Chapter 2

FleCSPH

2.1 Introduction

The previous part describes the method we want to implement and presents the wall in the smoothed particle hydrodynamics method. We showed that this application meet the communication and computation wall in an irregular behavior context. FleCSPH is a complex application that, beside of being interesting for our purpose, need to be accurate on the physics aspect to be use by the domain scientists from LANL. I also note that this is my first work on developing a production code from scratch.

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. For FleCSPH we give details on the domain decomposition strategy used with Morton Ordering. The tree traversal algorithm choices are then explained. We show the first results on the multi-CPU version of FleCSPH. In the last part we present our multi-GPU implementation of FleCSPH and the advantage using accelerators.

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 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

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

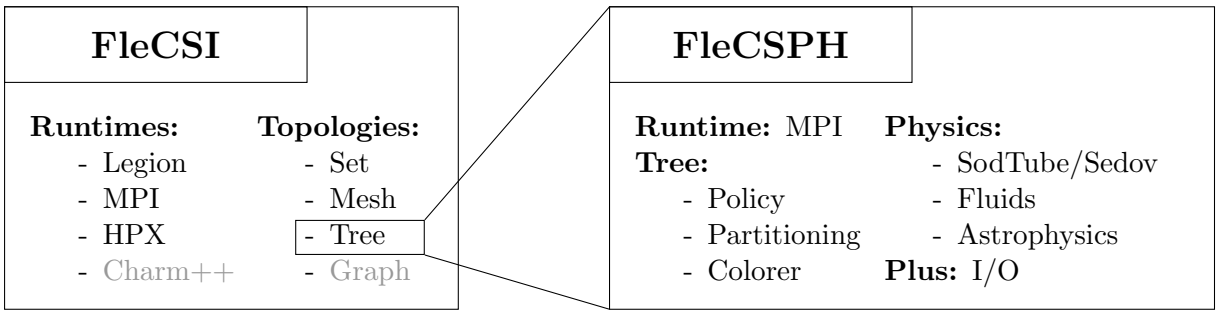

Figure 2.1: FleCSI and FleCSPH frameworks

Oct-tree topology. This implementation is base 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 FleCSPH implementation

[2] is a framework initially created as a part of FleCSI. FleCSPH intents to propose an SPH implementation and extends the topology provides in FleCSI, the tree topology. We propose the MPI implementation to generate group of particles and the data distribution on the processes.

Figure 2.2 present the file systems of the github repository. We use the tools from Cinch[4] for the CMake and the makefile generation. It also provide the GoogleTests API for our unit tests.

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.

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 spacial 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

[2] `http://github.com/laristra/flecsi`
[4] http://github.com/laristra/cinch

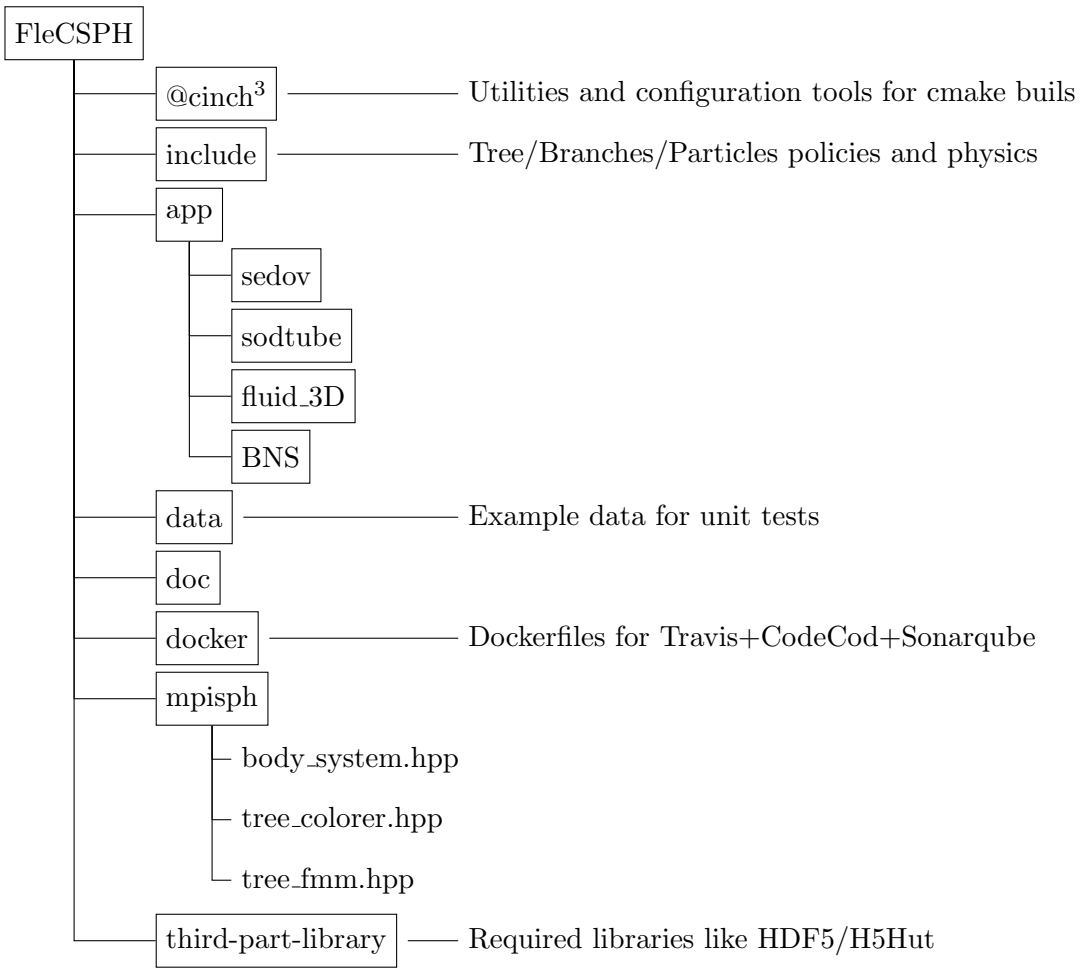

Figure 2.2: FleCSPH structures and files

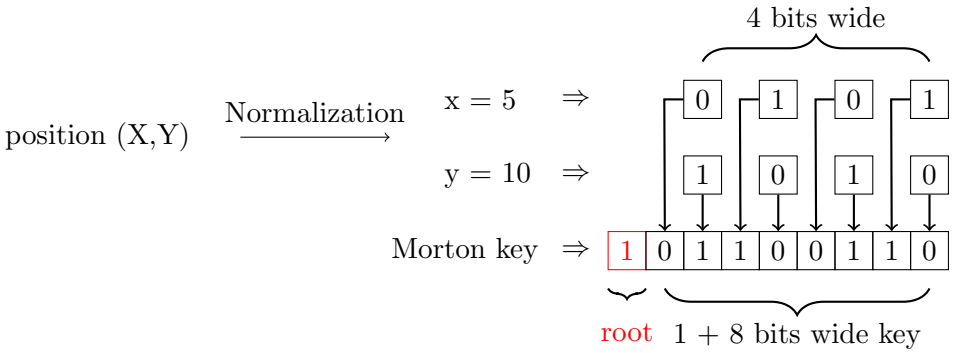

Figure 2.3: Morton order key generation

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:

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 Fig. 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.

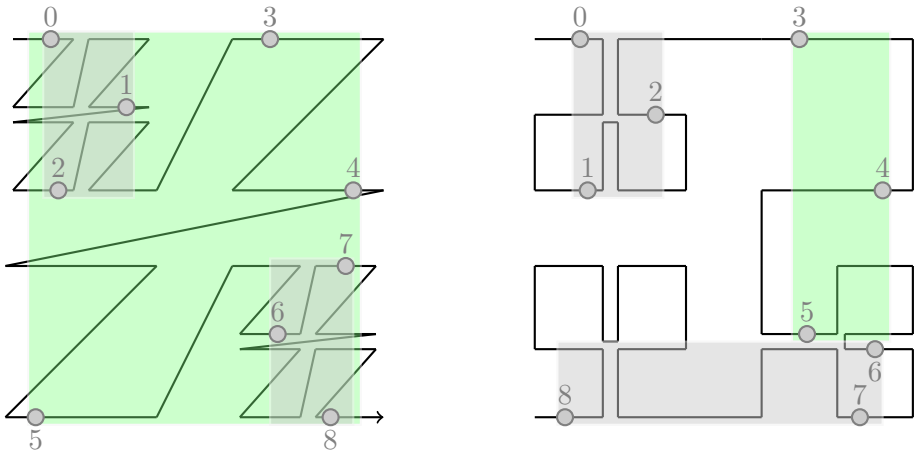

Figure 2.4: Morton and Hilbert space filling curves

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 Fig. 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 dimensions. 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.

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

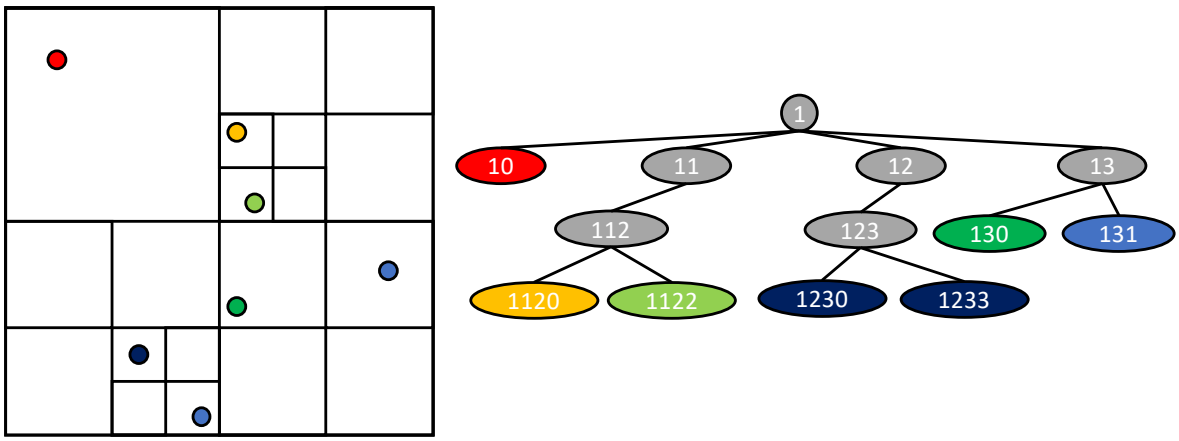

Figure 2.5: Quadtree, space and data representation

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 Fig. 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 levels until the particles are reached. An example of a final tree is shown on Fig. 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 1, 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 an user define angle. In this way we will not need a full tree traversal for each particle but a full tree traversal for every group of particles.

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.

Algorithm 1 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
```

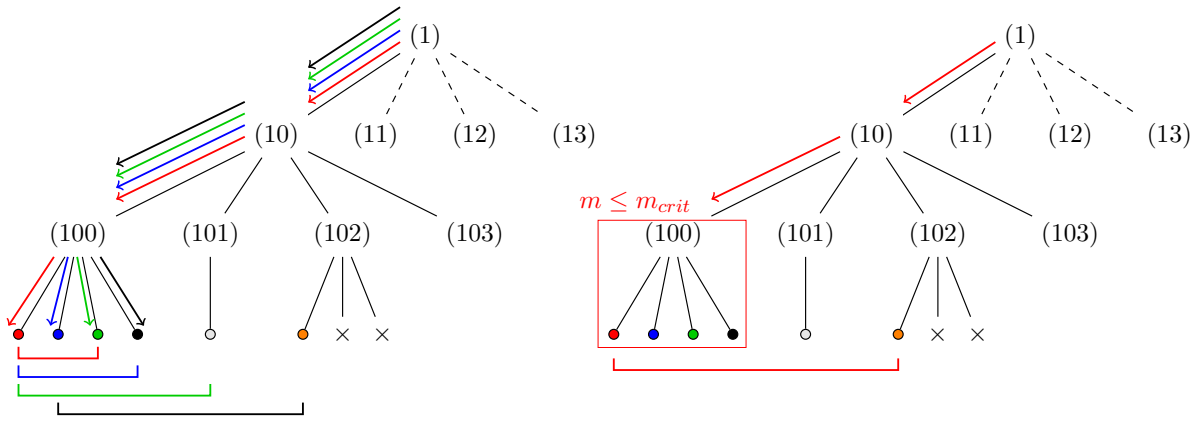

Figure 2.6: Neighbors search using a tree traversal per particle vs a group of particle and computing an interaction list

The current version of FleCSPH is still based on synchronous communications using the Message Passing Interface (MPI).

Algorithm 2 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
```

The main distributed algorithm is presented on algorithm 2:

Particle distribution

The sort step, line 5, is base 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 sort 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

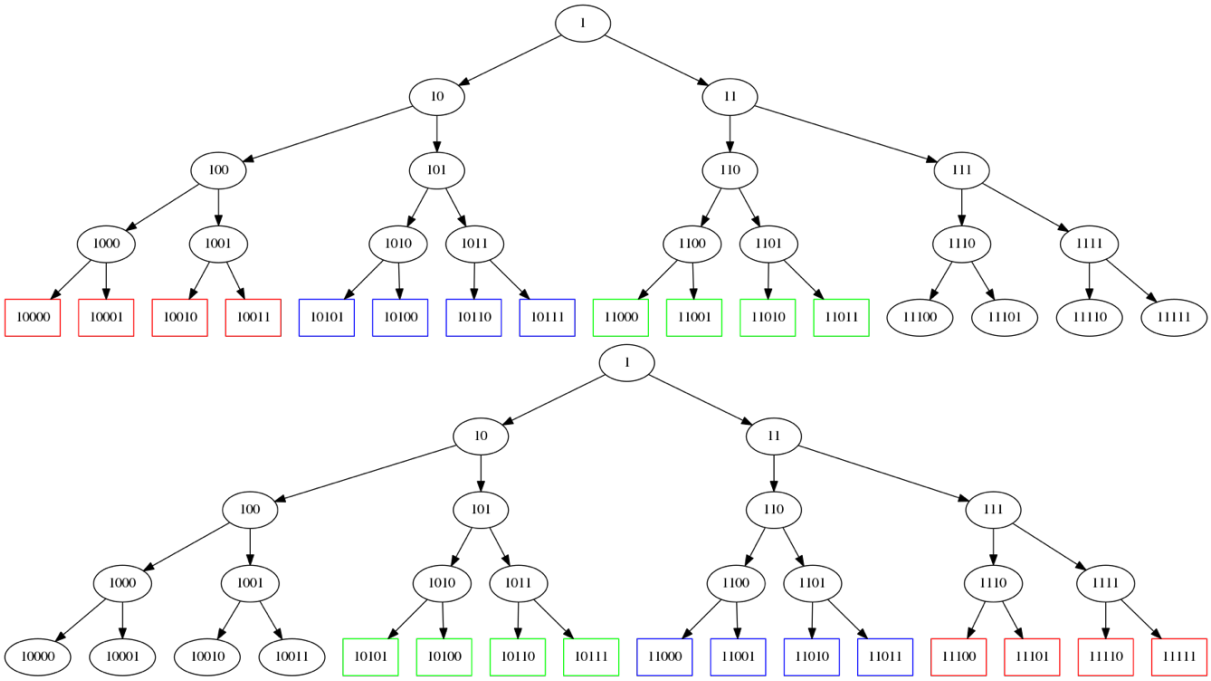

Figure 2.7: Binaries tree for a 2 processes system. Exclusive, Shared and Ghosts particles resp. red, blue, green.

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 get 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 processes regarding the number of neighbors, having the same amount of physical computation to perform on each processes.

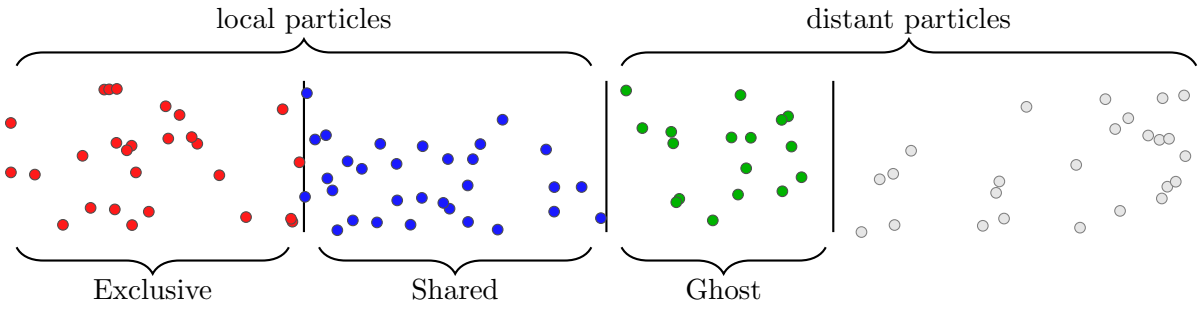

Figure 2.8: Particles "coloring" with local particles: exclusive, shared and distant particles useful during run: ghost particles

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 I/O

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[5] [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 files. We added the library H5hut to add normalization in the code to represent global data, steps, steps data and the particles data for each steps. 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 FleCSPH GPU implementation

For the purpose of this thesis we needed to compare the approaches of CPU and GPU on the FleCSPH application. This application have to be considered.

2.5 Conclusion

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

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.