional space into k clusters: in the end, each point should belong to the cluster with the nearest mean (centroid) according to the Euclidean distance. The algorithm starts by creating initial clusters and then iteratively updating them. There are multiple ways of initializing the clusters: despite sophisticated methods like kmeans++ [2], we choose random points from the dataset to serve as initial clusters. Each update performs two steps:

1. Compute the centroid of a cluster.
2. Reassign points to be in the cluster with the nearest centroid.

The iterative procedure ends when no points exchange clusters, or it reaches a maximum iteration limit. Fig. 1 illustrates the iterations of the k-means algorithms where $n = 1000$ and $k = 3$.

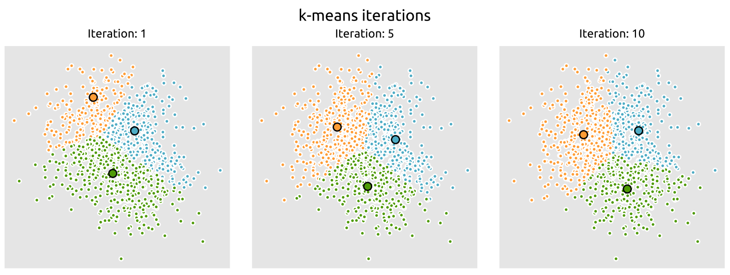


Figure 1 Iterations of the k-means clustering where $n = 1000$ and $k = 3$.

The algorithm is parallelized by distributing the points among the parallel workers and then independently calculating the nearest cluster. We compute global centroids by combining the local centroids of each worker. We decided to parallelize the k-means algorithm since it serves as an example of the map-reduce pattern [11]. It can be generalized to other machine learning algorithms that can be parallelized using the same pattern [10].

2.2 Samplesort

Sorting is a ubiquitous operation in computer science that has to handle large quantities of data in a short amount of time. That is why efficient parallel implementations are necessary. Samplesort [14] provides a good benchmark for testing whether new parallel languages are able to handle the mentioned constraints.

Samplesort is a divide-and-conquer sorting algorithm. It addresses the limitations of the Quicksort algorithm by allowing more than one partitioning element. Partitioning elements are determined by sampling the elements of the input array. Samplesort partitions the elements into m bins by constructing a $m \times m$ matrix where m is the number of parallel workers. Each column in the matrix corresponds to a bin. The entire algorithm consists of the following steps:

1. Sample $m - 1$ elements from the input array.
2. Distribute elements among workers.
3. Each worker partitions the local array into m bins according to the selected samples.
4. The bins matrix is re-distributed, and each worker takes ownership of a column.
5. Each worker sorts the elements in its column.
6. Finally, the columns are merged to get the sorted array.

The choice of the actual sorting algorithm in the fifth step is arbitrary. We used the Quicksort algorithm. Fig. 2 demonstrates sorting 20 input elements with three parallel workers. Each of the differently colored rows or columns indicate that a worker can execute them in parallel.

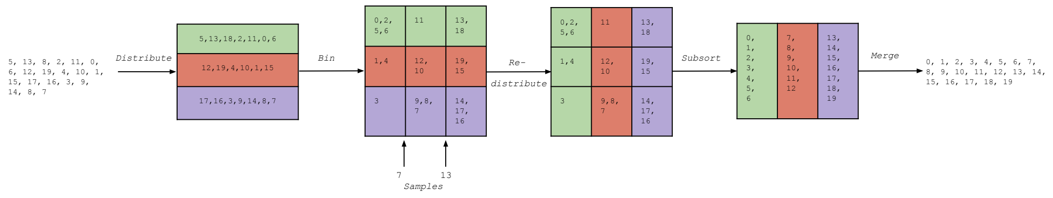


Figure 2 Samplesort with 20 elements and 3 parallel workers.

2.3 *n*-Body Simulation

The n-body problem requires that positions and velocities of n interacting particles is computed, usually in discrete time intervals. It is regularly used when evaluating parallel languages and frameworks [23, 25] as it was classified as a parallel "dwarf" [3]. The solution to the n-body problem is found by simulating the behavior of the particles.

The simplest way of simulating particle behavior is by calculating the effect of all other particles on a specific particle. This results in a $O(n^2)$ algorithm, which is computationally expensive for a large number of particles. The key to lowering the computational cost is to group nearby bodies and treat them as a single body. If it is far enough, we can approximate the gravitational effect of the group by using the center of its mass. The Barnes-Hut algorithm [5] achieves this by using the octree data structure [22] to split the particle space into cubic cells. Once the octree is constructed, the algorithm traverses the tree structure for each body and calculates the gravitational effect of each node. If the bodies contained in the node sub-tree are sufficiently far away, the effect is approximated by their center of mass. Otherwise, the algorithm reaches the leaf nodes which contain the original bodies. In

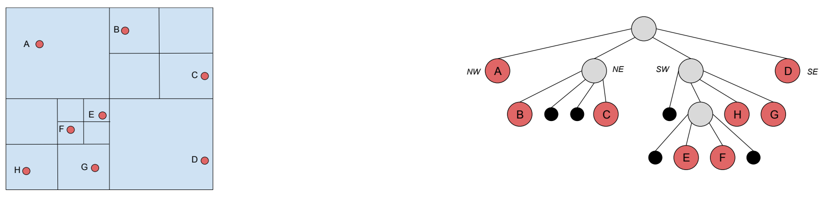


Figure 3 A 2-dimensional representation of the particle space and the corresponding Barnes-Hut quadtree: particle space divided into quadrants (left) and Barnes-Hut quadtree (right).

Fig. 3(a), we used 2-dimensional particle space to simplify the explanation. A 2-dimensional particle space is divided into quadrants and the Barnes-Hut algorithm constructs a quadtree as illustrated in Fig. 3(b).

We used different parallelization strategies for shared and distributed memory implementations. For shared memory, we statically divided the 3rd level of the octree (containing 64 nodes) among the threads. Each thread computes the necessary subtrees and the main thread combines the subtrees into the Barnes-Hut octree. For the distributed implementation we followed the algorithm outlined in [24]. Its main advantage is that it can balance the amount of work for each worker using orthogonal recursive bisection [13]. The algorithm works as follows:

1. Decompose the particle space using orthogonal recursive bisection so that each cell contains equal amount of work.
2. Each worker owns one cell and the particles within it.
3. Each worker builds a local octree from the particles (locally essential tree).
4. Next, it sends its local octree nodes to any other worker that might need them.
5. It receives the necessary octree nodes from other workers and inserts them into his octree.
6. Each worker can now compute the positions and velocities independently.

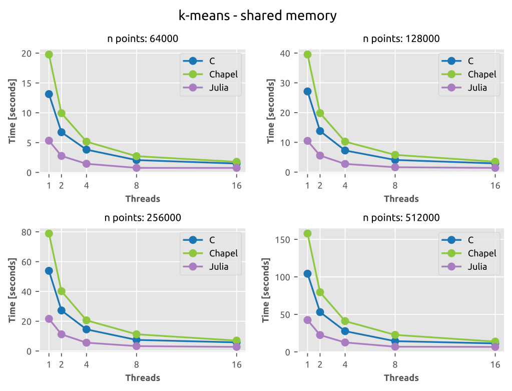
3 Evaluation

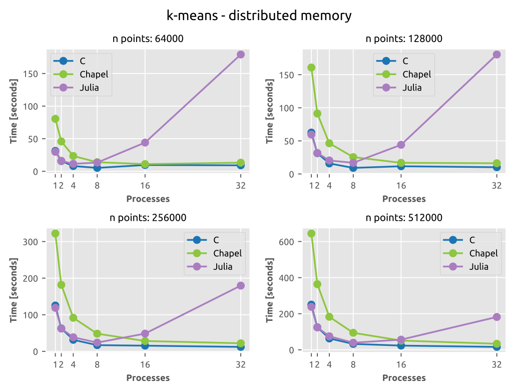
We evaluated the parallel capabilities of Chapel and Julia by implementing the test cases from the previous section. For each test case we implemented an algorithm targeted for a shared and a distributed memory computer. We used the programming language C with OpenMP for shared memory and MPI for distributed memory as the baseline for comparison. Since Julia is using JIT compilation, we also discarded the first two execution times to properly “warm-up” the JIT compiler.

We used Chapel version 1.18 and Julia version 1.1. For shared memory, we used Ubuntu 18.04, AMD Ryzen 7 2700X (8 cores, 16 threads, and 3.7GHz) processor and 16GB of RAM. For distributed memory, we used 32 HP DL160 G6 servers, each running Ubuntu Server 11.04 with Intel Xeon 5520 processor and 6GB of RAM.

3.1 *k*-Means Clustering

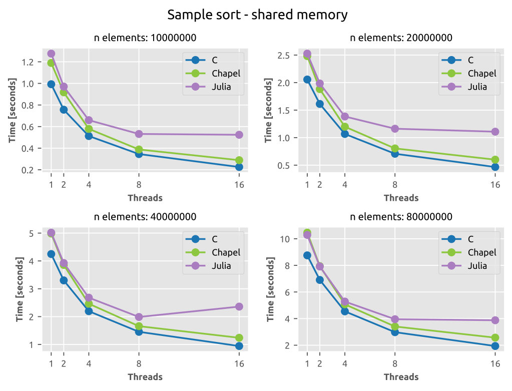
We generated 4 datasets for evaluation purposes. Each dataset contains 256 clusters with 128-dimensional points. The results in Fig. 4 indicate that the performance bottleneck in the k-means clustering is the computation of the Euclidean distance. Julia has a highly optimized linear algebra module based on the LAPACK library [1]. This is the main reason why we were able to outperform the C shared-memory implementation. The Julia distributed

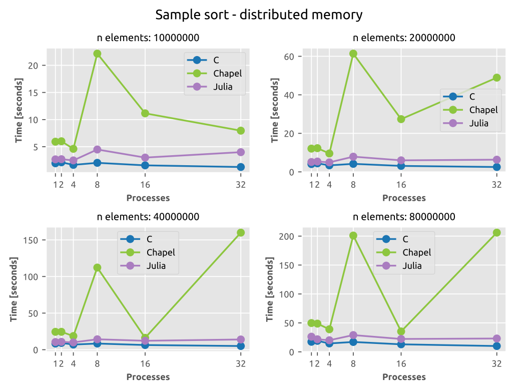




(a) k-means clustering benchmark for the shared memory implementation. **(b)** k-means clustering benchmark for the distributed memory implementation.

Figure 4 k-means clustering benchmarks.





(a) Samplesort benchmark for the shared memory implementation. **(b)** Samplesort benchmark for the distributed memory implementation.

Figure 5 Samplesort parallel benchmarks.

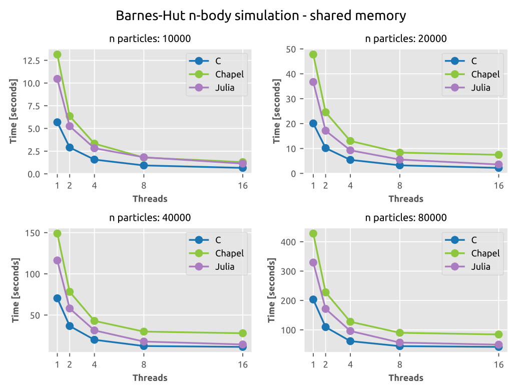
memory implementation suffered from a significant overhead when spawning remote processes and communicating through remote channels. Based on Hoare's CSP [15], a channel is a FIFO data structure enabling remote processes to send and receive data.

Increasing the number of processes only exacerbated the problem. To eliminate the overhead we would have to benchmark a much larger dataset.

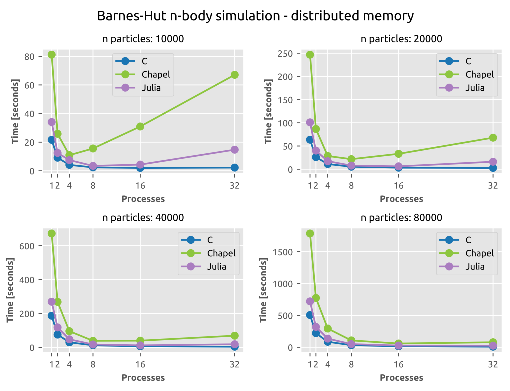
3.2 Samplesort

Sorting efficiency depends highly on the input elements themselves. Therefore, we could not pre-generate the input arrays. For each input size, we generated 100 random arrays and measured the execution time for each array. The final result was the average of all the execution times. The results are in Fig. 5.

In Julia, we add multi-threading by prefixing a for loop with the `@threads` macro. The macro wraps the body of the for loop in a closure and splits the iterations between the available threads. Using variables captured in a macro closure is a performance concern and is still an open issue in Julia [19]. The recommended solution, for now, is to extract the body



(a) Barnes-Hut n-body simulation benchmark for the shared memory implementation.



(b) Barnes-Hut n-body simulation benchmark for the distributed memory implementation.

Figure 6 Barnes-Hut n-body simulation benchmarks.

of the for loop in a separate function and thus bypass the macro closure. It is not a perfect solution and that is why we still see a deviation from the C and Chapel results in shared memory implementations.

The Chapel distributed memory implementation caused the most issues. The standard way of diagnosing distributed implementations is by using the `CommDiagnostics` module, which outputs all the implicit Chapel communication. It did not output any unnecessary or unintended communication. Our reasoning is that the Chapel compiler and the underlying GASNET [7] communication library are not able to optimize the transfer of large arrays during sorting. In spite of the mentioned issues, we found that using Chapel domains we were able to elegantly solve the samplesort problem.

3.3 *n*-Body Simulation

For the n-body simulation, we generated equally distributed particles in a pre-defined cube. The results are in Fig. 6. In the shared memory implementations we observed consistent results for all test cases. The execution times of C with OpenMP and Julia both running with 16 threads are almost equal. It would be interesting to see if Julia could outperform C and OpenMP if both are given more threads.

The algorithm we chose for the distributed memory implementations was designed for the SPMD (single program, multiple data) parallel model. As such, MPI was the ideal target when the algorithm was designed. With Julia and Chapel, we were forced to replicate the SPMD behavior. In Julia, it was relatively easy to replicate the SPMD model using channels. On the contrary, Chapel does not have a way for processes to directly communicate with each other. All communication has to be done through distributed global arrays. We used sync variables to prevent multiple processes from altering the same element in the array. Sync variables are roughly analogous to mutex locks in other languages with the exception that a sync variable is itself a lock. Any basic type (integer, float, etc.) can become a sync type by prefixing it with the `sync` keyword. Chapel does not support sync variables on complex types like arrays or objects. For complex types, we had to create two arrays one containing values and one containing sync variables. Considering the orchestration behind the Chapel implementation it is not surprising that for a small amount of data and a large number of processes it does not perform well. Increasing the amount of data hides the overhead from sync variable contention.

It is also important to note that both Julia and Chapel do not support fixed-sized arrays out-of-the-box. Using regular (dynamic) arrays was a huge performance bottleneck in the n-body simulation and the k-means clustering implementations. We used the `StaticArrays` package in Julia but resorted to using tuples in Chapel. Further research is necessary to design n-body simulation algorithms that would better fit Julia and Chapel.

4 Conclusion

In general, we had little to no issues with our C implementations. Once we eliminated all segmentation faults C implementations needed only minor improvements to run efficiently. In contrast, Chapel and Julia implementations were easier to write initially but required a lot of efficiency improvements and fine-tuning afterwards.

Julia provides a small but powerful set of parallel features. The multi-threading module is still marked as experimental, but we were still able to get good results with it. The Julia development team is planning to replace the current multi-threading module with an implementation based on parallel depth-first scheduling [9]. The main two issues with Julia were type instability and unexpected memory allocations. If Julia is not able to infer the type of variable it resorts to runtime type checking which adds significant overhead. The solution was to always annotate all variables with types to avoid runtime type checking. The Julia standard library contains multiple macros for code inspection and benchmarking. Examples include `@code_warntype` which outputs variables with ambiguous types and `@time` which outputs execution time and total memory allocated. Julia provides a web page [20] containing performance tips that have to be followed rigorously if performance is an issue.

Chapel provides a broad spectrum of parallel functionality. It can serve as an excellent introductory language for parallel and distributed programming courses. It includes enough the high-level features from other languages so that it immediately feels familiar to novices. Parallel concepts can be easily demonstrated using built-in constructs like domains, sync variables, parallel iterators, etc. Chapel itself is still under heavy development. Developers are currently trying to implement as many parallel features as possible. Consequently, the performance of Chapel programs is not as important right now. That causes it to lag behind C and Julia in our benchmark tests. During the development of this paper, we also found multiple bugs and performance issues in the compiler [17, 18]. We also experienced that the Chapel compiler is fairly slow when compiling large applications. As an example, our distributed n-body simulation takes roughly 1 minute to compile. During the implementation process, this prevented us from quickly iterating and examining new solutions.

The authors realize that the definitive comparison of languages for parallel computing would require a randomized controlled trial as advocated in [26]. Nevertheless, the results presented here can be understood as a justification for carrying out such experiment.

References

1 E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK *Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999.

2 David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

3 Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, 2006.

4 Henri E. Bal. A comparative study of five parallel programming languages. *Future Generation Computer Systems*, 8(1–3):121–135, 1992.

5 Josh Barnes and Piet Hut. A hierarchical O($N \log N$) force-calculation algorithm. *Nature*, 324(6096):446, 1986.

6 A. P. W. Böhm and R. R. Oldehoeft. Two Issues in Parallel Language Design. *ACM Transactions on Programming Languages and Systems*, 16(6):1675–1683, 1994.

7 Dan Bonachea and P Hargrove. GASNet Specification, v1.8.1. `https://gasnet.lbl.gov/dist/docs/gasnet.html`, 2017.

8 Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, pages 66–75, 2004.

9 Shimin Chen, Phillip B Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C Mowry, et al. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115. ACM, 2007.

10 Cheng-Tao Chu, Sang K Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y Ng. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288, 2007.

11 Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

12 John T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. North Holland, New York, NY, USA, 1992.

13 Geoffrey C Fox. A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube. In *Numerical Algorithms for Modern Parallel Computer Architectures*, pages 37–61. Springer, 1988.

14 W Donald Frazer and AC McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, 17(3):496–507, 1970.

15 Charles Antony Richard Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.

16 Ken Kennedy, Charles Koelbel, and Hans Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 7–1–7–22, 2007.

17 Chapel Programming Language. Chapel Issue: Performance issues when running n-body simulation. `https://github.com/chapel-lang/chapel/issues/11333`. Accessed: 2019-04-24.

18 Chapel Programming Language. Chapel Issue: Using with clause in coforall loop for distributed k-means. `https://github.com/chapel-lang/chapel/issues/12006`. Accessed: 2019-04-24.

19 Julia Programming Language. Julia Issue: performance of captured variables in closures. `https://github.com/JuliaLang/julia/issues/15276`. Accessed: 2019-04-24.

20 Julia Programming Language. Julia Performance Tips. `https://docs.julialang.org/en/v1/manual/performance-tips/index.html`. Accessed: 2019-04-22.

21 Stuart Lloyd. Least squares quantization in PCM. *IEEE transactions on information theory*, 28(2):129–137, 1982.

22 Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

23 Lars Nylons, Mark Harris, and Jan Prins. Fast n-body simulation with CUDA. `https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/nbody/doc/nbody_gems3_ch31.pdf`.

24 John K Salmon. *Parallel hierarchical N-body methods*. PhD thesis, California Institute of Technology, 1991.

25 Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, 1992.

26 Andreas Stefik and Stefan Hanenberg. Methodological Irregularities in Programming-Language Research. *Computer*, 50(8):60–63, 2017. `doi:10.1109/MC.2017.3001257`.