Multicore and Many-core Architectures in a Parallel O(N log N) Friends-of-Friends Algorithm for Astronomical Object Classification

Otavio Migliavacca Madalosso[a], Andrea Schwertner Charão[a 1],
Haroldo Fraga de Campos Velho[b], Renata Sampaio da Rocha Ruiz[b]
João Vicente Ferreira Lima[a]

[a]Universidade Federal de Santa Maria, RS, Brazil
[b]National Institute for Space Research, São José dos Campos, SP, Brazil

Abstract

Friends-of-Friends (FoF) algorithms have been used for classifying galaxies and clusters of galaxies for dark matter halos from N-body simulations. Taking into account the large amount of data usually processed by these algorithms, it is essential to develop efficient computational techniques to extract meaningful information in a reasonable time. In this work, we present an implementation of a FoF algorithm with complexity of $O(N \log N)$. We compare its performance against an existing $O(N^2)$ code, then we explore the parallelization of both codes on two types of parallel architectures: a multicore processor and a many-core GPU. In the parallel implementations, the programming models are based on compiler directives, provided by OpenMP and OpenACC standards. The results indicate that our combined approaches are able to achieve significant speedups, even with some limitations related to the work decomposition strategy and the directive-based parallel programming model.

Keywords: Friends-of-Friends algorithm, parallel computer architectures, astronomy.

1. Introduction

Computational analysis of astronomical data and N-body simulations have been used to promote various advances in the understanding of relevant issues in astrophysics. Such algorithm-based approaches play a key role in the study of cosmic evolution, on topics such as the distribution of dark matter in a large scale, the formation of halos of dark matter and the formation and evolution of galaxies and clusters of galaxies [4, 2, 6].

[1]E-mail Corresponding Author: andrea@inf.ufsm.br

Data from simulations and data from astronomical observatories produce large data sets that need to be manipulated and analyzed. Taking into account such amount of data, it is essential to develop efficient computational techniques to extract meaningful information from a given data source, in a reasonable time. In a previous work, Ruiz et al. [10, 9, 12] implemented a parallel Friends-of-Friends algorithm for classifying galaxies and clusters of galaxies for dark matter halos from N-body simulations. Such implementation achieved a significant speedup when running on a 27-node computer cluster, using the MPI (Message Passing Interface) standard for communication among the processes distributed over the computing nodes.

The original, sequential Friends-of-Friends algorithm implemented by Ruiz et al. has a high computational cost with $O(N^2)$ time complexity. Some authors [11] suggest that the computational cost of this type of method can be reduced in a Friends-of-Friends algorithm with lower, $O(N \log N)$ complexity. One may also consider the hybridity of existing parallel architectures, which currently offer multiple parallel opportunities as, for example, computer clusters with multicore processors using many-core GPUs (Graphics Processing Units) as co-processors.

In this work, we present an implementation of a FoF algorithm with $O(N \log N)$ complexity. We explore its parallelization on two types of parallel architectures: a multicore processor and a many-core GPU. In the parallel implementations, the programming models are based on compiler directives, provided by OpenMP and OpenACC standards.

2. Friends-of-Friends Algorithm

One of the methods used in N-body simulations to determine structures in the Universe is the percolation algorithm named "Friends-of-Friends" (FoF) [5]. The basic idea of this algorithm is as follows: consider a sphere of radius R around each particle of a given set. If within that sphere exist other particles, they will be considered as belonging to the same halo and be called "friends". Then, we take a sphere around each friend and the procedure continues using the rule "any friend of my friend is my friend". The procedure stops when no new friend can be added to the group. Time complexity for such algorithm is $O(N^2)$, which may be prohibitive for a large number of particles.

The complexity of a FoF algorithm can be reduced using a hierarchical, tree-based approach [11]. In our $O(N \log N)$ FoF algorithm, we use an oct-tree [1] to represent a physical three-dimensional space, i.e., a cube in which the particles are allocated. For each node of the oct-tree, there can be up

to 8 child nodes (or leaves) representing each sub-cube with a fraction of its parent's size. In our implementation, the root node of the tree contains the values of the borders of the three-dimensional space. These strategies reduce the search space and increase the performance of the FoF algorithm.

In the next paragraphs, we describe some key implementation issues for our sequential, $O(N \log N)$ FoF algorithm.

2.1 Clustering

The purpose of the FoF algorithm is to find clusters of particles based on physical proximity, i.e., having a linking length (distance between two particles) within a given threshold. The algorithm takes particle coordinates as input and calculates distances each time it reads a new particle from the input set. This procedure adds a new particle to the tree while linking it to a sub-cube.

As a particle deepens into the tree, the algorithm checks between nodes that are semi-cube neighbors to which the new particle will be allocated. Here lies a problem regarding nodes that are not "populated" (that is, a node that is no leaf), because it does not have linked particles to check for physical proximity. We solve these cases using what we called "border zones".

2.2 Border zones

Border zones were primarily designed to deal with the following case: our hierarchical algorithm could erroneously identify two clusters, i.e. in different sub-cubes, where there is actually a single one, because they are both very close to the physical limits of the semi-cubes.

To deal with this case, we use an array of references to the particles lying on the edges of each sub-cube. These limits are delimited by a difference between the limits of each sub-cube and the threshold radius at which the algorithm is running. This solution gave rise to a new case to be dealt with: a situation when the border zone is greater than or equal to the area of the cube itself, so the entire cube could be considered a border zone. In this case, the tree can stop growing and simply allocate all the nodes in a new array.

2.3 Relabeling

Another issue to be dealt with is when the algorithm links a particle to a cluster, but afterwards it discovers that the particle should belong to another cluster, that is, this new particle is a link between two clusters that

until now were distinct. This issue is solved by a function that "relabels" all particles of the two clusters, so that both unite and start to be treated as a single cluster. To do so, we maintain an array of references to particle clusters.

2.4 Performance of FoF algorithms

We performed experiments to compare the performance of our $O(N \log N)$ FoF algorithm against the original, $O(N^2)$ algorithm. All experiments were executed on a server with one Intel Xeon E5620 2.4 GHz quad-core processor, each core with two hardware threads (HT), 12 GB of DDR3 RAM, running Debian 6 (Squeeze) operating system. In both cases, we used the GNU C++ 4.6 compiler.

To exemplify our findings, we present here the results we obtained with an input sample data set comprising 38761 particles. The expected output of our FoF algorithm is to identify how many clusters with more than one element exist in the sample. Both algorithms achieved the same result: 226 clusters. The original FoF algorithm took 29496 ms to run, while the $O(N \log N)$ algorithm took only 149 ms. This performance gain is due to our hierarchical approach for processing the input data. In these experiments, the algorithms are fully sequential, so they do not take advantage of the parallel architecture.

3. Parallel Implementations

Our $O(N \log N)$ algorithm uses a domain decomposition approach which is suitable for parallel processing, as each sub-cube can be treated independently. The previous, parallel $O(N^2)$ FoF algorithm is also based on domain decomposition [9], but is targeted to distributed parallel architectures, i.e., a cluster of computing nodes interconnected by a high speed network. Current parallel architectures combine multiple parallelism opportunities as in, for example, clusters equipped with multicore processors (usually up to 12 cores) and many-core GPUs (tipically with hundreds or thousands of cores).

Thread-based programming is the mainstream approach to take advantage of multicore and GPU parallel architectures. There are many programming models and tools supporting such abstraction, but they usually expose the underlying parallel architecture to the programmer. Therefore, it is difficult to develop parallel code that performs well on multicore or GPUs, without major modifications. An interesting approach towards this direction is to use directive-based multithreaded parallel programming, as supported by OpenMP and OpenACC standards. While they do require some knowledge

of the parallel architecture, they offer an appropriate model for incremental parallelism of existing applications.

In the next paragraphs, we present and discuss our experiences on parallelizing our FoF algorithm using OpenMP and OpenACC, separately. We focus on our $O(N \log N)$ implementation, but we will also discuss the $O(N^2)$ algorithm with both programming tools, as a means of comparison.

3.1 Multicore Implementation

The OpenMP [3, 8] standard provides a parallel programming model targeted to parallel computer architectures where all processing elements have access to a shared memory. OpenMP programs are thread-based and follow a fork-join model. In such model, a master thread runs sequentially until it reaches a parallel region, where a fork occurs, i.e., a number of threads are created to run in parallel. When all threads finish execution inside a parallel region, the execution flow continues on a single, master thread.

Programming in OpenMP is essentially based on the insertion of compiler directives with clauses and options that guide the compiler to generate parallel code. These directives specify parallel regions, work sharing, synchronization and data access. The OpenMP specification is targeted to programs written in Fortran and C/C++ languages. OpenMP programs are portable across hardware architectures, only depending on the compiler support.

In our sequential $O(N \log N)$ algorithm, there is an out-most loop which guides the construction of the oct-tree. Each iteration is independent and refers to a partition of the computational space. To compute border zones, it is necessary to read data from neighboring sub-cubes, but there is no concurrent write operations and dependencies that could require synchronization. In this case, our OpenMP-based parallelization was straightforward and required only a few extra lines of code. There is a limitation, though: due to the nature of the oct-tree, the out-most loop have a fixed (8 iterations) size, so the parallel version scales up to 8 threads.

Using the computing server described in Section , we performed experiments with our parallel, OpenMP-based $O(N \log N)$ algorithm, in order to evaluate its performance. As we mentioned before, the server is equipped with a total of 8 hardware threads (4 cores with 2 hardware threads each). Our input data set comprised 318133 particles (the same experimental setup presented by Ruiz et al. [10]). Table 1 presents elapsed times for up to 8 threads)and their associated speedups. These results show that our ap-

Table 1: Performance of the OpenMP-based $O(N \log N)$ FoF implementation

# of Threads	Elapsed Time (ms)	Speedup
1 (Serial)	39920	-
2	22131	1.80
4	14858	2.68
8	13433	2.97

proach is able to accelerate executions with the addition of a few lines of code to the sequential version. However, the speedup degrades from 4 threads.

In order to better understand our results, we analyzed the execution of our OpenMP code using a performance profiling tool (Intel VTune Amplifier XE). Figure 1 shows a Gantt chart of an execution with 4 threads and its accompanying CPU usage chart, as provided by the profiler. The darker color represents running threads, so we can depict that there is a significant amount of time where only a single thread is running (lighter color), as the other threads have already finished computations. This is due to load imbalances that arise from the spatial work decomposition approach. It is worth noting that this behavior is highly dependent of the input data set and it could be tackled with a dynamic work balancing strategy. However, this would require significant changes on the algorithm and the parallel implementation.

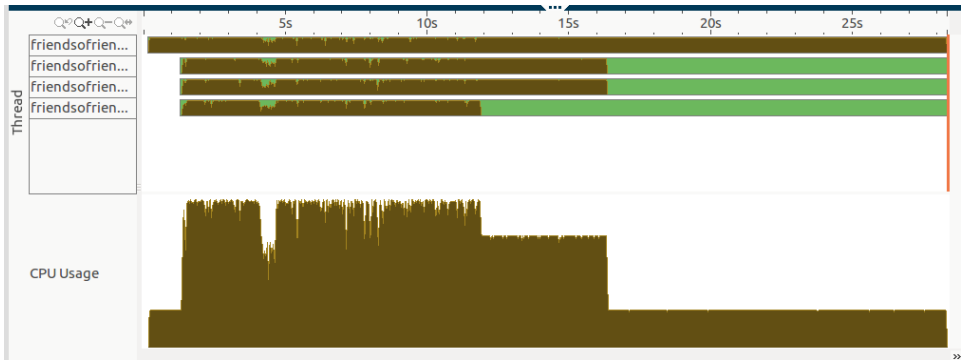

Figure 1: Performance profile of a 4-thread execution (OpenMP)

For comparison purposes, we also applied OpenMP to the previously developed $O(N^2)$ FoF algorithm. The code is mainly structured in nested

Table 2: Performance of the OpenMP-based $O(N^2)$ FoF implementation

# of Threads	Elapsed Time (ms)	Speedup
1 (Serial)	697848	-
2	428542	1.63
4	309329	2.26
8	507271	1.37

loops which update an array (cluster indices) while reading x, y and z coordinates for each input particle. We decided to apply OpenMP directives to the inner-most loop, which has no dependencies between subsequent iterations. This alternative was not expected to fully exploit the parallel architecture, because some computations remain serial. The main goal was to assess how much speedup (if any) could be achieved with minimal additions to the sequential code. In this direction, we could not apply OpenMP on upper loops because it would require significant changes to the program, in order to deal with data dependencies.

Table 2 presents the results we obtained with our OpenMP-based version of the $O(N^2)$ FoF code. The elapsed time is always higher than the $O(N \log N)$ version, but there is also some speedup, as the cost of minimal additions to the sequential code.

3.2 GPU-based Implementation

OpenACC [7] is a standard for parallel programming proposed in 2011, in a partnership between NVIDIA, Cray, Portland Group (PGI) and CAPS Enterprise. Designed to ease programming in hybrid systems consisting of CPU/GPU, this standard uses compiler directives to express parallelism, as well as in OpenMP. It also requires compiler support for recognizing the directives and generating optimized code for different architectures.

The OpenACC programming API comprises directives that specify loops and regions in C/C++ or Fortran code, which will run on a system composed of a host (CPU) and an accelerator device (GPU). These directives let you create high-level programs that implement interactions between host and accelerator, without having to explicitly initialize the accelerator, manage data and transfer programs between host and GPU. All these details are implicit in the programming model and are managed by the compiler that supports OpenACC. Currently, there is only a few compilers that implement the OpenACC standard. The PGI C++ compiler (pgc++ 16.9) is our choice for this work, as it is one of the most up to date OpenACC compilers.

Our OpenACC-based parallelization of the $O(N \log N)$ algorithm focused on the out-most loop of the program, as in OpenMP. The directives for specifying parallel regions are similar to OpenMP, but there is a major difference related to data management. Indeed, there are many options for copying data to/from the GPU, as data transfer are a keystone to high performance, hybrid CPU-GPU programs. This is less of an issue in OpenMP, because all data resides in the main memory. Despite our efforts to add OpenACC directives to our $O(N \log N)$ code, we did not succeed to compile our parallelized code with pgc++, due to our heavy use of `std::vector`, a standard object oriented data structure available in C++. Currently, the PGI C++ compiler do not allow a kernel region, i.e. a parallel region running on GPU, to access such data structure [13]. This limitation was not known when we developed our sequential, C++ code. An alternative would be to rewrite the code to replace the unsupported data structures, but this goes in an opposite sense of the incremental parallelization programming model advocated by OpenMP and OpenACC.

We then decided to focus on applying OpenACC to the previously developed $O(N^2)$ FoF algorithm. This code also uses some C++ constructs, but only adopts native one-dimensional C arrays as data structures. Our approach was to apply OpenACC directives to the same parallel region (innermost loop) as in our OpenMP-based version of the $O(N^2)$ code. The OpenACC compiler default behavior is trying to decide about data transfers to/from a parallel region, so we first tried this approach. However, it was not successful as it led to incorrect results. We then applied data transfer directives (`present`, `copy`, `copy in`) to manually guide the compiler to generate the corresponding code.

After verifying correctness of our numeric results, we then measured execution times in order to assess the performance of our OpenACC-based version of the $O(N^2)$ code. We used the same experimental setup as described in Section 3.1, with the addition of a NVIDIA Tesla M2050 GPU to our server. The average elapsed time was of 446774 ms, which gives a 1,56 speedup from the sequential version which took 697848 ms to run. This is not a significant speedup, but it is worth noting this was obtained at the cost of only a few compiler directives.

4. Conclusions

Friends-of-Friends algorithms usually take large amount of data as input, so it is important to develop efficient computational techniques to reduce processing times. In this work, we explored two approaches to this prob-

lem: tree-based FoF computation to reduce the computational cost of an original algorithm and parallel programming to take advantage of multicore CPUs and many-core GPUs. We also explored a directive-based programming paradigm, which fosters incremental parallelization without significant changes to the existing codes.

Combining these approaches, we have been able to obtain significant speedups, although our spatial decomposition strategy limits the scalability of our $O(N \log N)$ algorithm. The directive-based approach to parallel programming helped us to speed up executions with a few lines of extra code. The OpenACC standard revealed more limitations than OpenMP, but even so, the results support that the directive-based approach may be a good start when parallelizing existing code.

Acknowledgments. The authors gratefully acknowledge financial support from the National Institute of Science and Technology for Astrophysics (INCT-A), who granted the Scientific Initiation scholarship to the first author, and the CNPq, Brazilian agency for research support.

References

[1] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, Dec. 1986.

[2] E. Bertschinger. Simulations of Structure Formation in the Universe. *Annual Review of Astronomy and Astrophysic*, 36:599–654, 1998.

[3] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[4] G. Efstathiou, M. Davis, S. D. M. White, and C. S. Frenk. Numerical techniques for large cosmological N-body simulations. *Astrophysical Journal Supplement Series*, 57:241–260, Feb. 1985.

[5] J. P. Huchra and M. J. Geller. Groups of galaxies. I - Nearby groups. *Astrophysical Journal*, 257:423–437, June 1982.

[6] A. Jenkins, C. S. Frenk, F. R. Pearce, P. A. Thomas, J. M. Colberg, S. D. M. White, H. M. P. Couchman, J. A. Peacock, G. Efstathiou, and A. H. Nelson. Evolution of Structure in Cold Dark Matter Universes. *The Astrophysical Journal*, 499:20–40, May 1998.

[7] OpenACC. The OpenACC application program interface, 2015. Available: http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf.

[8] OpenMP.org. OpenMP specifications, 2015.

[9] R. S. R. Ruiz, H. F. Campos Velho, A. Caretta, C., S. Charão A., and P. Souto R. Grid Environment for Turbulent Dynamics in Cosmology. *Journal of Computacional Interdisciplinary Sciences*, 2:87, 2011.

[10] R. S. R. Ruiz, H. F. Campos Velho, and C. A. Caretta. Parallel algorithm friends-of-friends to identify galaxies and cluster of galaxies for dark matter halos. In *Proceedings...* Workshop dos Cursos de Computação Aplicada do INPE, 9. (WORCAP)., Instituto Nacional de Pesquisas Espaciais (INPE), 2009.

[11] V. Springel, N. Yoshida, and S. D. M. White. GADGET: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, 6(2):79–117, 2001.

[12] E. C. Vasconcellos, R. R. de Carvalho, R. R. Gal, F. L. LaBarbera, H. V. Capelato, H. F. Campos Velho, M. Trevisan, and R. S. R. Ruiz. Decision Tree Classifiers for Star/Galaxy Separation. *Astronomical Journal*, 141:189, June 2011.

[13] M. Wolfe. PGI C++ and OpenACC, 2015. Available: https://www.pgroup.com/lit/articles/insider/v6n2a1.htm.