Chapter 2

Complex simulations on hybrid architectures

2.1 Introduction

The previous part described the method we implement to set the last metric of our benchmark for hybrid architectures. It also presented the walls faced with smoothed particle hydrodynamics and gravitation. We showed that this application meets the communication and computation wall in an irregular behavior context.

We intend to target astrophysical simulation using hybrid architectures. This distributed multi-accelerator SPH and gravitation implementation is called FleCSPH. This is a complex application that, beside of being interesting for our purpose, needs to be accurate on the physics aspect to be used by the domain scientists from LANL.

This section gives details about the FleCSPH framework. We first present FleCSI, the base project in the Los Alamos National Laboratory on which FleCSPH is based. Then, we give details on the implementation itself and the tools we implemented to reach a working and efficient code. We give details on the domain decomposition strategy used with Morton Ordering. The tree traversal algorithm choices are then explained.

After this presentation of the native code using multi-CPU clusters we present our strategies for a multi-GPU implementation. The last section exposes the results for both implementation and the simulations.

A lot of code already exists for SPH simulation and some are already designed for hybrid architectures. The contribution of FleCSPH is, like FleCSI, to provide a transparent tool for domain scientists. Those frameworks provide a bunch of topologies and functions and will handle the load balancing and distribution for the domain scientists. This allows the computer scientists to keep track of last hardware evolution and provide efficient algorithms for them while the physicists/astrophysicists/chemists can focus on the simulations themselves. In this context FleCSPH has to take in charge the tree topologies and will be integrated in FleCSI later on.

2.2 FleCSI

FleCSI[1] [BMC16] is a compile-time configurable framework designed to support multi-physics application development. It is developed at the Los Alamos National Laboratory as part of the Los Alamos Ristra project. As such, FleCSI provides a very general set of infrastructure design patterns that can be specialized and extended to suit the needs of a broad variety of solver and data requirements. FleCSI currently supports multi-dimensional mesh topology, geometry, and

[1] `http://github.com/laristra/flecsi`

adjacency information, as well as n-dimensional hashed-tree data structures, graph partitioning interfaces, and dependency closures.

FleCSI introduces a functional programming model with control, execution, and data abstractions that are consistent both with MPI and with state-of-the-art, task-based runtimes such as Legion[BTSA12] and Charm++[KK93]. The abstraction layer insulates developers from the underlying runtime, while allowing support for multiple runtime systems including conventional models like asynchronous MPI.

The intent is to provide developers with a concrete set of user-friendly programming tools that can be used now, while allowing flexibility in choosing runtime implementations and optimization that can be applied to future architectures and runtimes.

FleCSI's control and execution models provide formal nomenclature for describing poorly understood concepts such as kernels and tasks. FleCSI's data model provides a low-buy-in approach that makes it an attractive option for many application projects, as developers are not locked into particular layouts or data structure representations.

FleCSI currently provides a parallel but not distributed implementation of Binary, Quad and Oct-tree topology. This implementation is based on space filling curves domain decomposition, the Morton order. The current version allows the user to specify the code main loop and the data distribution requested. The data distribution feature is not available for the tree data structure needed in our SPH code and we provide it in the FleCSPH implementation. The next step will be to incorporate it directly from FleCSPH to FleCSI as we reach a decent level of performance. As FleCSI is an on-development code the structure may change in the future and we keep track of these updates in FleCSPH.

Based on FleCSI the intent is to provide a binary, quad and oct-tree data structure and the methods to create, search and share information for it. In FleCSPH this will be dedicated, apply and tested on the SPH method. In this part we first present the domain decomposition, based on space filling curves, and the tree data structure. We describe the HDF5 files structure used for the I/O. Then we describe the distributed algorithm for the data structure over the MPI processes.

2.3 Distributed SPH on multi-core architectures

Before going further in the SPH implementation and our development, we need to give details on the algorithm and the work involved. The general algorithm is presented on algorithm 1

Algorithm 1 SPH implementation

```
1: Read particles from file
2: while not last step do
3:     Generate keys for particles
4:     Load balance particles
5:     Generate local tree data structure
6:     while Physics not done do
7:         Physics: search and distributed search
8:         Update data
9:     end while
10:    Gravitation = FMM
11:    if Output step then
12:        Output data in H5part format
13:    end if
14: end while
```

and shows the features of SPH exposed in the previous section. In this section we present the main features of our code and the problems involved: the domain decomposition, the tree data

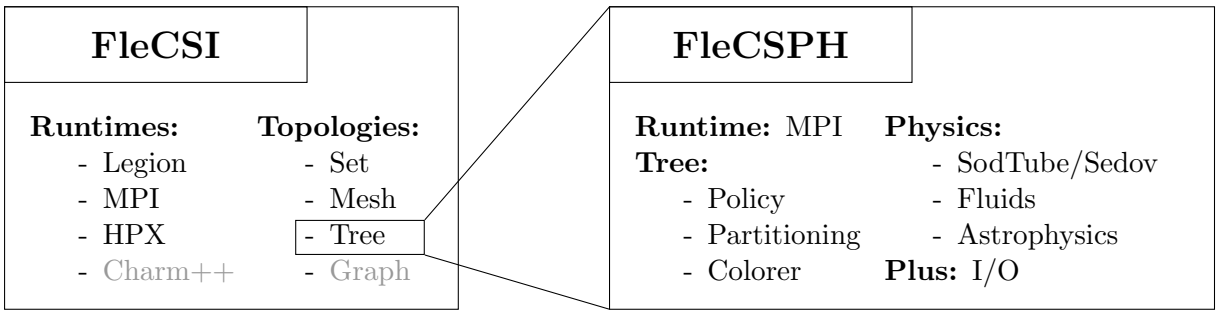

Figure 2.1: FleCSI and FleCSPH frameworks

structure and traversal and our distribution strategy. This project, in collaboration with the LANL, is named FleCSPH.

FleCSPH[2] is a framework initially created as a part of FleCSI. The purpose of FleCSPH is to provide a data distribution and communications patterns to use the tree topology provided in FleCSI applied to SPH. The final purpose of FleCSPH is to complement the tree topology and provide the *colorer* to FleCSI. The distribution strategies associate to a data structure. As presented in the previous sections, SPH is a very good candidate to benchmark the binary, quad and oct tree topology. The demand is high for astrophysics simulation regarding the recent discoveries of the LIGO and the SPH method allows us to generate those simulations. Figure 2.1 presents how FleCSI and FleCSPH are integrated. FleCSPH is based on the tree topology of FleCSI and follows the same structure define in FleCSI. The ideal runtime in FleCSI is Legion but this in-development code does not allow us to do more than static data distribution. This is why we decided to work with the MPI runtime in FleCSPH. Those MPI functions can then be integrated to FleCSI to generate group of particles and labeled them, the coloring.

Figure 2.2 present the file systems of the github repository. We use the tools from Cinch[4] developed at LANL for the CMake and the makefile generation. It also provides the GoogleTests API for our unit tests. The FleCSPH code is currently public and available on github under the *laristra* (Los Alamos Ristra) project. The continuous integration is ensured by using Travis based on Docker and *Dockerfiles* provided in the Docker folder. In addition to Travis for the unit tests we use tools as CodeCov[5] for the code coverage and SonarQube[6] for the quality gates. In the current version we use external libraries: HDF5, H5hut and a specific library for I/O based on H5hut. Those elements are present and installed using scripts in *third-part-library* folder.

For the first implementation we present the code without considering accelerators. We intent to provide an efficient multi-CPU distributed code. The description starts with the domain decomposition strategy which is a basic element for the tree implementation. We explain the tree data structure for the construction and the search of particles along with the distribution strategy.

2.3.1 Domain decomposition

The number of particles can be high and represent a huge amount of data that does not fit in a single node memory. This implies the distribution of the particles over several computational nodes. As the particles moves during the simulation the static distribution is not possible and they have to be redistributed at some point in the execution. Furthermore, this distribution need to keep local particles in the same computation node to optimize the exchanges and computation itself.

[2] http://github.com/laristra/flecsph
[4] http://github.com/laristra/cinch
[5] http://codecov.io
[6] http://sonarqube.org/

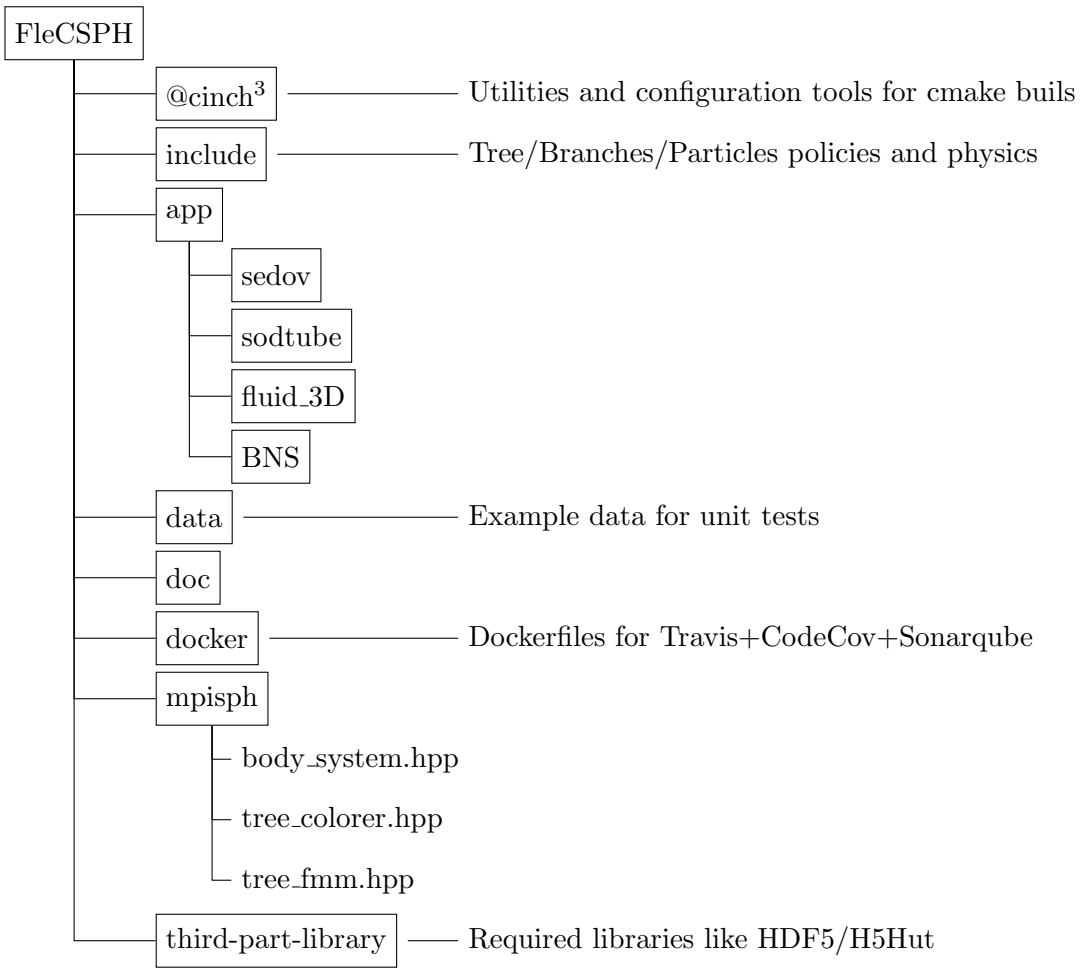

Figure 2.2: FleCSPH structures and files

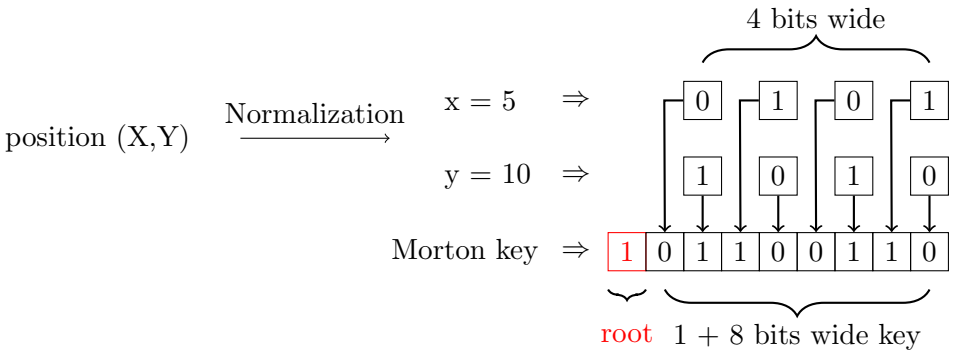

Figure 2.3: Morton order key generation

A common approach is to distribute the particles over computational nodes using *space filling curves* like in [War13, Spr05, BGF+14]. It intends to assign to each particle a key which is based on its spatial coordinates, then sorting particles based on those keys keeps particles grouped locally. Many space filling curves exists: Morton, Hilbert, Peano, Moore, Gosper, etc.

This domain decomposition is used in several layers for our implementation. On one hand, to spread the particles over all the MPI processes and provide a decent load balancing regarding the number of particles. On the other hand, it is also used locally to store efficiently the particles and provide a $O(Nlog(N))$ neighbor search complexity, instead of $O(N^2)$, using a tree representation describe in part 2.3.2.

Several space filling curves can fit our purposes:

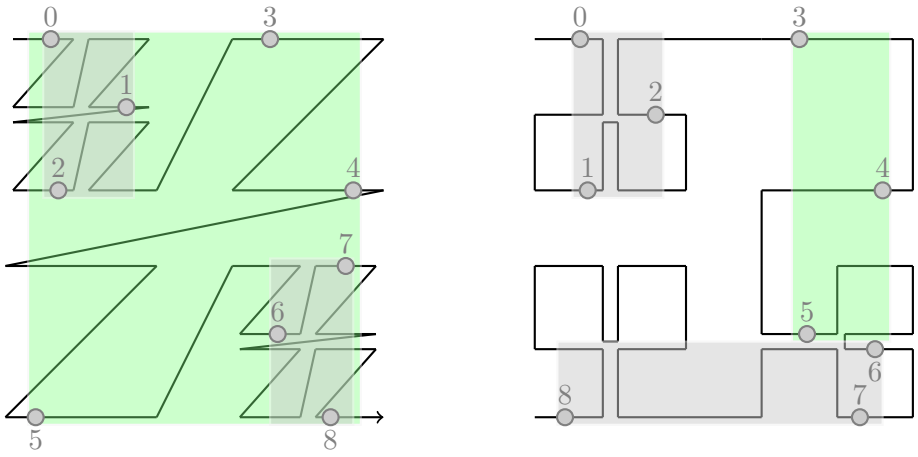

Figure 2.4: Morton and Hilbert space filling curves

The Morton curves

[Mor66], or Z-Order, is the most spread method. This method can produce irregular shape domain decomposition like shown in green on figure 2.4. The main advantage is to be fast to compute, the key is made by interlacing directly the X, Y and Z bits without rotations.

The Hilbert curves

[Sag12] are constructed by interlacing bits but also adding rotation based on the Gray code. This work is based on the Peano curves and also called Hilber-Peano. The construction is more complicated than Morton but allows a better distribution.

On figure 2.4, the Morton (left) and Hilbert (right) space-filling curves are represented in this example. The particles are distributed over 3 processes. The set of particles of the second process appears in green. As we can see there are discontinuities on the Morton case due to the Z-order "jump" over the space. This can lead to non-local particles and over-sharing of particles that will not be needed during the computation. In the Hilbert curve, the locality over the processes is conserved.

In this first implementation of FleCSPH we used the Morton ordering due to the computational cost. The next step of this work is to compare the computation time of different space filing curves.

Technically the keys are generated for each particle at each iteration because their position is expected to change over time. To be more efficient, the keys can stay the same during several steps and the final comparison can be made on the real particles positions. This increase the search time but allows less tree reconstructions.

We use 64 bits to represent the keys to avoid conflicts. The FleCSI code allows us to use a combination of memory words to reach the desired size of keys (possibly more than 64 bits) but this will cost in memory occupancy. The particle keys are generated by normalizing the space and then converting the floating-point representation to a 64 bits integer for each dimension. Then the Morton interlacing is done, and the keys are created. Unfortunately, in some arrangements, like isolated particles, or scenarios with very close particles, the keys can be either badly distributed or duplicate keys can appear. Indeed, if the distance between two particles is less than $2^{-64} \approx$ 1e-20, in a normalized space, the key generated through the algorithm will be the same. This problem is then handle during the particle sort and then the tree generation. In both case two particles can be differentiate based on their unique ID generated at the beginning execution.

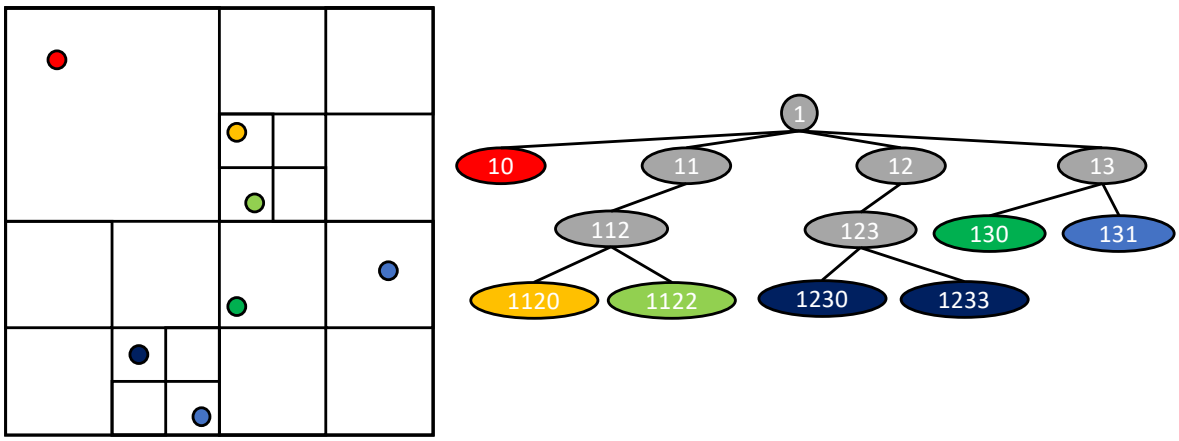

Figure 2.5: Quadtree, space and data representation

2.3.2 Hierarchical trees

The method we use for the tree data structure creation and research comes from Barnes-Hut trees presented in [BH86, Bar90]. By reducing the search complexity from $O(N^2)$ for direct summation to $O(Nlog(N))$ it allows us to do very large simulations with billions of particles. It also allows the use of the tree data structure to compute gravitation using multipole methods.

We consider binary trees, for 1 dimension, quad-trees, for 2 dimensions, and oct-trees, for 3 dimensions. The construction of those trees is directly based on the domain decomposition using keys and space-filling curve presented in section 2.3.1.

As explain in the previous section, we use 64 bits keys. That give us up to 63, 31 and 21 levels in the tree for respectively 1, 2 and 3 dimensions. As presented on figure 2.5 the first bit is use to represent the root of the tree, 1. This allows us to have up to 2^{63} different keys and unique particles.

Tree generation

After each particle get distributed on its final process using its space-filling curve key, we can recursively construct the tree. Each particle is added, and the branches are created recursively if there is an intersection between keys. Starting from the root of key "1" the branches are added at each level until the particles are reached. An example of a final tree is shown on figure 2.5.

Tree search

When all the particles have been added, the data regarding the tree nodes are computed with a bottom up approach. Summing up the mass, position called Center of Mass (COM), and the boundary box of all sub-particles of this tree node.

For the search algorithm the basic idea would be to do a tree traversal for all the particles and once we reach a particle or a node that interact with the particle smoothing length, add it for computation or in a neighbor list. Beside of being easy to implement and to use in parallel this algorithm requires a full tree traversal for every particle and will not take advantage of the particles' locality.

Our search algorithm, presented on Algorithm 2, is a two-step algorithm like in Barnes trees: First create the interaction lists and then using them on the sub-tree particles. In the first step we look down for nodes with a target sub-mass of particles *tmass*. Then for those branches we compute an interaction list and continue the recursive tree search. When a particle is reached, we compute the physics using the interaction list as the neighbors. The interaction list is computing using an opening-angle criterion comparing the boundary box and a user define angle. This way we will not need a full tree traversal for each particle but a full tree traversal for

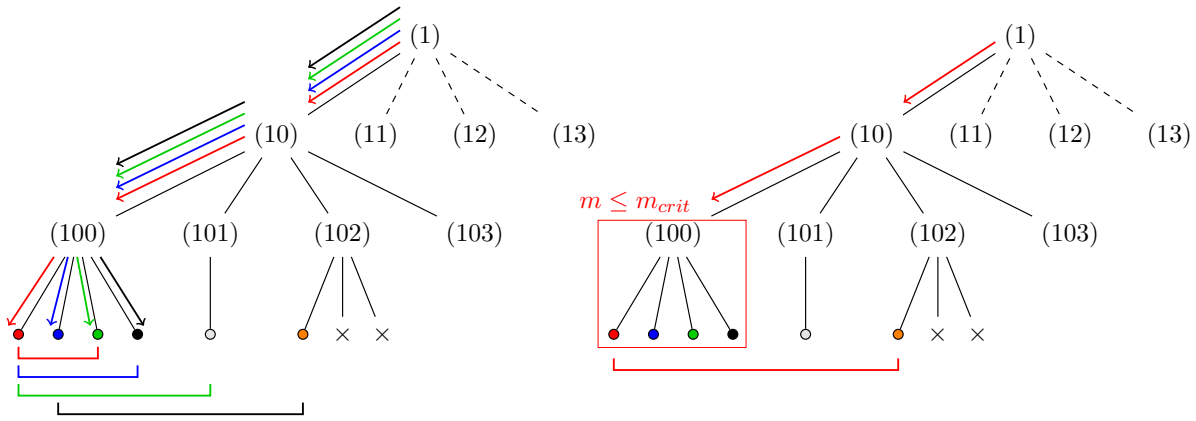

Figure 2.6: Neighbors search using a tree traversal per particle vs a group of particle and computing an interaction list

every group of particles. On figure 2.6 we present the classical and the two steps algorithm. We see that the first method, on left, force to do one walk per particle, compute the interaction list and then apply to particles. On the left, the two-step method, only perform one tree traversal for the whole block of particles, compute the interaction list and then processed to the local computation. Indeed, this implies a check of particle distance during the computation since all the particles from the interaction list are not useful for every particle.

2.3.3 Distribution strategies

The previous section presented the tree data structure that can be use locally on every node. The distribution layer is added on top of it, keeping each sub-tree on the computation nodes. The current version of FleCSPH is still based on synchronous communications using the Message Passing Interface (MPI).

The main distributed algorithm is presented on algorithm 3:

Particle distribution

The sort step, line 5, is based on a distributed quick sort algorithm. The keys are generated using the Morton order described in part 2.3.1. As we associate a unique number to each particle we are able to sort them using the keys and, in case of collision keys, using their unique ID. This gives us a global order for the particles. Each process sends to a master node (or submaster for larger cases) a sample of its keys. We determined this size to be 256 KB of key data per process for our test cases, but it can be refined for larger simulations. Then the master determines the general ordering for all the processes and shares the pivots. Then each process locally sorts its local keys, and, in a global communication step, the particles are distributed to the process on which they belong. This algorithm gives us a good partition in term of number of particles. But some downside can be identified:

- The ordering may not be balanced in term of number of particles per processes. But by optimizing the number of data exchanged to the master can lead to better affectation.
- The load balance also depend on the number of neighbors of each particle. If a particle gets affected a poor area with large space between the particles, this can lead to bad load balancing too.

This is why we also provide another load balancing based on the particles neighbors. Depending on the user problem, the choice can be to distribute the particles on each process regarding

Algorithm 2 Tree search algorithm

```
 1: procedure FIND_NODES
 2:     stack stk ← root
 3:     while not_empty(stk) do
 4:         branch b ← stk.pop()
 5:         if b is leaf then
 6:             for each particles p of b do
 7:                 apply_sub_tree(p,interaction_list(p))
 8:             end for
 9:         else
10:             for each child branch c of b do
11:                 stk.push(c)
12:             end for
13:         end if
14:     end while
15: end procedure
16:
17: procedure APPLY_SUB_TREE(node n, node-list nl)
18:     stack stk ← n
19:     while not_empty(stk) do
20:         branch b ← stk.pop()
21:         if b is leaf then
22:             for each particles p of b do
23:                 apply_physics(p,nl)
24:             end for
25:         else
26:             for each child branch c of b do
27:                 stk.push(c)
28:             end for
29:         end if
30:     end while
31: end procedure
32:
33: function INTERACTION_LIST(node n)
34:     stack stk ← root
35:     node-list nl ← ∅
36:     while not_empty(stk) do
37:         branch b ← stk.pop()
38:         if b is leaf then
39:             for each particles p of b do
40:                 if within() then
41:                     nl ← nl + p
42:                 end if
43:             end for
44:         else
45:             for each child branch c of b do
46:                 if mac(c,angle) then
47:                     nl ← nl + c
48:                 else
49:                     stk.push(c)
50:                 end if
51:             end for
52:         end if
53:     end while
54: end function
```

Algorithm 3 Main algorithm

```
1:  procedure SPECIALIZATION_DRIVER(input data file f)
2:      Read f in parallel
3:      Set physics constant from f
4:      while iterations do
5:          Distribute the particles using distributed quick sort
6:          Compute total range
7:          Generate the local tree
8:          Share branches
9:          Compute the ghosts particles
10:         Update ghosts data
11:         PHYSICS
12:         Update ghosts data
13:         PHYSICS
14:         Distributed output to file
15:     end while
16: end procedure
```

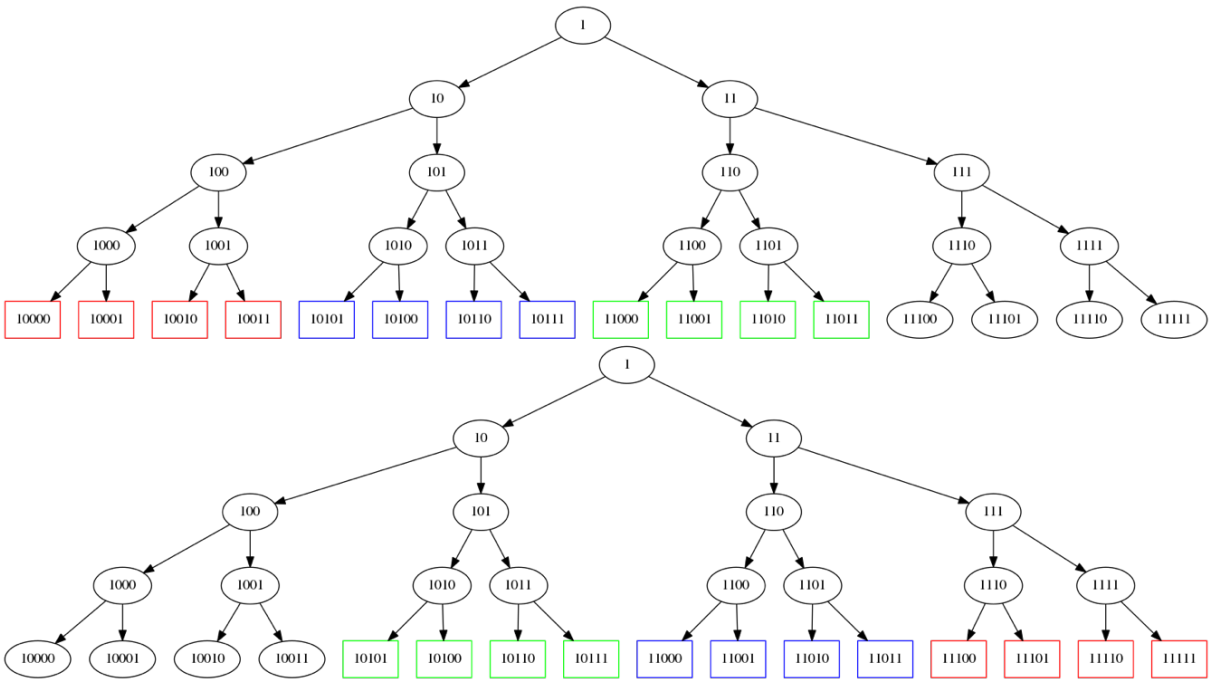

Figure 2.7: Binary tree for a 2 processes system. Exclusive, Shared and Ghosts particles resp. red, blue, green.

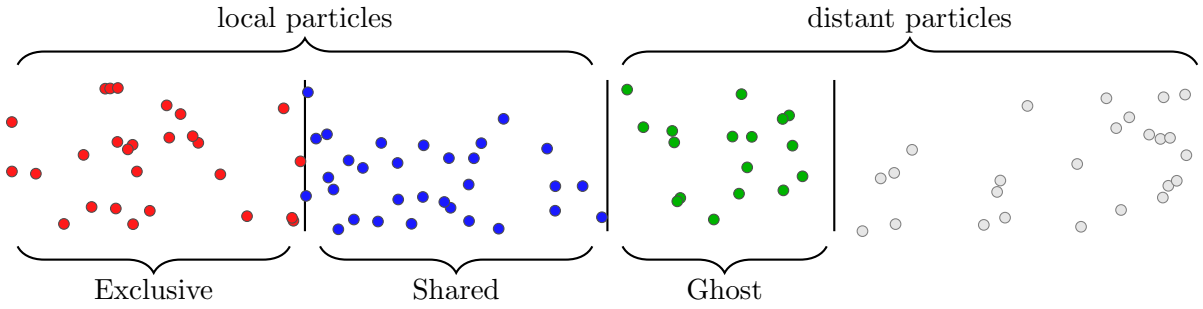

Figure 2.8: Particles "coloring" with local particles: exclusive, shared and distant particles useful during run: ghost particles

the number of neighbors, having the same amount of physical computation to perform on each process.

After this first step, the branches are shared between the different processes, line 8. Every of them send to its neighbors several boundaries boxes, defined by the user. Then particles from the neighbors are computed, exchanged and added in the local tree. Those particles are labeled as NON_LOCAL particles. At this point a particle can be referenced as: EXCLUSIVE: will never be exchanged and will only be used on this process; SHARED: may be needed by another process during the computation; GHOSTS: particles information that the process need to retrieve from another process. An example is given for 2 processes on figure 2.8 and on a tree data structure on figure 2.7.

Exchange Shared and Ghosts particles

The previous distribution shares the particles and the general information about neighbors' particles. Then each process is able to do synchronously or asynchronously communications to gather distant particles. In the current version of FleCSPH an extra step is required to synchronously share data of the particles needed during the next tree traversal and physics part. Then after this step, the ghosts' data can be exchanged as wanted several times during the same time step.

2.3.4 Fast Multipole Methods

We described in the previous chapter a method to compute gravitational interactions faster than the $O(N^2)$ n-body algorithm, the Fast Multipole Method, FMM. This allows an approach with a precision depending of a parameter called the Multipole Acceptance Criterion, MAC. This is needed for us to target binary neutron stars simulations with high number of particles.

This approach is also based on the tree topology used for the SPH method. Three main functions are used:

- `mpi_exchange_cells`

  share the centers of mass computed in the tree up to a determined mass. By default, the program shares the lowest COM, the leaves.

- `mpi_compute_fmm`

  After gathering the COM this step performs the M2M computation and also isolate the particles needed for the distant P2P step.

- `mpi_gather_cells`

  Gather The contribution of all the other processes and sum in the local branch. Then this step performs both the M2P and P2P computations. A specific P2P for distant particles is added to take in account the particles found on other processes in the M2M step.

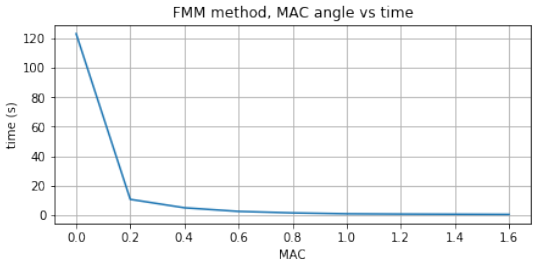

Figure 2.9: Evolution of time regarding the multipole acceptance criterion for FMM method.

The speed of computation varies with the choose of the MAC angle. An example is given on figure 2.9. It presents the time varying with the angle. We note that bigger the angle, faster the computation time. The limit of the angle is $\Theta = \frac{\pi}{2}$. The loss of precision can be quantified with the evolution of linear and angular momentum. We fixed our MAC to $\Theta = 1$.

2.3.5 Input/Output

Regarding the high number of particles, an efficient, parallel and distributed I/O implementation is required. Several choices were available, but we wanted a solution that can be specific for our usage. The first requirement is to allow the user to work directly with the Paraview visualization tool and splash[7] [Pri07].

We base this first implementation on HDF5 [FCY99] file structure with H5Part and H5Hut [HAB+10]. HDF5 support MPI runtime with distributed read and write in a single or multiple file. We added the library H5hut to add normalization in the code to represent global data, steps, steps data and the particles data for each step. The I/O code was developed internally at LANL and provides a simple way to write and read the data in H5Part format. The usage of H5Hut to generate H5part data files allows us to directly read the output in Paraview without using a XDMF descriptor like requested in HDF5 format.

2.4 Distributed SPH on hybrid architectures

We constructed an efficient and reliable SPH code working on classical architecture clusters. In this section we present our multi-GPU implementation and compare it with our multi-CPU code. We kept the same data structure, distribution strategy and code architecture in the accelerator code. Several options are possible for the accelerator implementation, but we wanted to keep the code usable and working for the domain scientists. As the current version of FleCSI does not allow utilization of accelerator like GPU for the data structure, we decided to embed the GPU code directly on FleCSPH. We provided the same approach we studied in the Langford problem offloading part of the tree computation on the accelerators.

2.4.1 Distribution strategies

The FleCSPH framework provides all the tools and distribution strategies for multi-CPU and distributed computation. In order to target hybrid architectures several approaches were possible. They were identified in the previous metrics for the Langford problem. The first one is to implement the whole tree traversal and data representation on GPU. This strategy imposes several downsides especially for asynchronous communication. The data structure of FleCSI and FleCSPH does not allows the full transformation of the data structure into CUDA code and, furthermore, this would transform the framework into a problem dependent API. Even if the performances would be slightly better, the aim of this framework is to target multi-physics problems and thus general.

[7] `http://users.monash.edu.au/~dprice/splash/`

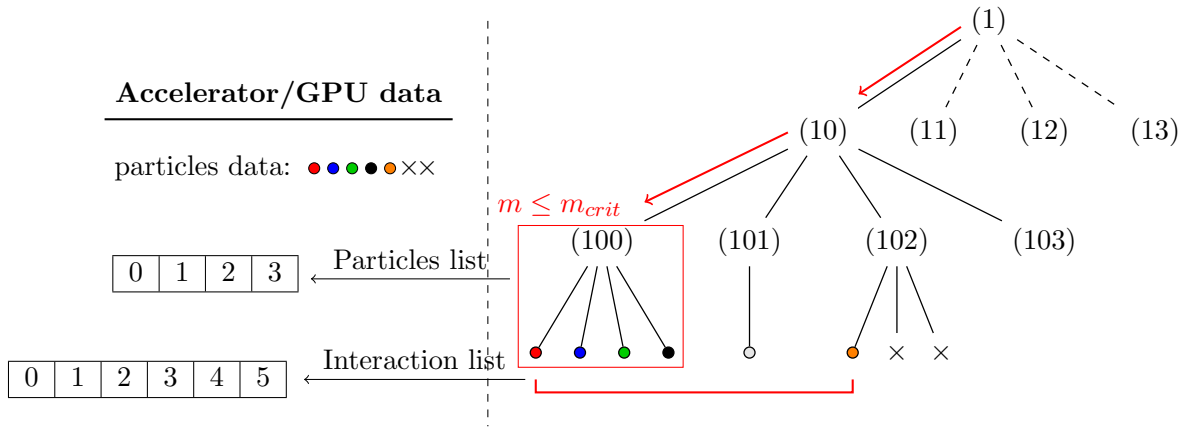

Figure 2.10: Task resolution using GPUs

The second strategy is the one used in the Langford problem case. The smoothing length computation is a tree traversal that's lead to a group of particles and their neighbors. We decided to offload the physics computation on accelerators.

Figure 2.10 presents the distribution of tasks with the accelerator. The tree traversal itself stays on the host processor and lower part of the tree are offloaded to the accelerators. The traversal is done in parallel on the host and, when a group of particles and its neighbors list is reached, the data are transferred to the GPU for computation. With this method the GPU is fully used for regularized computation and the CPU handle the data structure. When the tree traversal is done, the CPU wait for last GPU tasks to complete and can gather the result or start another traversal leaving data on GPUs.

2.4.2 Physics on accelerators

The computation of physics is also slightly different on accelerators. Indeed, the CPU send to the GPU indexes with the particles and their possible neighbors. The GPU perform a brute force computation with $O(n^2)$ algorithm. It keeps checking if the particles received are inside the smoothing length radius. The target particle is loaded in local memory and its neighbors are stored in the local memory for a WARP based computation. The threads then iterate on the local particles and output together in global memory.

2.5 Results

In this part we compare the results of the multi-GPU version and the multi-CPU version of FleCSPH. We show the benefit of using hybrid architectures even on irregular problem with high communications and computations levels.

2.5.1 Simulations

The results and tests were done on several physical and astrophysical simulations in order to check the code behavior and reliability.

The first tests, done on Sod shock tube and Sedov blast wave were presented in the previous chapter, showed perfect results.

The fluid simulation is presented on figure 2.11. On this figure we can see a 40,000 particles simulation executed on multiple nodes of the ROMEO supercomputer. This dam break simulation gave us the opportunity to represent the behavior of boundary conditions with a high number of particles.

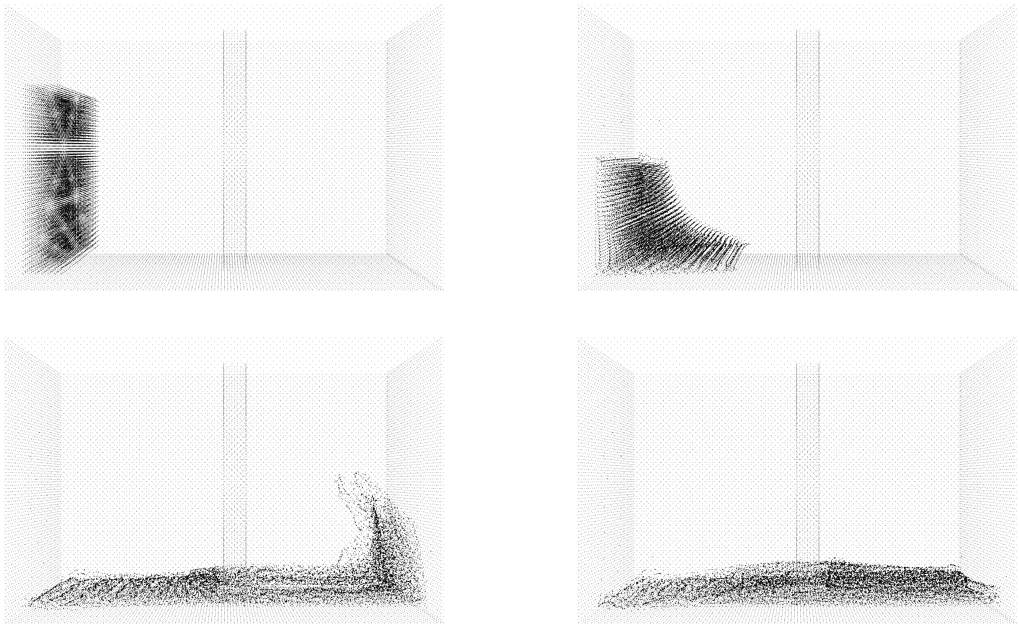

Figure 2.11: Fluid flow simulation, the dam break. For $t = 0$, $t = 0.4$, $t = 0.8$ and $t = 1$ seconds

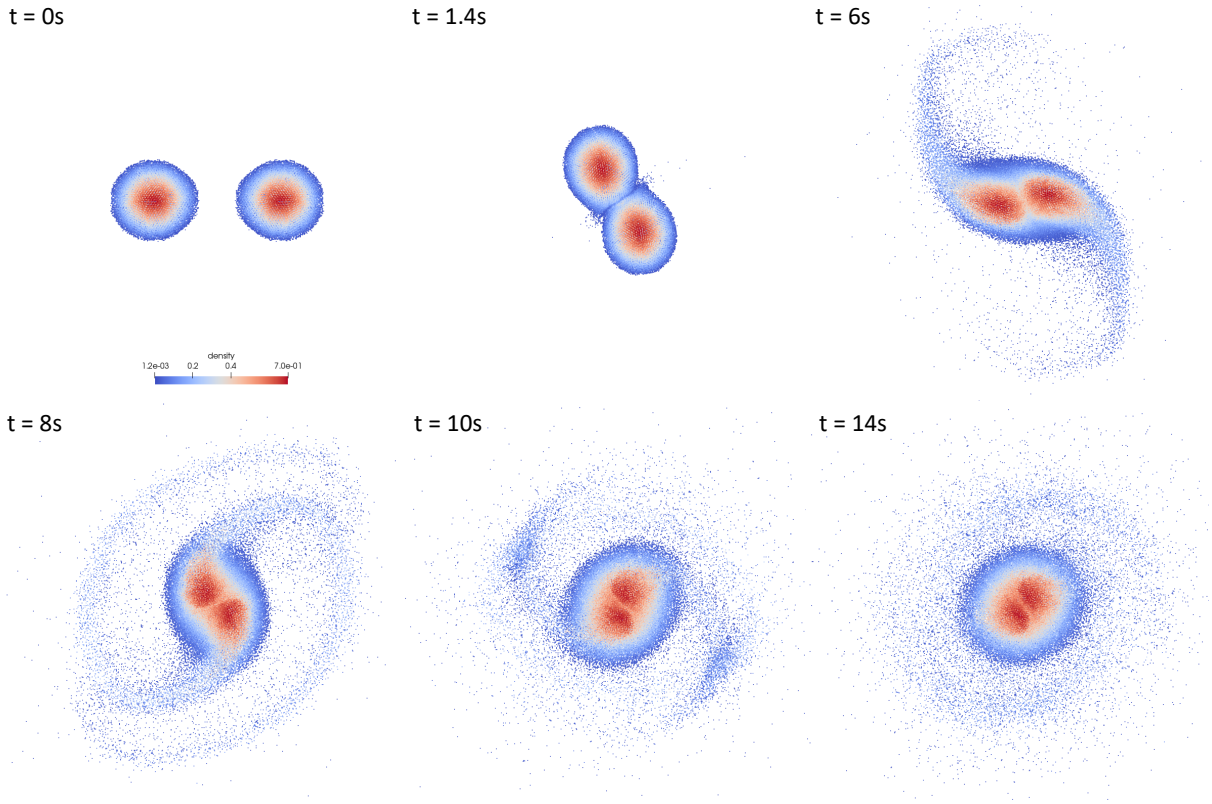

Figure 2.12: Binary Neutron Stars coalescence with 40.000 particles.

In order to target a problem with both SPH and gravitation we decided to work on Astrophysics events like Binary Neutron Stars. The initial data were generated using python 3.5 to compute the position, mass and smoothing length of every particle. A first step is done for the relaxation, so that the particles take their location. The system relaxed then evolve following the merging equations presented in the previous chapter.

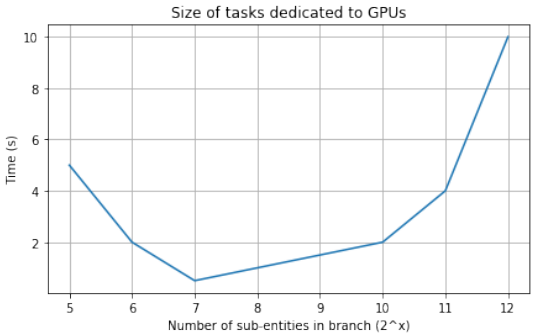

Figure 2.13: CPU-GPU tasks work balancing

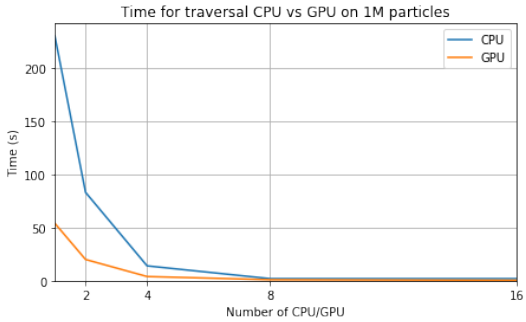

Figure 2.14: CPU vs GPU time per iteration

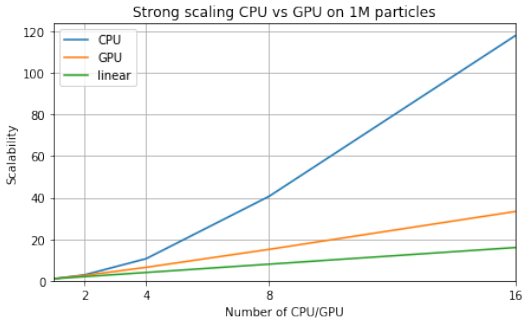

Figure 2.15: CPU vs GPU scalability per iteration

Figure 2.12 presents the binary neutron star coalescence for 40,000 particles. The computation took 5 hours on four nodes of the supercomputer ROMEO. This simulation is done with a total of 750 outputted steps with more than 100,000 iterations.

2.5.2 Performances

The two metrics we worked on, in part II, gave us a hint in order to reach performances in this simulation code. The first step was to find the best repartition between the host and device.

Figure 2.13 shows the work balancing. With the smallest distribution the CPU and GPU keep exchanging data for very small amount of computation. At the opposite, if the CPU value is too high the $O(n^2)$ part is too important. We choose the value to be configured by the user and defaulted at 7.

On figure 2.14 and figure 2.15 we find the strong scaling and scalability tests for CPU and GPU versions using the empiric best depth of repartition. These tests have been led on 500,000 particles binary neutron star merging during the coalescence step, when the particles are very close. The time comparison with strong scaling shows us that the GPU version goes faster than the CPU one with a peak of 5 times faster. On the other hand, the scalability graph presents that the CPU version is more scalable than the GPU one. Indeed, the number of particles to handle by the GPUs gets too small regarding their computational power and the transfers between CPU and GPU take a more important part of time.

2.6 Conclusion

In this section we presented a tool completing our metric. This production application dedicated to smoothed particles hydrodynamics and gravitation simulation is called FleCSPH. It allows us to target both computation and communication walls in a highly irregular context.

We based our hybrid implementation on the knowledge from the two first metrics of our benchmark. We showed the advantage of using hybrid architectures on this kind of application and this behavior can be found in many others.

It is important to notice that the GPU usage is not the best in this case. Indeed, a dedicated application for each specific SPH problem would be more efficient. The aim is to provide a framework that can be used on a large set of clusters and architectures.

In this version, the GPUs enabled us to provide an acceleration of up to 5 times faster compared to the full-CPU computation. It proves the benefit of hybrid architectures even when confronted to computation and communication wall over high irregular applications. The work is shared between host and device by sharing part of the tree of particles to the GPU.

Bibliography

[Bar90] Joshua E Barnes. A modified tree code: don't laugh; it runs. *Journal of Computational Physics*, 87(1):161–170, 1990.

[BGF+14] Jeroen Bédorf, Evghenii Gaburov, Michiko S Fujii, Keigo Nitadori, Tomoaki Ishiyama, and Simon Portegies Zwart. 24.77 pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 54–65. IEEE Press, 2014.

[BH86] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[BMC16] Ben Bergen, Nicholas Moss, and Marc Robert Joseph Charest. Flexible computational science infrastructure. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2016.

[BTSA12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[FCY99] Mike Folk, Albert Cheng, and Kim Yates. Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of Supercomputing*, volume 99, pages 5–33, 1999.

[HAB+10] Mark Howison, Andreas Adelmann, E Wes Bethel, Achim Gsell, Benedikt Oswald, et al. H5hut: A high-performance i/o library for particle-based simulations. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[KK93] Laxmikant V Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[Mor66] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[Pri07] Daniel J Price. Splash: An interactive visualisation tool for smoothed particle hydrodynamics simulations. *Publications of the Astronomical Society of Australia*, 24(3):159–173, 2007.

[Sag12] Hans Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[Spr05] Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[War13] Michael S Warren. 2hot: an improved parallel hashed oct-tree n-body algorithm for cosmological simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 72. ACM, 2013.