Master in High Performance Computing

Forwarding GADGET to exa-scale

Supervisors:
Luca Tornatore,
Alberto Sartori

Candidate:
Eduardo Quintana Miranda

5th edition
2018–2019

This document is the thesis for the degree of Master in High Performance Computing offered by the Scuola Internazionale Superiore di Studi Avanzati (SISSA) and the International Center for Theoretical Physics (ICTP) in the year 2019. The manuscript was written in TEX, the typesetting system created by Donald Knuth.

A la memoria de mis queridos abuelos

Table of Contents

Almost blank page.

Chapter I

INTRODUCTION

Since the advent of digital computing machines in the last century, natural scientists have used their aid to explore physical theories and analize experimental data at a scale of computational burden that *pen and paper* alone would have never been able to tackle. Computational Cosmology in particular, a line of research derived from Astrophysical Cosmology, deals with the problems the latter proposes but whose solution involves the use of computer simulations of physical systems of interest to Cosmology; eg. the time evolution of a portion of the universe. Just like a laboratory experiment, a computational physicist can use the simulations to estimate physical properties of the cosmos using as instruments computing stations. In general these computational experiments can help support cosmological theories when the result of the simulations are in agreement with observation, or debunk them otherwise if the evidence against is strong enough.

This thesis will focus on one particular code for cosmological simulations named GADGET, an acronym that stands for GAlaxies with Dark matter and Gas intEracT, whose lastest public version is GADGET-2. Its relevance to the scientific community can be judged by the 3'690 citations (according to the *Web of Science*) to the paper [Springel V. 2005] that presented it. The GADGET code has been publicy available since the beginning, and that has lead to a large number of custom versions by several different groups that developed special features of their own interest. The mainline and the custom codes have been used to perform many of the most advanced state-of-the-art cosmological simulations since almost 20 years, bringing results that were, and still are, at the cuting edge of the research in cosmology.

In spite of several efforts in recent years, GADGET is still conceived and written as a *monolithic* code, meaning that:

- the details of the tree data structure are profoundly and explicitly intertwined in all the parts that rely on the tree itself for their operations;
- in order to develop any new physical module that needs to interact with fundamental data structures and routines, every developer must add, or modify parts of, code situated in core parts like domain decomposition, tree routines etc—he or she also has to add new variables inside fundamental data structures;
- a large number of intertwined `#ifdef` regions are used to switch on and off different features or competing implementations of some algorithm or code section;
- there is significant code replication, specially in tree-based algorithms and *all-to-all* communication schemes.

Moreover, the overall code architecture does not allow it to efficiently scale to many thousands of processors when tackling exceptionally challenging problems at the current cutting edge of research. The most representative of those cases are the zoomed-in simulations of very massive single objects. The above points imply that both to maintain and to develop the code are increasingly difficult, as well as implementing different algorithms or experimenting with different implementation for the same algorithms in a clean way. The data layout for particle-related structures may quickly become sub-optimal in many respects due to the almost casual insertion of new variables and the considerable size of the data structures that host each particle's data. In front of these issues, and of the need of updating the code's design for the forthcoming exa-scale architectures, a thorough re-design of crucial parts of the code has been undergone by a core team of long-term developers of the code's mainline.

As a general strategy, the re-design aims to enhance the modularity of the code, so to have a clear separation of different modules that should communicate through well-defined APIs. In turn, this would allow to easily develop, test and adopt different algorithms, or implementations of an algorithm, to solve a given problem. This thesis is part of that effort, focusing from some of its fundamental pillars: the domain decomposition and the tree building (see Chapter 2 for details). It analyzes their current behaviour and weaknesses, and proposes some well-motivated improvements.

The rest of this document is structured as follows: Chapter 2 explores the implementation details regarding the core of the current version under developement of PGADGET-3, in Chapter 3 some improvements of the code are discussed and proposed, the results of the benchmarks for testing all the changes proposed are shown in Chapter 4, finally conclusions and suggestion for further work are discussed in Chapter 5. For the proofs of the propositions presented throughout the book see Appendix D.

Chapter II

GADGET

§1. THE CORE

THE current stable version of PGADGET-3 is a huge collaboration project made from different modules comprising internal utilities and physics that make up to 250 source files written in `C` or `Fortran`. Together they sum up to 250 thousand lines of code. Due to the sheer size of it, this thesis will only concentrate on a smaller subset of those modules that when put together can be regarded as the *core* of the software. The most relevant modules in the core are:

- `allvars.c`, it hosts global variables that constitute the state of the program, they can be either relative to the local MPI process or the entire MPI *world*;
- `begrun.c`, it handles the initialization of the global variables, it calls the module that reads an initial state from disk and calls the first domain decomposition;
- `domain.c`, it handles the domain decomposition;
- `endrun.c`, it smoothly finishes the execution by freeing the memory from the heap and writting the final state to disk;
- `forcetree.c`, it contains routines relative to the gravitational oct-tree of the simulation;
- `look_around.c`, it contains the routines that searches in the local oct-tree for neighbors (close by particles);
- `main.c`, the main function;
- `mymalloc.c`, a memory manager module;
- `peano.c`, routines relative to particles ordering or indexes according to the Hilbert and Z space filling curves;
- `read_ic.c`, it reads the parameters and initial condition for the simulation.

The workflow of the code is simplified in the following listing:

```
main()
    call begrun():
        initialize variables, eg. call mymalloc_init();
        call read_ic(), read the initial conditions from a file;
        call domain_Decompose(), the first domain decomposition;
    repeat:
        perform a time-step evolution;
        compute physical quantities;
        call domain_Decompose();
    call endrun(), to release memory and save the final state to disk;
```

Listing 1. Simplified workflow in GADGET.

§2. DOMAIN DECOMPOSITION

THE MILLENIUM SIMULATION, a high-resolution run of GADGET, used $N = 2160^3 \approx 10^{10}$ particles to study the evolution of a cubic region of the universe of size $500h^{-1}$ Mpc from 380'000 years after the Big Bang until the present time [Springel. Millenium]. If a single process was to run a simulation of that size on a shared memory paradigm, the host machine would need around 0.5TB of memory just to store the velocities, position and mass of each of those particles. Therefore, in practice such feat and larger can only be achieved with distributed memory parallelization. That is, the particles involved in the simulation must be distributed among MPI processes running on different machines. The algorithm that performs that operation is called *domain decomposition.*

Each MPI process is then responsible for the time evolution of the subset of particles that it *owns* in its local memory. Furthermore, the dynamics on this type of cosmological simulations involve the interaction of pairs of particles and in order to minimize the amount of computational *bottlenecks* caused by MPI processes communication outside of physical shared memory regions, processes are better off *owning* particles which are nearby neighbors in 3-dimensional space. GADGET achieves that by using a Hilbert space-filling-curve [Springel V. 2005].

The Hilbert curve is controlled by a parameter K which is called *bits per dimension*. That means that the 3-dimensional simulation box will be divided along each cartesian axis into 2^K segments of equal length, leading to a total subdivision of space into $N_K = 2^{3K}$ cubic cells. The Hilbert curve traverses the discretized space of cells visiting each one of them exactly once in such a way that neighboring cells in this order are also neighbors in the 3-dimensional space by sharing a vertex, an edge or a face. Consequently, there will be a *one-to-one* correspondence between cells and their traversal index along the curve, here referred to as *Hilbert-key*. All mathematical points within a single cell are mapped to the Hilbert-key associated to that cell. Considering the possibility that the distribution of particles in space could be very inhomogeneous, the value of K should be big enough to resolve those regions of space on which particles are very close to each other. By default K is choosen to be highest value possible: $K = 21$, leading to $N_K = 2^{63}$ so that the Hilbert-keys fit into unsigned 64 bits integers. For details on the Hilbert curves see [Haverkort, 2011].

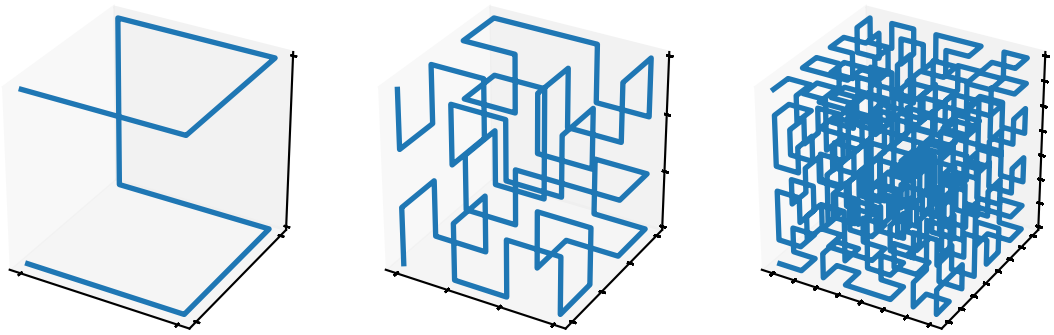

Figure 1. Hilbert curves for $K = 1, 2, 3$ from left to right respectively.

Put together, consecutive values of the Hilbert-keys, represent connected regions of space whose surface to volume ratio is relatively small. Then it makes perfect sense to let each MPI process *own* all particles within one or more of those connected regions. In general a **domain decomposition** is a match between the N_K cubic cells of space, identified by their Hilbert-keys, and the MPI processes, such that every key is assigned to exactly one process.

To explain the methods used in GADGET for the domain decomposition and the modifications proposed to those methods, it is better to formalize some terminology into the language of set theory. Let's start by denoting P as the set of all particles in the simulation and $N_P = |P|$ be the total number of particles. Then the Hilbert-key is a map $\mathsf{key}\colon P \to \mathbb{N}$. Let us also denote $\mathcal{X}$ as the set of all MPI processes involved in the simulation and $p = |\mathcal{X}|$ the size of that set.

DEFINITION 2.1. A **domain** $[a, b)$ is the set of all Hilbert-keys k that satisfy: $a \leq k < b$. Since the number of bits per dimension is fixed, a valid domain must be a subset of $[0, N_K)$.

DEFINITION 2.2. Let $d = [a, b)$ be a domain, P_d denotes the set of all the particles in that domain:

$$P_d = \{i \in P \mid \mathsf{key}(i) \in d\}\,.$$

In order to construct domains, GADGET uses a data structure called *top-tree* to encode a topology of the particle distribution which is coarser and more manageable than a 3-dimensional grid division of space with a single Hilbert-key per cell:

DEFINITION 2.3. A **top-tree** $\mathcal{T}$ is an oct-tree for which each vertex represents a domain, such that the root is the key interval $[0, N_K)$ (all the Hilbert-keys), any vertex is either a leaf or it has exactly 8 children and for any non-leaf vertex $[a, b)$ the i^{th} child represents the domain $\big[a + i l_{ab}/8, a + (i+1) l_{ab}/8\big)$ where $l_{ab} = b - a$; ie. the interval $[a, b)$ is partitioned into 8 equal size segments and each of the 8 children is assigned to one.

PROPOSITION 2.1. By construction, each Hilbert-key is mapped to exactly one leaf of the top-tree and when visited in *depth first search* (DFS) order, consecutive leaves will correspond to consecutive domains—ie. if $[a_i, b_i)$ and $[a_{i+1}, b_{i+1})$ are the domain represented by leaves i and $i+1$ respectively, then $b_i = a_{i+1}$.

See Appendix D for the proof the previous claim. As a collorary of Proposition 2.1 it follows that the union of an arbitrary number of consecutive leaves is a domain.

EXAMPLE 2.1. The following: $[7, 8)$, $[8, 16)$ and $[16, 17)$ could be three consecutive leaves of a particular top-tree, and their union: $[7, 8) \cup [8, 16) \cup [16, 17) = [7, 17)$ is again a valid domain. □

The leaves of the top-tree—not the Hilbert-keys—become the building blocks for the domain decomposition. Furthermore, it is possible to construct a tree with a reasonable number of leaves and still be able to have a fine grained topology of highly populated regions of space while keeping a coarse grained topology on the lesser populated regions.

DEFINITION 2.4. Let $L = \{L_1, L_2, \ldots\}$ be the sequence of leaves of the top-tree in DFS order, a **domain split** $\mathcal{S}$ of L is a partition of L into contiguous sub-sequences.

EXAMPLE 2.2. Let $L = \{L_1, L_2, L_3, L_4, L_5\}$ then:
a) $\mathcal{S} = \{\{L_1, L_2\}, \{L_3, L_4, L_5\}\}$ is a valid domain split;
b) $\mathcal{S} = \{\{L_1\}, \{L_3, L_4, L_5\}\}$ is not a valid domain split because L_2 is not present in any element of $\mathcal{S}$, thus $\mathcal{S}$ fails to be a partition of L;
c) $\mathcal{S} = \{\{L_1, L_3\}, \{L_2, L_4, L_5\}\}$ is not a valid domain split, that's because neither $\{L_1, L_3\}$ nor $\{L_2, L_4, L_5\}$ are contiguous sub-sequences of L. □

Since the elements of a split $\mathcal{S}$ are contiguous sub-sequences of the ordered sequence of leaves, for any $s \in \mathcal{S}$ the union of all the leaves in s gives a valid domain—ie. a range of Hilbert-keys—and one can regard the elements of $\mathcal{S}$ indistinctively as sub-sequences of L as well as domains. On the other hand if the size of the domain split $\mathcal{S}$ equals the number of MPI processes p, then one can match one MPI processes to exactly one element of the $\mathcal{S}$ and that would constitute a valid domain decomposition.

EXAMPLE 2.3. Let $L = \{L_1, L_2, L_3, L_4, L_5\}$ be the ordered list of leaves of the top-tree and assume there are $p = 2$ MPI processes. Then if one chooses a split of size $|\mathcal{S}| = 2$, $\mathcal{S} = \{\{L_1, L_2\}, \{L_3, L_4, L_5\}\}$, one can assign to process with rank 0 the domain $d_0 = L_1 \cup L_2$ and the process with rank 1 the domain $d_1 = L_3 \cup L_4 \cup L_5$. Then all particles in P_{d_0} and P_{d_1} will be owned by processes with respective ranks 0 and 1. □

Intuitively the domain decomposition should produce a distribution of particles among processes in such a way that every process owns no more particles than its memory capacity and during the subsequent time step evolution the computational burden of each one of them will be almost the same. Therefore, the quality of a domain decomposition will be based on two maps: $\mathsf{work}\colon P \to \mathbb{R}^+$ that gives the computational work required for one time step evolution of each particle and $\mathsf{load}\colon P \to \mathbb{R}^+$ that gives the memory necessary to host each particle. For any domain d one can define $\mathsf{work}(d)$ as the sum of the work of all particles contained in that domain

$$\mathsf{work}(d) = \sum_{i \in P_d} \mathsf{work}(i);$$

and similarly the load of a domain d is

$$\mathsf{load}(d) = \sum_{i \in P_d} \mathsf{load}(i).$$

A domain split $\mathcal{S}$ can then be judged by how well distributed is the load and work among its constituents.

DEFINITION 2.5. The **work imbalance** of a domain split $\mathcal{S}$ is defined as:

$$\eta(\mathsf{work}, \mathcal{S}) = \frac{\sup_{s \in \mathcal{S}} \mathsf{work}(s)}{\frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \mathsf{work}(s)}. \tag{1}$$

DEFINITION 2.6. The **load imbalance** of a domain split $\mathcal{S}$ is defined as:

$$\eta(\mathsf{load}, \mathcal{S}) = \frac{\sup_{s\in\mathcal{S}} \mathsf{load}(s)}{\frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} \mathsf{load}(s)}. \tag{2}$$

Notice that the work imbalance $\eta(\mathsf{work}, \mathcal{S})$ is bounded in the real interval $[1, |\mathcal{S}|]$. The equality $\eta(\mathsf{work}, \mathcal{S}) = 1$ happens if and only if all the elements of the domain split $\mathcal{S}$ have the same work, in that case it is said that $\mathcal{S}$ is *perfectly balanced*. On the contrary, $\eta(\mathsf{work}, \mathcal{S}) = |\mathcal{S}|$ if and only if there is exaclty one domain $d \in \mathcal{S}$ with $\mathsf{work}(d) > 0$ while for any other $s \in \mathcal{S}$ ($s \neq d$): $\mathsf{work}(s) = 0$, in that case the domain split is *perfectly unbalanced*. A similar remark can be extracted from the definition of the load imbalance $\eta(\mathsf{load}, \mathcal{S})$.

EXAMPLE 2.4. Let $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$ and $\mathsf{work}(s_i) = 2, 0, 3, 2$ for $i = 1, 2, 3, 4$ respectively. The work imbalance of $\mathcal{S}$ is then $\eta(\mathsf{work}, \mathcal{S}) = 3/(1/4 \cdot 7) = 12/7$. □

DEFINITION 2.7. The size of the domain split—ie. the number of domains—is controlled by an integer quantity called **multiple-domains parameter** $\mathcal{M}$, then $|\mathcal{S}| = p \cdot \mathcal{M}$.

Having $\mathcal{M} > 1$ implies, by pigeon-hole principle, that some processes will have to be matched to more than one domain, and then it becomes necessary to introduce another object to represent that correspondence which can no longer be one-to-one.

DEFINITION 2.8. A **domain assignment** is a map $\mathcal{A}\colon \mathcal{S} \to \mathcal{X} \subset \mathbb{N}$, that represents a match of each one of the $p \cdot \mathcal{M}$ domains of the split $\mathcal{S}$ to exactly one the MPI processes in $\mathcal{X}$ (all processes in the simulation).

Similar to the split, a domain assignment $\mathcal{A}$ can be judged by how well distributed is the load and work among its p target elements.

DEFINITION 2.9. The **work imbalance** of a domain assignment $\mathcal{A}$ is defined as:

$$\eta(\mathsf{work}, \mathcal{A}) = \frac{\sup_{i\in\mathcal{X}} \mathsf{work}(i)}{\frac{1}{p}\sum_{i\in\mathcal{X}} \mathsf{work}(i)}; \tag{3}$$

where

$$\mathsf{work}(i) = \sum_{s\in\mathcal{A}^{-1}(i)} \mathsf{work}(s).$$

DEFINITION 2.10. The **load imbalance** of a domain assignment $\mathcal{A}$ is defined as:

$$\eta(\mathsf{load}, \mathcal{A}) = \frac{\sup_{i\in\mathcal{X}} \mathsf{load}(i)}{\frac{1}{|p|}\sum_{i\in\mathcal{X}} \mathsf{load}(i)}; \tag{4}$$

where

$$\mathsf{load}(i) = \sum_{s\in\mathcal{A}^{-1}(i)} \mathsf{load}(s).$$

EXAMPLE 2.5. Let $p = 2$ and $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$, while the functions $\mathsf{work}(s_i) = 2, 0, 3, 2$ and $\mathcal{A}(s_i) = 0, 1, 0, 1$ for $i = 1, 2, 3, 4$ respectively. The work of process with rank 0 is $\mathsf{work}(0) = \mathsf{work}(s_1) + \mathsf{work}(s_3) = 2 + 3 = 5$, and the work of process with rank 1 is $\mathsf{work}(1) = \mathsf{work}(s_2) + \mathsf{work}(s_4) = 0 + 2 = 2$. The work imbalance of the assignment $\mathcal{A}$ is then $\eta(\mathsf{work}, \mathcal{A}) = 5/(1/2 \cdot 7) = 10/7$. □

It might be intuited that there must be a relationship between the imbalance of the split $\mathcal{S}$—either load or work—and the imbalance that could be obtained for the assignment $\mathcal{A}$. To illustrate this idea consider the example on which the $\eta(\mathsf{load}, \mathcal{S}) = 1$, then every $s \in \mathcal{S}$ have the same load and no matter how $\mathcal{A}$ is choosen, the equality $\eta(\mathsf{load}, \mathcal{A}) = 1$ will hold. That relation is better understood through the followin Lemma:

LEMMA 2.1. The imbalance of the assignment $\mathcal{A}$ is never greater than the imbalance of the split $\mathcal{S}$ from which the assignment is built:

$$\eta(X, \mathcal{A}) \leq \eta(X, \mathcal{S}), \qquad \text{for } X = \mathsf{load}, \mathsf{work}\,.$$

See Appendix D for a proof.

Summarizing, the outcome of a domain decomposition is completely determined by three objects alone: the top-tree $\mathcal{T}$, the domain split $\mathcal{S}$ and the domain assign $\mathcal{A}$. The routines within the module `domain` must produce a particular choice of the triple $(\mathcal{T}, \mathcal{S}, \mathcal{A})$ based on the given particles P, and their properties key, work and load. While the quality of that outcome should be judged through the work / load imbalanced of $\mathcal{A}$.

Here is the workflow of the domain decomposition implemented on PGADGET-3:

```
domain_decomposition()
    call determineTopTree(), to build the top-tree and produce the sequence of leaves;
    call findSplit_work_balanced(), to split the leaves into balanced domains according
    to work;
    call assign_load_or_work_balanced(), to assign domains to MPI processes;
    if that domain decomposition violates the memory constraints, then:
        call findSplit_load_balanced(), to split the leaves into balanced domains accord-
        ing to load;
        call assign_load_or_work_balanced(), to assign domains to MPI processes;
    while there are particles on a different process different from that assigned:
        call countToGo(), to select as many particles for export that don't violate memory
        contraints;
        call exchange(), to send the particles to their respective process;
```

Listing 2. PGADGET-3's domain decomposition.

§3. ISSUES

No computer code is ever finished, unless it becomes obsolete. GADGET is no exception, and its own life cycle is all but ending. There's constant improvement of it's core algorithms as the research becomes more demanding on the accuracy of results and as the computer machines grow in the number of processing elements. Not least important are the efforts of the maintainers to get rid of bugs, and increase the robustness and stability of the code. This thesis constitutes the latest branch of improvements to the domain decomposition module. That said, the moment is right to remark the weak points and key issues on this module that could be improved in the light of the forthcoming exa-scale challenge.

Scalability.

The domain decomposition as it is currently implemented, does not scale with the number of processes. For a fixed number of particles, the time it takes for the domain decomposition to finish hardly decreases with inverse proportionality relative to the number of MPI processes—see figure 2. A time profile of the domain decomposition runing on 1 and 5 computing nodes* respectively, using an initial state file with 2×10^8 particles shows that the share of time spent on the routine `exchange` is the main cause for this bad scalability—see tables 1 and 2 and figure 3. It is so because it implements an *all-to-all* communication pattern among the MPI processes participating in the simulation which leads to a linear time delay $\mathcal{O}(p)$, where p is the number of processes. If this linear trend is kept—see fig. 4—a domain decomposition in an exa-scale machine ($p \approx 10^5$) would take more than a dozen minutes to complete.

* all the runs on this thesis where performed on a high performance computer constituted of 48 cores per node. The details of this machine are given at the begining of Chapter 4

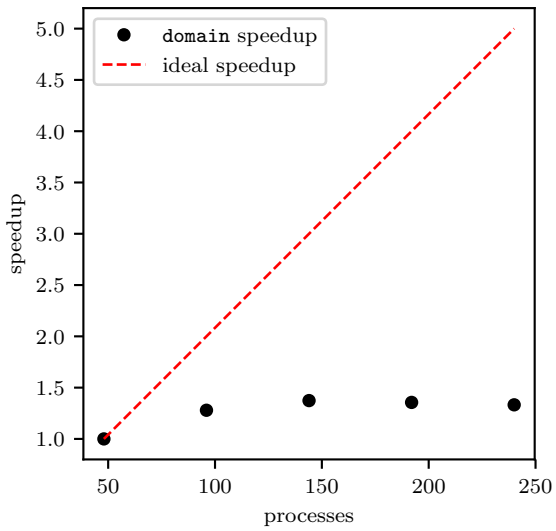

Figure 2. The graph shows the strong scalability of `domain` in terms of speedup. The number of particles $\approx 2 \times 10^8$ is fixed, and number of MPI processes varies from 48 to 240.

routine	time (s)	% of total
`domain`	5.91	100.0
`exchange`	1.87	31.7
`determineTopTree`	2.68	45.4
`countToGo`	0.46	7.8
other	0.90	15.2

Table 1. Domain profile, 1 node, 48 cores.

routine	time (s)	% of total
`domain`	4.43	100.0
`exchange`	3.52	79.4
`determineTopTree`	0.57	12.9
`countToGo`	0.13	2.8
other	0.22	4.9

Table 2. Domain profile, 5 nodes, 240 cores.

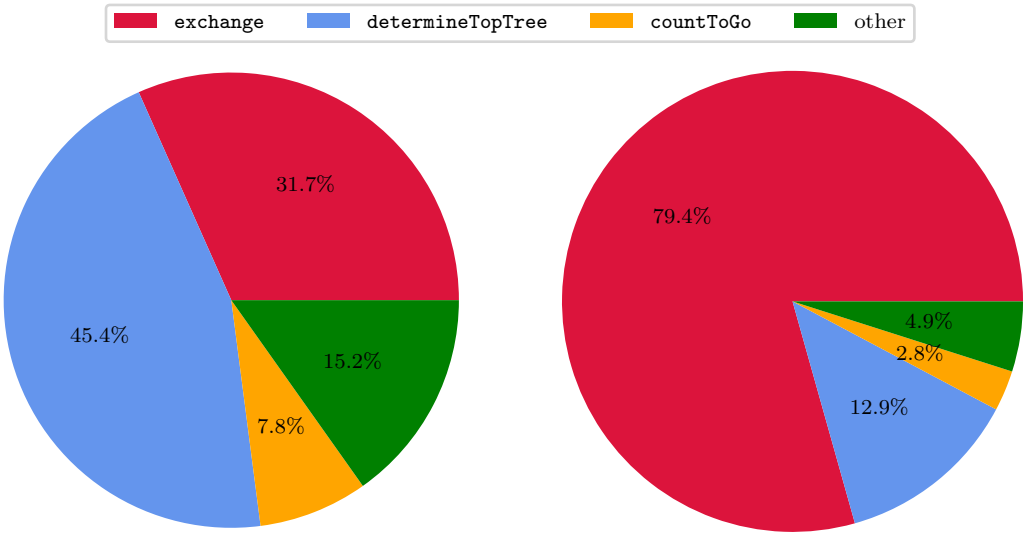

Figure 3. Profile of the domain decomposition, time consumed by each routine.

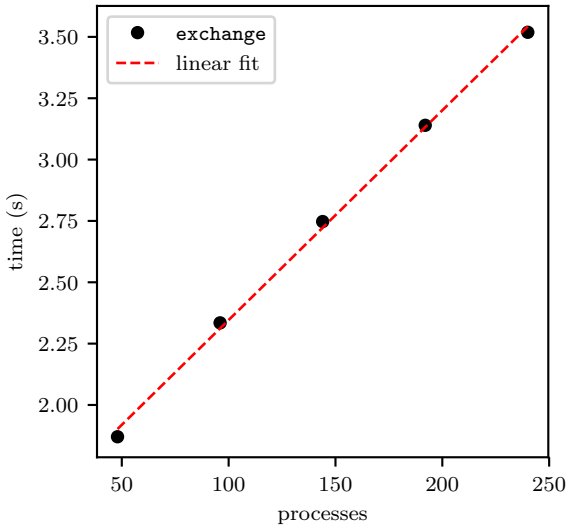

Figure 4. The graph shows the strong scalability of the `exchange` routine in terms of time elapsed. There is a clear linear trend. The number of particles $\approx 2 \times 10^8$ is fixed, and number of MPI processes varies from 48 to 240.

Communication patterns in the construction of the top-tree.

Another issue regards the routine `determineTopTree` that constructs the top-tree. This routine starts by computing a local top-tree using only the local particles P^{local}. Then it merges the trees from all MPI processes into a single one.

This approach can be argued to be unstable—see section §4.4. In the sense that the algorithm does not guarantee that the top-tree obtained at the end is a function of the particles P alone. The same configuration of particles P, initially distributed in different manners among MPI processes, results in different top-trees.

On the other hand, the merging of the trees is implemented using a crafted communication pattern with a time delay of growth $\mathcal{O}(p)$. Being an associative operation, merging trees can be re-designed to a lower complexity of $\mathcal{O}(\log p)$.

Domain split and assignment.

It was previously discussed that the quality of the domain decomposition is evaluated through the load and work imbalance of the assignment $\mathcal{A}$. The algorithm that computes $\mathcal{A}$ takes as input the split $\mathcal{S}$ found in a previous step of the domain decomposition. If $\mathcal{S}$ is well balanced then $\mathcal{A}$ in turn will also be balanced. That's why there is a clear intention of the developers to perform $\mathcal{S}$ in the best way possible to minimize the work or load imbalance. While minimizing the work imbalance produces a gain in performance during the simulation, the load imbalance, on the other hand, is a necessity because there are physical limits to the memory capacity on each processes. That implies that the domain decomposition can be performed if and only if the load on each element of the split does not exceed a given value $\mathsf{load}_{\max}$.

The domain split on PGADGET-3 is perfomed with a combination of two routines:

- `domain_findSplit_work_balanced`, and
- `domain_findSplit_load_balanced`.

The first combines the functions work and load into $\mathsf{wl} = \mathsf{work} + \mathsf{load}$ and searches for the split that approximately minimizes the wl imbalance, it does that in a single greedy scan of the array of values of wl on the leaves. If the resulting split violates the $\mathsf{load}_{\max}$ constraint, then the second routine is called, which does a similar scan of the array of load to find a split that approximately minimizes the load imbalance. If the resulting split exceeds again $\mathsf{load}_{\max}$ the program stops with and error message indicating that no split could be found that satisfies the memory constraints.

The problem with this approach is that there is no robust strategy in the search for the split that satisfies the $\mathsf{load}_{\max}$ constraint. These greedy routines neither guarantee that any solution will be found in case of existence nor that the result will be optimal.

On the other hand, the domain assignment, which comes after $\mathcal{S}$ has been found, is performed with the intention to further reduce the load and work imbalance of the final domain decomposition. Unfortunately, the current implementation of that routine has an algorithmic complexity of $\mathcal{O}(p^2\mathcal{M})$ that becomes unpractical for $p \sim 10^5$ or greater.

OpenMP.

The given code is paralellized using the distributed memory paradigm through MPI alone. It could be worth to implement a hybrid MPI+OpenMP parallel version, to profit from the benefits of multiprocessors with shared memory hardware. For example, one MPI processes could run on each computer socket which is a shared memory region, and saturate the computing capacity of its multicore architecture spawning multiple threads. The MPI communications, heavily dependent on the number of processes, would greatly reduce their time of execution. Without mentioning the fact that the rest of the code is already undergoing hybrid MPI+OpenMP parallelization, and leaving the domain decomposition thread-serial would be a waste of computational resources and could affect the scalability of the whole code.

Chapter III

IMPLEMENTATION

A GOOD DOMAIN DECOMPOSITION is vital for the performance of GADGET. It must provide a near equally distributed computational work and memory requirements among MPI processes and yet it should be fast enough so that the whole procedure can be called on a regular basis as particles change their location in space. This chapter presents the theoretical and practical aspects of the implementation of the domain decomposition that the author proposes to the GADGET collaboration.

As it was mentioned in section §2.2, the domain decomposition is uniquely determined by the triple $(\mathcal{T}, \mathcal{S}, \mathcal{A})$. Thus, each of the following sections of this chapter will focus on the choice of each of the elements of that triple to satisfy an overall strategy. The principles of that strategy are:

1. The top-tree $\mathcal{T}$ should be deeper for those regions of space more densely populated, with respect to the regions which are not.
2. The top-tree $\mathcal{T}$ should be a function of the set of particles alone P, ie. taking into account their positions in space independently of how their are distributed among MPI processes.
3. The domain split $\mathcal{S}$ should minimize the work imbalance while constraining the load imbalance under a fixed threshold.
4. The assignment $\mathcal{A}$ is to be performed such that the total number of particles changing the host processes be minimized or reduce the imbalance even further, depending on what's needed at runtime.

§1. TOP-TREE

THE TOP-TREE is chronologically the first important element for the domain decomposition (see definition 2.3). The architectural choices made for its construction will affect the result of the subsequent domain routines and performance of the time steps of the simulation that follow the decomposition. Hence the *quality* of the tree is of utter importance. It must follow a precise set of rules that lead in every situation to a partition of space into sub-domains in agreement with the principles previously stated.

To ensure the first principle is satisfied we define the quantities

$$\mathsf{work}_{\text{limit}} = \frac{\mathsf{work}(P)}{p\mathcal{M}\alpha_{\mathcal{T}}}, \quad \mathsf{load}_{\text{limit}} = \frac{\mathsf{load}(P)}{p\mathcal{M}\alpha_{\mathcal{T}}};$$

denominated the work and load limits respectively, where $\alpha_{\mathcal{T}}$ is the **top-tree allocation factor** that can be tuned accordingly by the user as a parameter of the simulation. We then impose that: *a vertex v in the top-tree $\mathcal{T}$ cannot be a leaf if* $\mathsf{work}(v) > \mathsf{work}_{\text{limit}}$ *or* $\mathsf{load}(v) > \mathsf{load}_{\text{limit}}$*, except when v contains a single Hilbert-key*, in which case it is not possible to divide v into smaller sub-domains. That statement will be referred to as the **leaf condition** and it is in fact not different from the one already present in GADGET's `domain` module.

In order to *know* what the work and load functions are for a vertex $v \in \mathcal{T}$, the sums $\sum_{i \in P_v} \mathsf{work}(i)$ and $\sum_{i \in P_v} \mathsf{load}(i)$ must be computed. But, the particles of a cosmological simulation are distributed among MPI processes. Our approach to solve that problem is to compute for every $v \in \mathcal{T}$ the $\mathsf{work}(v)$ and $\mathsf{load}(v)$ using only local particles P^{local} and then *all-reduce* the entire tree among MPI processes. To be precise, a top-tree $\mathcal{T}$ is stored in memory as an array of vertexes where each vertex $\mathcal{T}_i$ can accessed with an index $0 \leq i < |\mathcal{T}|$, and at the memory location of $\mathcal{T}_i$ the values of $\mathsf{work}(\mathcal{T}_i)$ and $\mathsf{load}(\mathcal{T}_i)$ as well as other information concerning the topology of $\mathcal{T}$ are stored. Then two top-trees $\mathcal{T}$ and $\mathcal{T}'$ with the same topology and memory layout can be *reduced* into a third top-tree $\mathcal{T}''$ in two steps. Starting with an identical tree

$$\mathcal{T}'' = \mathcal{T};$$

and then for every $0 \leq i < |\mathcal{T}|$

$$\begin{aligned} \mathsf{work}(\mathcal{T}_i'') &= \mathsf{work}(\mathcal{T}_i) + \mathsf{work}(\mathcal{T}_i'), \\ \mathsf{load}(\mathcal{T}_i'') &= \mathsf{load}(\mathcal{T}_i) + \mathsf{load}(\mathcal{T}_i'). \end{aligned}$$

This reduction operation is clearly associative, therefore it can implemented into an `MPI_Op` object of the Message Passing Interface [MPI], and then let the MPI library perform the *all-reduce* on our behalf by calling `MPI_Allreduce` which is optimized to have an assymptotical complexity of $\mathcal{O}(\log p)$ communications. In addition, this reduction operation is free of branches and it is cache-friendly.

The top-tree is then constructed simultaneously on every MPI process. Starting from a tree with a single vertex, the root, that holds all the Hilbert-keys and then iteratively enlarging the tree when the current leaves do not satisfy the leaf condition.

```
iterative_toptree(load, work, P^local, load_limit, work_limit):
    initialize root as the set of all Hilbert-keys;
    T = {root};
    repeat:
        for each v ∈ T:
            if v is a leaf and work(v) > work_limit or load(v) > load_limit then:
                create 8 new vertexes children of v;
                for each u child of v;
                    work(u) = work(v)/8;
                    load(u) = load(v)/8;
        if no new vertexes were added then:
            exit the loop;
        for each v ∈ T:
            work(v) = Σ_{i∈P_v^local} work(i);
            load(v) = Σ_{i∈P_v^local} load(i);
        all-reduce T;
    return T;
```

Listing 3. Iterative construction of the top-tree.

At the begining of every iteration of the main loop of the algorithm (see listing 3) all MPI processes are guaranteed to have in local memory the same top-tree $\mathcal{T}$. Then all the leaves are checked for compliance with the leaf condition. Every leaf vertex that fails that condition becomes the father of 8 new leaves, whose work and load will be estimated as $\frac{1}{8}$ of the father's. If no new vertexes where added in the previous step, then the loop finishes. Otherwise the values of $\mathsf{work}(v)$ and $\mathsf{load}(v)$ are computed exactly for every vertex $v \in \mathcal{T}$, using the aforementioned reduction operation. This algorithm guarantees that the top-tree is exactly the same independently of the local particles P^{local} on the processes. Furthermore, the data structure that holds the tree is also preserved and it will be identical for all the MPI processes, making possible the use of the reduction previously mentioned.

§2. DOMAIN SPLIT

According to the third principle of our strategy for the domain decomposition the algorithm that computes the domain split $\mathcal{S}$ should minimize the work imbalance while keeping the load imbalance below a certain value. The reason behind this choice is that in a parallel execution the time to solution equals the time elapsed from the begining of the run until the last process finishes the execution. Therefore, by minimizing $\eta(\mathsf{work}, \mathcal{S})$ also $\sup_{s\in\mathcal{S}} \mathsf{work}(s)$ is minimized—see equation (1)—and since the time to solution is proportional to it, then the 3rd principle tries to reduce the time to solution for every time step after the domain decomposition. The constraint on $\eta(\mathsf{load}, \mathcal{S})$ is necessary to ensure that the partition of the domain will not lead to some processes carrying more particles than those allowed by the physical memory limit. In a formal language the domain split algorithm solves the following optimization problem:

PROBLEM 3.1. *Give a sequence of leaves L, the functions of* work *and* load *evaluated over the leaves, and two numbers η_l and n: find a domain split $\mathcal{S}_o$ that satisfies $|\mathcal{S}_o| = n$, $\eta(\mathsf{load}, \mathcal{S}_o) \leq \eta_l$ and $\eta(\mathsf{work}, \mathcal{S}_o)$ is minimal.*

Before presenting the algorithm that finds the optimal split $\mathcal{S}_o$, some preparations are necessary. First it is worth trying to solve a simpler problem:

PROBLEM 3.2. *Give a sequence of leaves L, the functions of* work *and* load *evaluated over the leaves, and three numbers η_l, η_w and n: find a domain split $\mathcal{S}_o$ that satisfies $|\mathcal{S}_o| = n$, $\eta(\mathsf{load}, \mathcal{S}_o) \leq \eta_l$ and $\eta(\mathsf{work}, \mathcal{S}_o) \leq \eta_w$.*

Problem 3.2 can be solved with the following greedy algorithm:

```
greedy_split(L, work, load, η_l, η_w, n):
    S_o = ∅;
    w_max = (η_w / n) Σ_{d∈L} work(d);
    l_max = (η_l / n) Σ_{d∈L} load(d);
    while |S_o| < n:
        s = the longest prefix of L such that work(s) ≤ w_max and load(s) ≤ l_max;
        insert s into S_o;
        remove s from L;
    if L = ∅ then:
        return S_o;
    otherwise:
        return no solution;
```

Listing 4. Greedy split with constraints.

The proof of Listing 4 can be found in Appendix D.

The `greedy_split` algorithm not only constructs a particular $\mathcal{S}$, but also tells about the existence of the solution for the given constraints η_w and η_l. Notice that if $\mathcal{S}_o$ is an optimal solution, by definition the predicate (5) is true for any $\eta_w \geq \bar{\eta}_w$, where $\bar{\eta}_w = \eta(\mathsf{work}, \mathcal{S}_o)$.

$$\exists\, \mathcal{S}\colon \eta(\mathsf{work}, \mathcal{S}) \leq \eta_w \wedge \eta(\mathsf{load}, \mathcal{S}) \leq \eta_l; \tag{5}$$

That's because we can choose $\mathcal{S} = \mathcal{S}_o$; while it remains false for any $\eta_w < \bar{\eta}_w$ simply because that would imply that $\mathcal{S}_o$ is not the optimal solution. The logical expression (5) becomes then a binary-valued optimality function of the work imbalance parameter η_w (see figure 5) and the search for an optimal split reduces to the search of the lowest value of η_w for which the optimality function is true, ie. $\eta_w = \bar{\eta}_w$. Since η_w itself is bounded to the region $[1, n]$ a binary search can be used to find $\bar{\eta}_w$. Then the algorithm that solves problem 3.1 stated at the begining of this section is formulated in Listing 5.

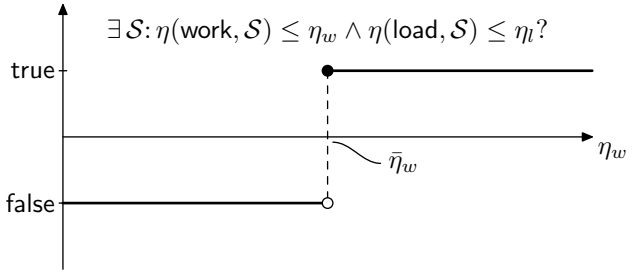

Figure 5. The optimality function.

```
bs_split(L, work, load, η_l, n):
    S_o = greedy_split(L, work, load, η_l, η_w = n, n);
    if S_o is not a valid solution, then:
        return no solution;
    a = 1, b = n;
    while (b − a) > ϵ:
        m = (b + a)/2;
        S = greedy_split(L, work, load, η_l, η_w = m, n);
        if S is a valid solution, then:
            S_o = S;
            b = m;
        otherwise:
            a = m;
    return S_o;
```

Listing 5. Binary search split with constraints.

The time to solution of the method `bs_split` is proportional to the number of iterations in the binary search $\log(p\mathcal{M}/\epsilon)$ and the time for one call of `greedy_split`, where p is the number of MPI processes and $\mathcal{M}$ is the multiple-domains factor, and $p\mathcal{M} = |\mathcal{S}_o|$ is the total number of domains as well as the maximum value the imbalance η_w can take. On the other hand, the execution time of `greedy_split` is proportional to the number of leaves $|L|$. Then the algorithmic complexity of `bs_split` is $\mathcal{O}\big(|L|\log(p\mathcal{M}/\epsilon)\big)$. In a typical GADGET simulation the number of leaves of the top-tree is greater than the number of MPI processes, ie. $|L| \gg p\mathcal{M}$. Hence, the time complexity can be reduced even further by using cummulative arrays and another binary search inside `greedy_split` to find prefixes. That leads to an overall time complexity $\mathcal{O}\big(|L| + p\mathcal{M}\log(|L|)\log(p\mathcal{M}/\epsilon)\big)$.

§3. DOMAIN ASSIGN

GRAVITY IS AN ATTRACTIVE FORCE, that makes massive point particles in a simulation to cluster together in some regions of space while leaving some patches almost empty. Hence, a typical cosmological simulations at low *redshift*†, will be characterized by a highly inhomogeneous distribution of particles. If a domain decomposition was only about equal sharing of the number particles among MPI processes, the top-tree construction and the previously discussed split of the leaves would be enough. The truth is that particles which *live* in densely populated regions of space, being subjected to stronger and constantly changing forces, would be computationally more expensive than others lying in void regions. This implies that there is no linear relation between the load and work functions, and it can happen that the best split of the sequence of leaves does not yield a good work balance.

EXAMPLE 3.1. Suppose there are $p = 2$ MPI processes and each one of them cannot host more than 7 particles and the sequence of leaves of the top-tree has the following work and load values:

L index:	1	2	3	4	5	6	7	8
load:	1	1	1	1	1	1	3	3
work:	1	1	1	1	1	1	9	9

The only possible split that satisfies the memory constraint, is $\mathcal{S} = \{s_1 = \{1,2,3,4,5,6\}, s_2 = \{7,8\}\}$, with the following work/load values: $\mathsf{load}(s_1) = 6$, $\mathsf{work}(s_1) = 6$, $\mathsf{load}(s_2) = 6$, $\mathsf{work}(s_2) = 18$, then the work imbalance is $\eta(\mathsf{work}, \mathcal{S}) = 18/(1/2(6+18)) = 1.5$. In this case one process will have 3 times more work than the other. □

To reduce the imbalance in those situations, GADGET's developers decided to introduce the concept of multiple-domains (see definition 2.7). The idea is that having more splits $|\mathcal{S}| = p\mathcal{M}$ instead of p, where $\mathcal{M}$ is the multiple-domains integer parameter, one can assign several domains to each process in a clever way to minimize the imbalance. For simplification one always imposes that every processes has to be matched to exactly $\mathcal{M}$ domains.

† *redshift* is a function of the cosmological time, cosmological systems evolve from high to low values of the redshift.

EXAMPLE 3.2. Consider again the situation of example 3.1, with $\mathcal{M} = 2$. Then we can have $\mathcal{S} = \{s_1 = \{1, 2, 3\}, s_2 = \{4, 5, 6\}, s_3 = \{7\}, s_4 = \{8\}\}$, and by assigning s_1 and s_3 to process 1 and s_2 and s_4 to process 2, one has that the load function satisfies the memory constraints: $\mathsf{load}(s_1 \cup s_3) = 6$, $\mathsf{load}(s_2 \cup s_4) = 6$. While $\mathsf{work}(s_1 \cup s_3) = 12$, $\mathsf{work}(s_2 \cup s_4) = 12$ implies that the work is perfectly balanced among processes. □

It can be argued that the choice of an assignment function that minimizes the imbalance, leads to a domain decomposition procedure that everytime it runs, it will reshuffle particles among processes. Therefore in the situations when GADGET runs on 10^5 processes, the `exchange` part of the domain decomposition, which involves an *all-to-all* communication pattern will become a troublesome bottleneck for the execution times. Already for 240 processes, the `exchange` takes 3 quarters of the execution time (see figures 3 and 4), and if the linear trend is extrapolated to a higher number of processes—a typical domain exchange involving 2×10^8 particles takes around 3.5 seconds when running on 240 cores of a High Performance Computer—it can be roughly estimated that for 10^5 processes the delay would be $3.5 \times 10^5/240$ seconds ≈ 25 minutes for a single domain decomposition.

The existence of a *one-size-fits-all* solution to our balance and timing problem is unlikely. Therefore, the 4th principle of our strategy—at the begining the chapter—is chosen as a guideline for the domain assignment. It is then proposed the construction of two assignment functions: $\mathcal{A}_{\text{fast}}$ and $\mathcal{A}_{\text{good}}$. The first one is such that the domain `exchange` involves few intra processes communications regardless of the value of the work imbalance. The other kind of assignment will instead minimize the work imbalance of the processes, possibly leading to a complete reshuffle of the particles and $\mathcal{O}(p)$ communications per process during the domain `exchange` in the worst case. The main routine of the domain decomposition will then decide whether $\mathcal{A}_{\text{good}}$ should be used, based on how bad is the imbalance if $\mathcal{A}_{\text{fast}}$ is used instead.

The search for an assignment $\mathcal{A}_{\text{fast}}$ that minimizes the total number of particles exchanges, is equivalent to a bipartite matching problem with minimum cost, with the bipartite graph $\mathcal{S} \cup \mathcal{X}$ and using tne number of particles exchanges as cost function. Unfortunately, the best known algorithm that solves that problem: Kunh-Munkres' *Hungarian* algorithm, has a cubic complexity dependence on the number of vertexes $\mathcal{O}(p^3)$. For $p > 10^2$ it becomes unpractical.† Due to this practical limitation, $\mathcal{A}_{\text{fast}}$ is better built as the same assignment from the previous domain decomposition. If one takes into consideration that after each time step, particles move from their initial positions by a small amount in comparison to the simulation box size, it might be expected that only a tiny fraction of them move outside their starting domains and into neighboring regions that share a common border. Hopefully, by requiring simply a re-adjustment of the particle distributions, the following domain decomposition based on $\mathcal{A}_{\text{fast}}$ would lead to every processes communicating with $\mathcal{O}(1)$ other processes—independently of p.

On the other hand, the assignment $\mathcal{A}_{\text{good}}$, that further reduces the work imbalance is properly defined as the solution to the following optimization problem.

> PROBLEM 3.3. *Given the sets $\mathcal{S}$ and $\mathcal{X}$, and the function $\mathsf{work}\colon \mathcal{S} \to \mathbb{R}^+$, with $|\mathcal{S}| = p\mathcal{M}$ and $|\mathcal{X}| = p$: find a domain assignment $\mathcal{A}\colon \mathcal{S} \to \mathcal{X}$ such that every proceses $i \in \mathcal{X}$ is assigned to exactly $\mathcal{M}$ domains, $|\mathcal{A}^{-1}(i)| = \mathcal{M}$, and the work imbalance $\eta(\mathsf{work}, \mathcal{A})$ is minimal.*

If the restriction $|\mathcal{A}^{-1}(i)| = \mathcal{M}$ is lifted, this problem is known as *multiprocessor scheduling* and it has been proved to be NP-Complete [Garey, 1979]. It is reasonable then to search for an approximate method such that the imbalance η obtained is very close to the optimal η_o. [Graham, 1969] has proven that the upper bound $\frac{\eta}{\eta_o} \leq \frac{4}{3} - \frac{1}{p}$ holds if for each domain $s \in \mathcal{S}$ in decreasing order of work: s is assigned to the process in $\mathcal{X}$ with the smallest cumulative work. Using a logarithmic update-query data structure such as a red-black tree, the previous algorithm can be implemented with a complexity $\mathcal{O}(p\mathcal{M} \log p)$. Listing 6 sketches that algorithm and a possible implementation can be found in Appendix C.

† For details about bipartite matching and the *Hungarian* algorithm see [Ravindra, 1993] and [TopCoder].

```
greedy_assign(𝒮, work, p, ℳ):
    𝒜 = ∅;
    q = ∅;
    for each i ∈ 𝒳:
        insert i with work w = 0 into q;
    sort 𝒮 increasingly according to work;
    for each s ∈ 𝒮:
        extract from q the pair (i, w) with lowest w;
        𝒜(s) = i;
        w = w + work(s);
        if the number of assignments of i is less than ℳ:
            insert (i, w) into q;
    return 𝒜;
```

Listing 6. Greedy domain assign, according to [Graham, 1969].

The keen reader may have noticed that the `greedy_assign` routine does not take into consideration the load function and minizes the work imbalance alone. This decision has been consciously taken in order to simplify the assignment problem, and because according to Lemma 2.1 once the load imbalance of the split has been fixed, then the load imbalance of whathever assignment will be bounded by that value. Thus the constraint on the load imbalance is left entirely to be taken care off by the `bs_split` routine.

Chapter IV

RESULTS

To support the claims presented in this thesis, a git repository was created from what was previously defined as GADGET's *core* (see §2.1), extracted from the PGADGET-3 code. The pristine version of that code will be hereby referred to as the `master` branch of that repository, while all further modifications due to the author's work were implemented in the `dev` branch, unless otherwise noted. This Chapter overviews all sorts of benchmarks to compare both branches using metrics of interest for the developers of GADGET, such as time to solution, scalability, robustness and *quality* of the results—based on the imbalance parameters presented in section §2.2. All tests were performed on a high performance computer with Intel® Xeon® Gold 5118 CPU @ 2.30GHz processors with 12 cores each, every compute node on this machine has 4 sockets for a total of 48 cores per node, and its memory hierarchy is displayed below:

```
$ lstopo
Machine (512GB total)
  Package L#0
    NUMANode L#0 (P#0 128GB)
    L3 L#0 (17MB)
      L2 L#0 (1024KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
 ...
```

§1. UNIT TEST: SPLIT

In the previous chapter a thorough explanation was given for an algorithm to find the optimal domain split with constraints (see §3.2). PGADGET-3's routine to compute the domain split, on the other hand, was found to be non-robust because it doesn't guarantee that a solution satisfying the constraints will be found, and when it does find a solution it is not necessarily optimal (see §2.3). Both split implementations have been tested using randomly generated data, each one of which consists in two input arrays: work and load of the sequence of leaves. Figure 6 shows the results for multiple test cases. Each point represents a single test case, where the x-coordinate is the value of the work/load imbalance found using the `master` branch routines and the y-coordinate corresponds to the imbalance found by the routines in `dev`. The test cases are divided into 12 classes according to $\eta_l = \eta(\mathsf{load})_{\text{limit}}$ constraint and the way the work and load arrays were generated. The labels have the following meanings:

- "constant-random" indicates that the test cases where constructed with a constant value in the array of the load and random values in the array of the work;
- "random-constant" indicates that the test cases where constructed with random values in the array of the load and a constant work;
- "random-random" indicates that the test cases where constructed with random values in the array of the load and uncorrelated random values in the array of work;
- finally, "random-equal" indicates that the test cases where constructed with random values in the array of the load and work was taken equal to the load, to have a perfect correlation.

In order to explore a wider region in the space possible situations, the tests had values of $|\mathcal{S}|$ ranging from 10 to 1000, and $|L|$ ranging from $|\mathcal{S}|$ to $200 \times |\mathcal{S}|$.

The plots in fig. 6 confirm the the issue regarding GADGET's split being sub-optimal. There are no blue dots below the line $x = y$, that means that the implementation of the domain split with binary search on the `dev` branch gives a work imbalance always less than the `master`'s. Some of the test cases, for instance "random-random" with $\eta(\mathsf{load}) \leq 2.0$, put the split routines into such stress that `master`'s solution returns with a work imbalance greater than 4 while the optimal imbalance is found below 1.5. When the load imbalance limit is dialed down to 1.1, there are situations for which `master` fails to find a solution, even though the solution does exist, clearly those points are not shown.

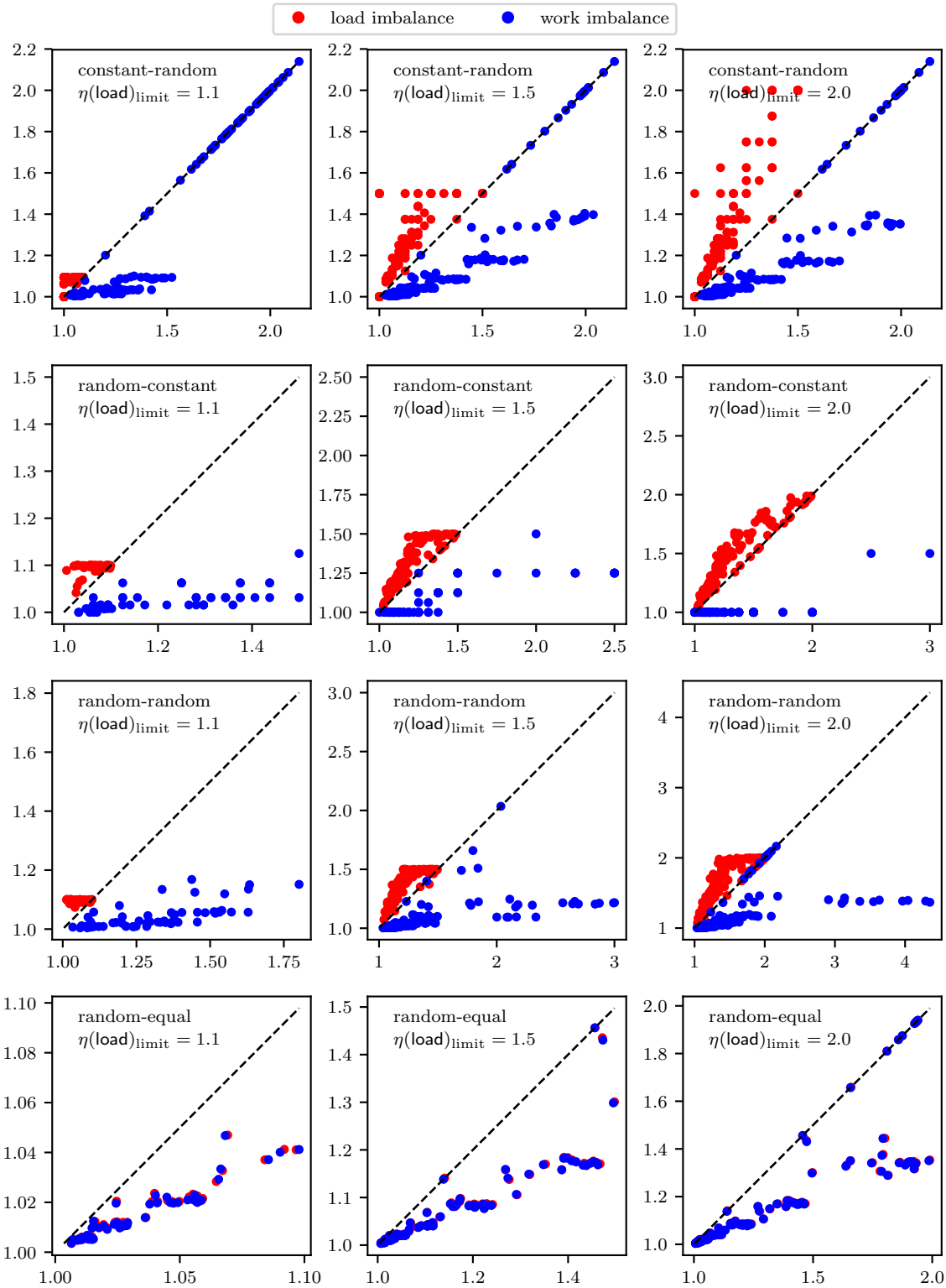

Figure 6. Work and load imbalance obtained by the split routines for several test cases. Chaging the imbalance limit for load, the number of leaves $|L|$, number of splits $|\mathcal{S}|$, and using different generating functions of load and work per leaf. The x-axis is the imbalance obtained by `master` and the y-axis corresponds to the `dev` version.

To address the other issue regarding the robustness of the split, fig. 7 shows a survey of `master`'s split success rate relative to the optimal solution as a function of the load imbalance limit η_l—in fact the success percentage is also absolute, because it was shown in section §3.2 that if the solution exists, the algorithm in listing 4 will find it. The sample space was the same as that used for fig. 6.

Finally, concerning the time to solution of the optimal domain split's implementation, some test cases were prepared with different values of $|\mathcal{S}|$ up to 10^5 and $|L| = 100\times|\mathcal{S}|$ (see fig. 8). It can be seen that the asymptotic complexity is quasilinear for both split implementations—recall that the optimal domain split on `dev` has a theoretical complexity of $\mathcal{O}\big(|L| + p\mathcal{M}\log(|L|)\log(p\mathcal{M}/\epsilon)\big)$ and `master`'s greedy split is simply an array scan $\mathcal{O}(|L|)$. It should be noted that for $|L| \approx 10^6$, which is already a very extreme situation possibly common for exa-scale top-trees, the time to solution of the optimal domain split is around 40 milliseconds, and yet for $|L| \approx 10^7$ the solution is reached in less than a second.

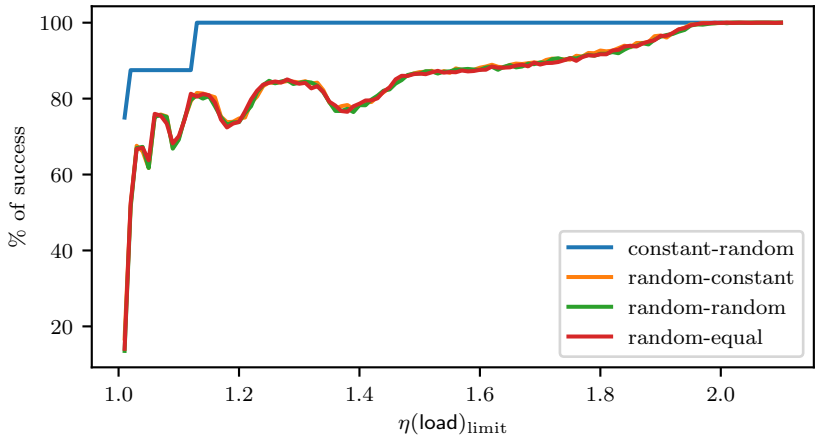

Figure 7. PGADGET-3's split success, relative to the optimal algorithm.

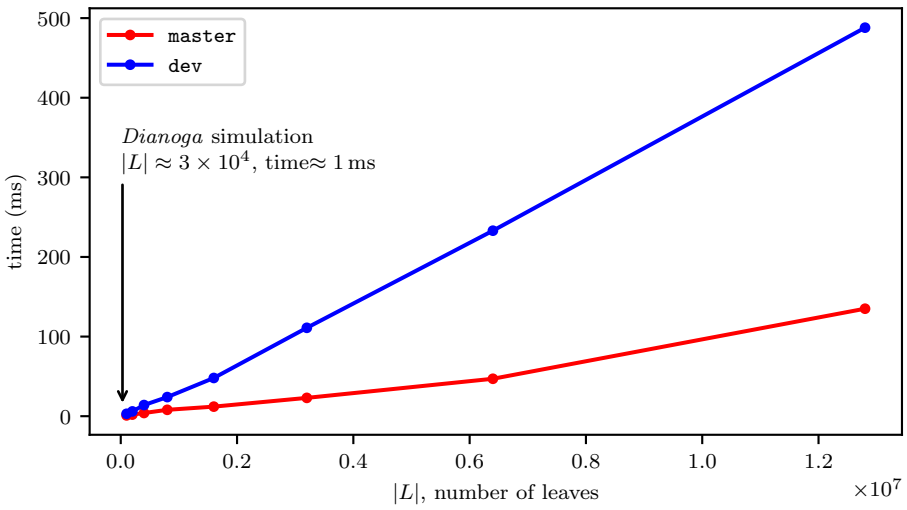

Figure 8. Split routines' time to solution. A *Dianoga* simulation with 2×10^8 particles running on 240 processes would produce a top-tree with no more than 30'000 leaves, in that case both algorithms reach the solution in few milliseconds.

§2. UNIT TEST: ASSIGN

Assigning domains to MPI processes was shown in section §3.3 to be resolved with the choice of a method depending on what it is needed, whether a fast domain decomposition or a smaller work imbalance. One of those methods was simply recovering the assignment for the previous domain decomposition, the assignment thus produced was denoted $\mathcal{A}_{\text{fast}}$. The other used a greedy and almost optimal algorithm to reduce the work imbalance, and the resulting assignment was denoted $\mathcal{A}_{\text{good}}$. Since, the construction of $\mathcal{A}_{\text{fast}}$ is trivial, this section concentrates solely on the implementation on the branch `dev`of the algorithm that computes $\mathcal{A}_{\text{good}}$ and compares it to its counterpart on the branch `master`—which was highlighted on one of the issues (see §2.3) because of its complexity $\mathcal{O}(p^2\mathcal{M})$.

Similar to the previous section for the study of the split algorithms, random input datasets were generated combining constant and random numbers for the load and work functions. Other test parameters were the number of processes $p = |\mathcal{X}|$, ranging from 100 to 2×10^5, and the number of multiple-domains $\mathcal{M}$ ranging from 1 to 16. Figure 9 shows the results of those tests. Again, each dot is a pair of imbalance values for a single realization, the x-coordinate is the imbalance obtained by the routine in `master` and the y-coordinate is the imbalance for the routine in the `dev` branch. As it might have been expected, `dev`'s assignment does not deal well with load imbalance, since it was not built to minimize that. While the work imbalance is very often equally matched to the `master`'s. In a few cases of the type "random-random" the `dev`'s routine finds a better work imbalance than the `master`'s.

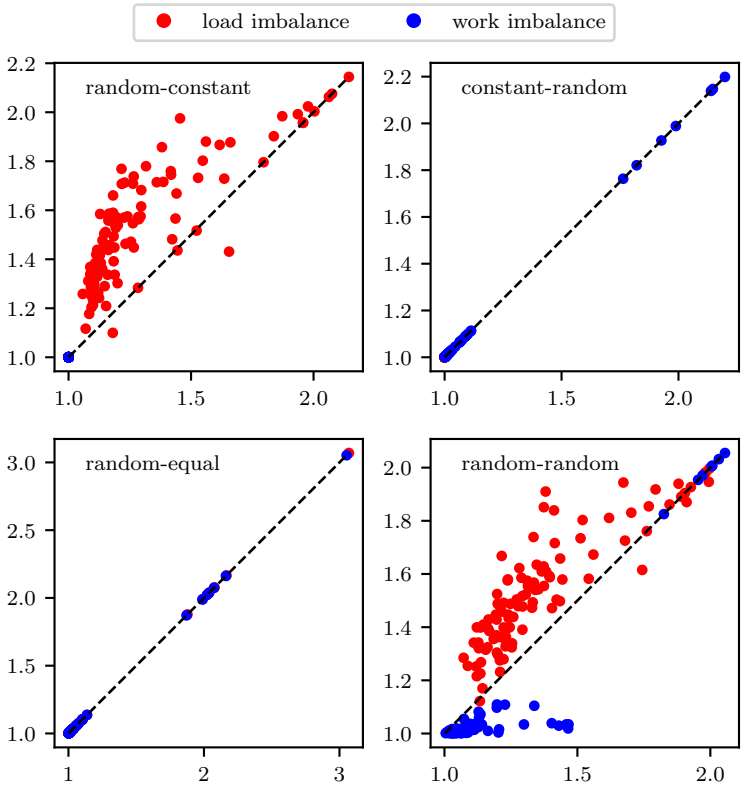

Figure 9. Work and load imbalance obtained by the assign routines for several test cases. Chaging the number of leaves $|L|$, number of splits $|\mathcal{S}|$, and using different generating functions of load and work per leaf. The x-axis is the imbalance obtained by `master` and the y-axis corresponds to the `dev` version.

Figure 10, shows the time to solution compared for both implementations for different values of p, at $\mathcal{M} = 4$. The quadratic dependence on p for the `master` version is immediately recognized, while `dev`'s seems to be almost linear—it is in fact proportional to $p \log p$, as it was remarked in §3.3. Clearly the most urgent disadvantage of the `master`'s assignment is its algorithmic complexity, since for $p = 10^5$ the routine takes over 6 minutes to arrive to the solution, while `dev`'s implementation does it in just 100 milliseconds.

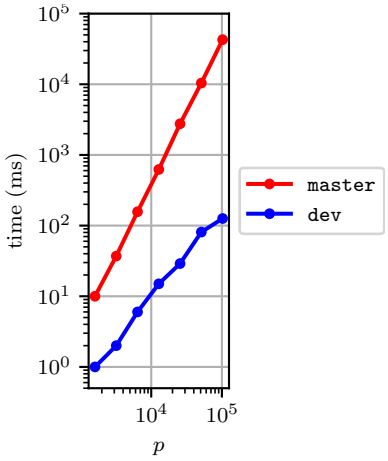

Figure 10. Assign routines' time to solution, with multiple-domains $\mathcal{M} = 4$. As a result of these algorithms $p\mathcal{M}$ domains are assigned to p processes.

§3. PERFORMANCE

One of the issues discussed in Chapter 2 concerned the scalability of GADGET's domain decomposition. It was remarked that for the same global data, the time spent in the routine `exchange`—in the `master` branch—grows linearly with the number of processors, taking over ¾ of the total execution time (see §2.3). The strategy adopted in this thesis and implemented in the developing branch `dev` (see Chapter 3) would improve the speed of the `exchange` by building a domain assignment $\mathcal{A}_{\text{fast}}$ for which the number of exchange communications is expected to be reduced.

It is worth to remark that on every execution of GADGET (see Listing 1) there will be two types of domain decompositions: the one that takes place right after the initial conditions are read from a file (and before any time-step evolution) and those performed between time-steps. The main difference between them is the way particles are distributed among MPI processes. In the first case any particle can be hosted by any processes because they have been just read from an initial condition file by the module `read_ic`, that decides which processes should read which portion of that file with no regard whatsoever on the particles spatial coordinates. On the other hand, the *second domain decomposition* will deal with a state in which particles could be slightly off their domains and the procedure is more like a readjustment of the domain decomposition. Being the latter the most common situation by far, the tests presented in this section will concern only the second domain decomposition.

Figures 11 and 12 show a comparison between the `master` and `dev` branches' full domain decompositions according to their time to solution. The data consist of a set of $|P| = 2 \times 10^8$ particles extracted from a cosmological simulation code-named *Dianoga-g1212639-10x*. And the tests were run on 4 computing nodes (192 cores and MPI processes). On figure 11, for each test case a fraction of the particles were shuffled from their original processes to a fraction of other neighboring processes. While in figure 12, all particles were perturbed from their initial coordinates in space by adding a gaussian noise with zero mean and variance σ ranging from 10^{-10} to 1 times the length of the simulation box l_{box}. From figs. 11 and 12 it can be seen that most part of the domain decomposition is spent in the `exchange` routine. Furthermore, either by shuffling or translating the particles by different amounts, the `master` branch will always perform a costly domain decomposition involving the exchange of almost all particles. While the `dev` branch, preserving a similar domain decomposition structure through the assignment $\mathcal{A}_{\text{fast}}$, is able to reduce the number of particles exchanged and the number of communications with other processes, thus gaining in speed.

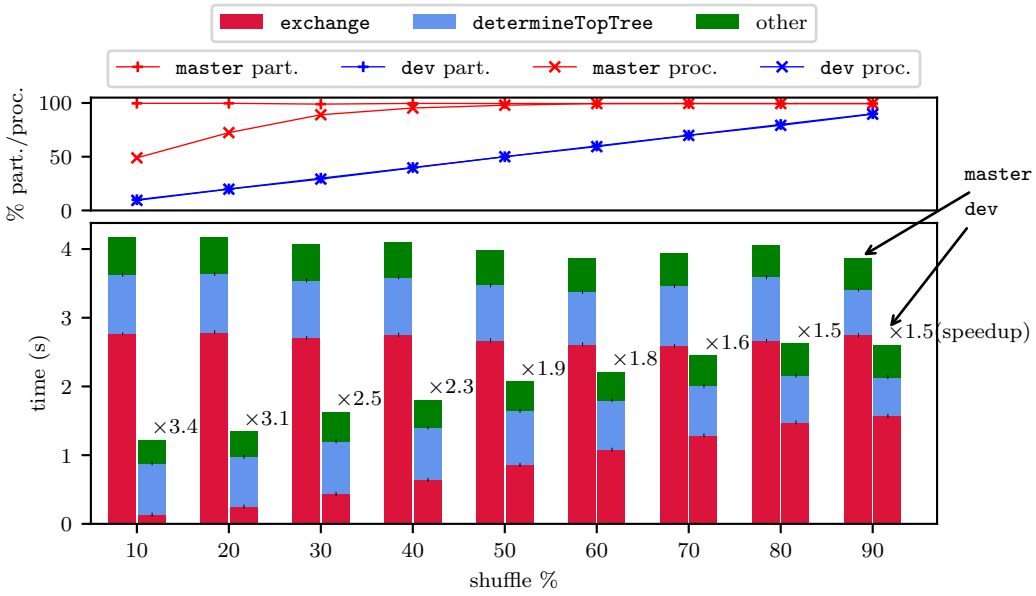

Figure 11. The effect on the domain decomposition of shuffling particles across MPI processes. The x-axis represents the percentage of particle and processes participating in the shuffling. The bar plots come in pairs: the `master` branch to the left and `dev` branch to the right, for each value of shuffle. The speedup of `dev` with respect to `master` is also shown on top of the `dev` bars. The graph on top shows the percentage of particles exchanged and processes communications. These instances were run on $4 \times 48 = 192$ processors.

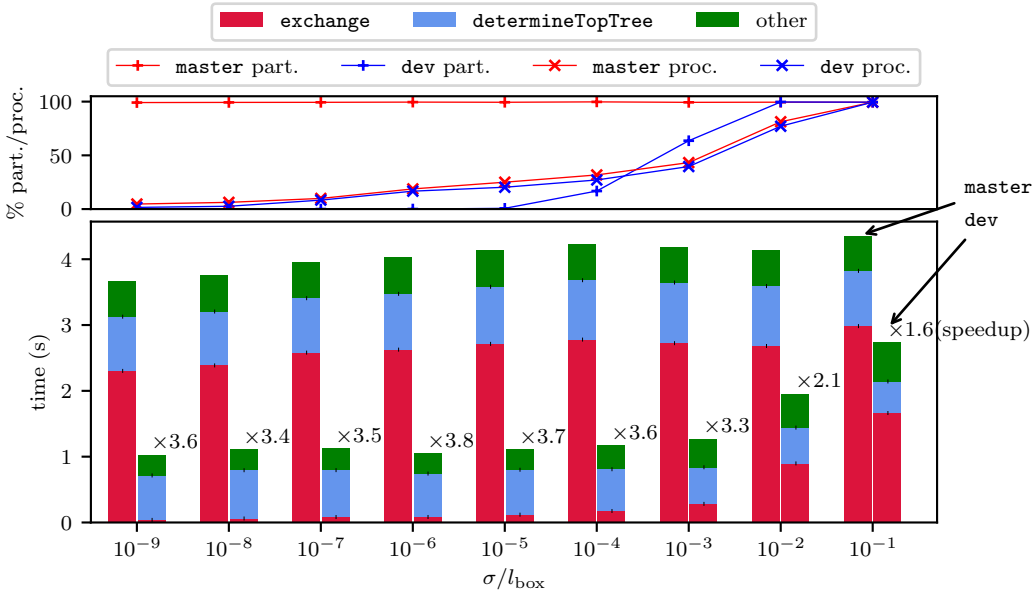

Figure 12. The effect on the domain decomposition of shifting particles' positions. The x-axis represents the value of the variance σ of the gaussian noise used to shift particles, relative to the size of the simulation box l_{box}. The bar plots come in pairs: the `master` branch to the left and `dev` branch to the right, for each value of σ. The speedup of `dev` with respect to `master` is also shown on top of the `dev` bars. The graph on top shows the percentage of particles exchanged and processes communications. These instances were run on $4 \times 48 = 192$ processors.

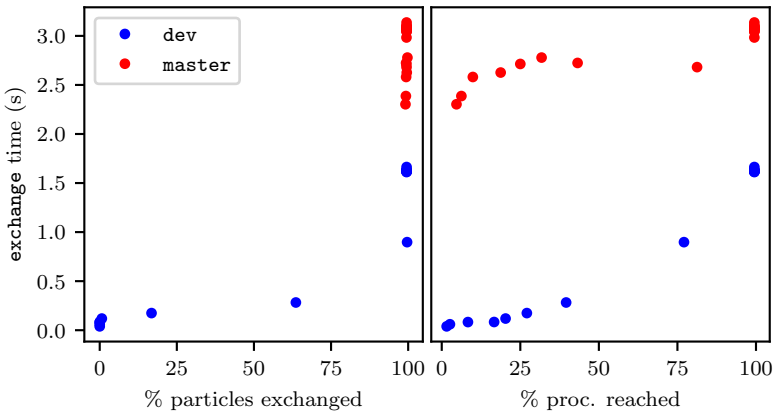

Figure 13. Time of `exchange` dependency on the total number of particles exchanged and the maximum number of communications with other processes from each process.

The relationship between particles exchanged (or number of communications with other processes) and time of `exchange` for the previous test cases is extracted in fig. 13 for a close-up inspection. The time variability of the routine on the `master` branch indicates that the number of processes involved in the communication is an important factor for the performance. Notice that all red dots correspond to almost 100% particles being exchanged, however the time varies from 2 to 3 seconds (roughly $\times 1.5$ speedup) depending on the number of processes to be reached. The blue dots indicate that there's huge variability on the time if the number of particles to be exported is reduced as well—for instance, when 60% of the particles are exchanged with 40% neighboring processes, the speedup of `exchange` is $\times 5$.

§4. QUALITY OF THE DOMAIN DECOMPOSITION

FASTER DOESN'T ALWAYS MEAN BETTER, and a fast but clumsy domain decomposition could result in a time-step evolution extremely slow due to work imbalance or the end of the execution with an error message if at least one process is loaded with more particles than it can save into memory. That's why imbalance of the work and load functions are the quantities on which any judgement of the quality of the domain decomposition will be based hereafter—see §2.2 for the definitions of work, load and imbalance.

Figures 14 and 15 show a comparison between the `master` and `dev` branches' full domain decompositions according to their work and load imbalances. Again, the data consist of a set of $|P| = 2\times10^8$ particles extracted from the cosmological simulation *Dianoga-g1212639-10x*. And the tests were run on 4 computing nodes (192 cores and MPI processes). The gravitational potential ϕ stored in the initial condition file was used to estimate the computational burden associated to each particle: work $= -\phi$. This choice seems like a reasonable approximation, since for any point $\vec{r}$ in space, $\phi(\vec{r})$ is always negative and it slowly decays to zero as $\vec{r}$ goes far from the gravitational sources.

On figure 14 a bundle of test cases are shown for which a fraction of the particles were shuffled from their original processes to some other neighboring processes—using the notion of locality induced by the MPI ranks, eg. processes with ranks 10 and 11 are close neighbors, as well as 0 and $p-1$. This type of test case is oriented mainly to the stability of the construction of the top-tree. The tree on the `dev` version, is resilient. It is always the same independently on how particles are shuffled among processes, because their physical properties, like the positions in space, are left unchanged. Thus `dev`'s imbalances give straight horizontal lines. On the contrary, the top-tree on the `master` version, is not so stable. It changes when particles are shuffled among processes by different amounts, thus there are fluctuating imbalance curves associated to it.

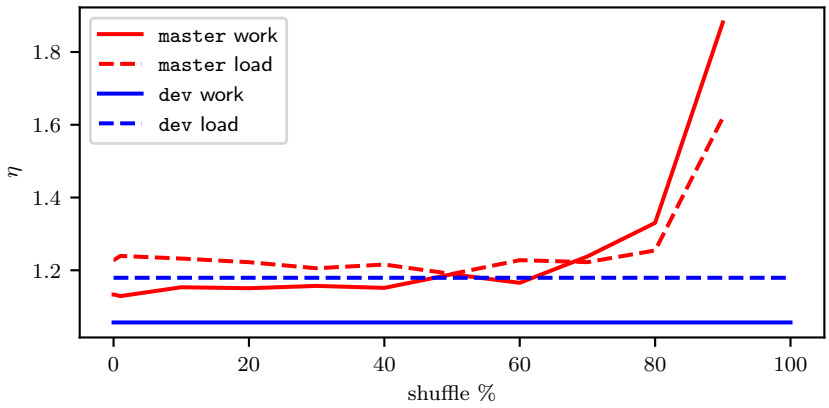

Figure 14. Imbalance of work and load obtained from the domain decomposition, after shuffling particles among processes. Each test run on $4 \times 48 = 192$ processors, and consisted of 2×10^8 particles.

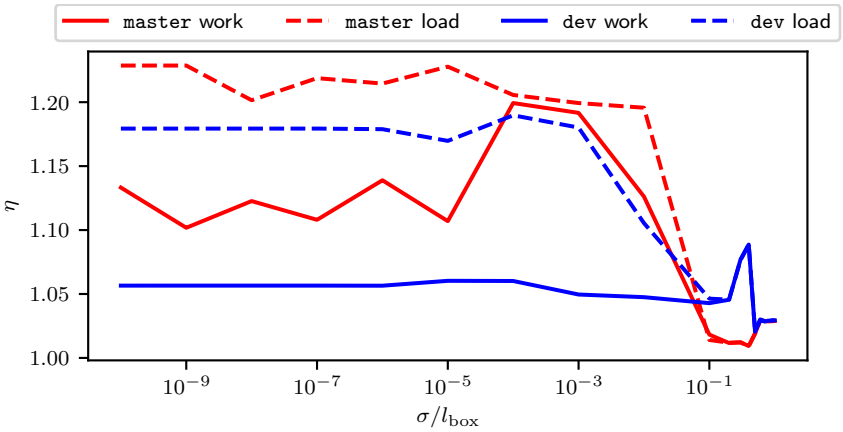

Figure 15. Imbalance of work and load obtained from the domain decomposition, after randomizing particles's positions with a gaussian noise of variance σ and zero mean. Each test run on $4 \times 48 = 192$ processors, and consisted of 2×10^8 particles.

The trend seems to indicate that when more than 80% of the particles are shuffled, the domain decomposition of the `master` branch *breaks*, in the sense that unexpected high values of the work/load imbalance are produced. As a matter of fact, the top-tree constructed by `master`'s implementation is, up to a certain degree, and approximate tree, ie. the work and load on the vertexes are in some situations estimated values. That said, a plausible hypothesis for the divergent imbalances is that the work/load associated to the vertexes are underestimated or overestimated leading to the creation of a set of leaves L which doesn't necessarily satisfy the leaf condition (see Section §3.1), meaning that there is no fine partition fo the regions that might require it. The subsequent steps of the domain decomposition: split and assign, being fed with tainted information will produce dull results.

Figure 15 shows how the quality of the domain decomposion is affected by the displacement of the particles. The test cases were generated by adding to the initial positions of the particles a gaussian noise with zero mean and variance σ ranging from 10^{-10} to 1 times the length of the simulation box l_{box}, in an attempt to recreate a possible situation of particle displacements between domain decompositions due to the dynamics of the system. It can be seen that as the cosmological structures smooth out with the gaussian noise the smaller the imbalance obtained by the `master` branch. It is also worth remarking that the `dev` branch implementation was set to replace $\mathcal{A}_{\text{fast}}$ for $\mathcal{A}_{\text{good}}$ whenever $\eta(\mathsf{work}, \mathcal{A}_{\text{fast}}) \geq 1.10$. As a result, $\eta(\mathsf{work})$ for `dev`'s implementation (blue solid line) does not go above 1.10. The two points where the solid blue and red lines intercept divide the test cases into three regimes:

- in the first region, for $\sigma/l_{\text{box}} < 0.1$, the `dev` branch produces a better domain decomposition than `master`, possibly due to the optimal split;
- for $0.1 < \sigma/l_{\text{box}} < 0.5$ the `master` branch instead gives a better domain decomposition than `dev`, because it uses the assignment of sub-domains to processes to reduce the imbalance, while `dev` is using a not so good, but faster assignment $\mathcal{A}_{\text{fast}}$;
- for $0.5 < \sigma/l_{\text{box}}$ the imbalance $\eta(\mathsf{work}, \mathcal{A}_{\text{fast}})$ exceeds 1.10, triggering in `dev` the use of $\mathcal{A}_{\text{good}}$ and both versions, `master` and `dev`, are able to find equally balanced domain decompositions.

§5. SCALABILITY

Scalability is a property of the code that comes in two flavours, namely *strong* and *weak*. The first indicates how the code speeds-up as the number of processes p grows while keeping a fixed problem size. The second indicates how the code's time to solution changes as p grows but keeping fixed the problem size per processes, ie. the total problem size grows proportional to p. An ideal parallel program will speed-up proportionally to p in the first case, and its execution time will remain constant for the second case. A code that scales very well is not necessarily the fastest. Nevertheless, scalability provides understanding on how well parallelized is the code and how will it behave for bigger problems sizes and increased resources. That information can be useful when asking for computational resources for the developement of a project or when the code goes through performance improvements.

In section §2.3, it was mentioned that the domain decomposition barely scales. A problem which was mainly rooted in the costly communication pattern within the `exchange` routine. Cutting the number of particles exchanged and pairs of processes communications at every domain decomposition—which was already proved to be a key factor for gaining performance, see fig. 12—could improve the scalability of the code. Assuming that during a time step, particles move from their previous positions by a small amount compared to the simulation box size l_{box}, the scalability of the domain decomposition can be tested, in the absence of dynamics, after perturbing the particle's positions with a gaussian noise of zero mean and σ variance. The bigger the value of σ, the farther some particles will travel in space, resulting in a bigger volume of particles exchanged and a bigger number of processes pairs participating in that operation.

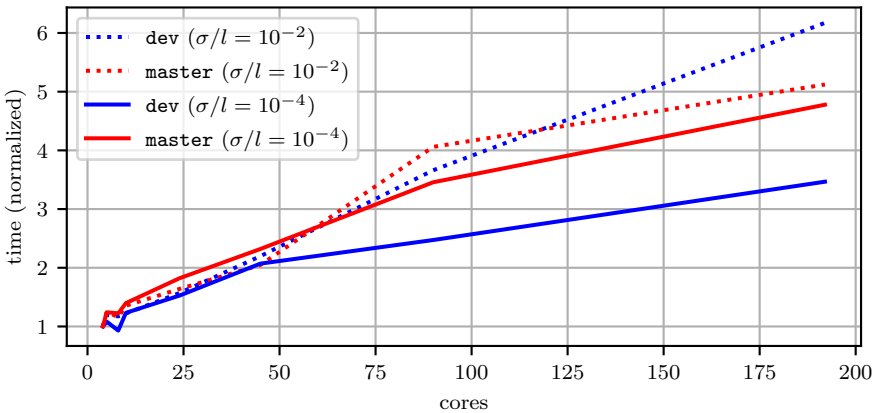

Figure 16. Weak scalability of the domain decomposition. The red color indicates the `master` version, and blue for `dev`. Two cases are represented: $\sigma/l = 10^{-4}$ for which particles are slightly moved out from their initial positions and $\sigma/l = 10^{-2}$ for which their change in positions is comparable to the box size. Different initial condition files where used consisting of 4.8, 6.4, 8.8, 11.7, 27.9, 52.6, 105.2 and 225.1 million particles, and the number of MPI processes was scaled accordingly to keep the ratio of particles per process almost constant.

Figures 16 and 17 show the weak and strong scalability respectively, of the entire domain decomposition of the two branches `dev` and `master`, for two different test cases: $\sigma/l_{\text{box}} = 10^{-2}, 10^{-4}$. The results are coherent with the hypothesis elaborated aboved. Furthermore, at least for small particle displacements, `dev`'s implementation scales slightly better than `master`.

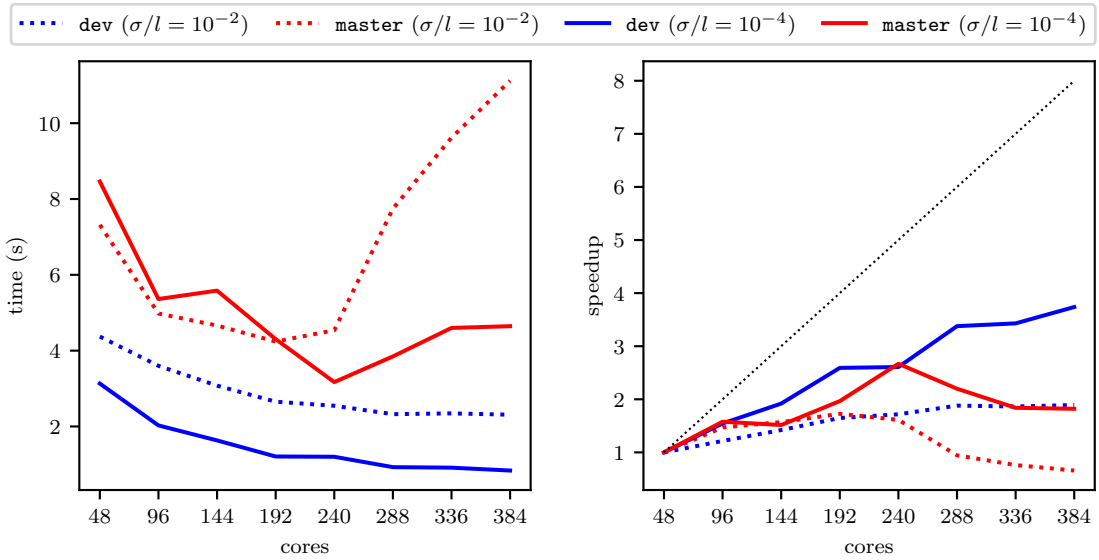

Figure 17. Strong scalability of the domain decomposition. The red color indicates the `master` version, and blue for `dev`. Two cases are represented: $\sigma/l = 10^{-4}$ for which particles are slightly moved out from their initial positions and $\sigma/l = 10^{-2}$ for which their change in positions is comparable to the box size. The initial conditions file with 225.1 million particles is the same for all cases.

§6. THREADS

As the size of the MPI communicators increases, also does the time it takes to perform collective communications—in particular the expensive *all-to-all* type of pattern which is present in the `exchange` routine. That means that simply by reducing the number of MPI processes participating in a simulation, there can be speed-up on some parts of the code, which are communication bound. However, other sections, which are computationally intensive will surely slow down unless the number of processing element remains constant. This latter requirement can be satisfied by implementing a hybrid MPI-OpenMP parallelization of the code.

The profile of the domain decomposition indicates that the most significant computationally intensive section of the code is the construction of the top-tree—see fig. 3, that operation is carried out by the routine `determineTopTree`. That's where the efforts for OpenMP parallelization were concentrated. On top of the `dev` branch a threaded version named `omp` has been created mainly by introducing `#pragma omp` on for loops that are linearly dependent on the number of particles. It is worth mentioning that the relative simplicity of the algorithms of `dev` with respect to the `master` branch, made this work a lot easier. The latter is mostly implemented through recursive functions and data structures which are difficult to parallelize without significant overhead—for example sorting. As a result of this OpenMP parallelization, the function `determineTopTree` on the branch `omp` scales very well with the number of threads. See for instance fig. 20 that shows its strong scalability and fig. 18 that shows the profile of this routine for different versions and number of threads.

Consequently, a full domain decomposition running on threads and less MPI processes, benefits from a speedup in the `exchange` routine due to the reduction of the communicator size—compare the time of `exchange` of `dev` for 12 MPI processes per socket and the same function of `omp` for 1 MPI process per socket, fig. 19—while `determineTopTree`'s time is kept constant due to the good strong scaling of the threaded implementation. A downside of the `omp` version is the fact that the routine `countToGo` is still running serial and it is linearly dependent on the problem size—that's why this function's execution time grows when passing from 12 to 1 MPI processes, and it remains constant as the number of threads varies, see fig. 19.

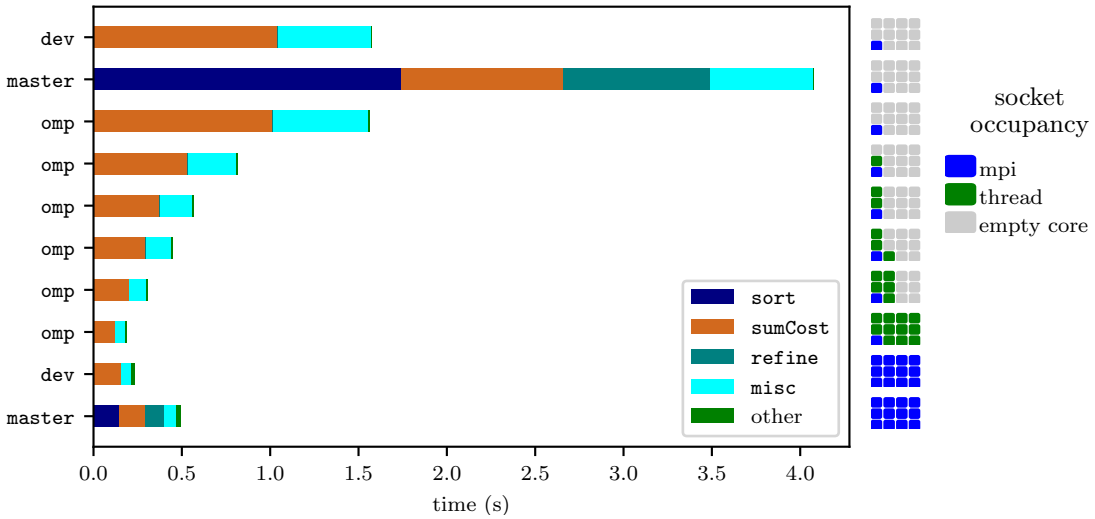

Figure 18. Profile of the top-tree construction for the `master`, `dev`, and `omp` versions, using 1 or 12 MPI processes per socket and different number of threads. These instances were run on 8 computing nodes using an initial condition file consisting of 2×10^8 particles.

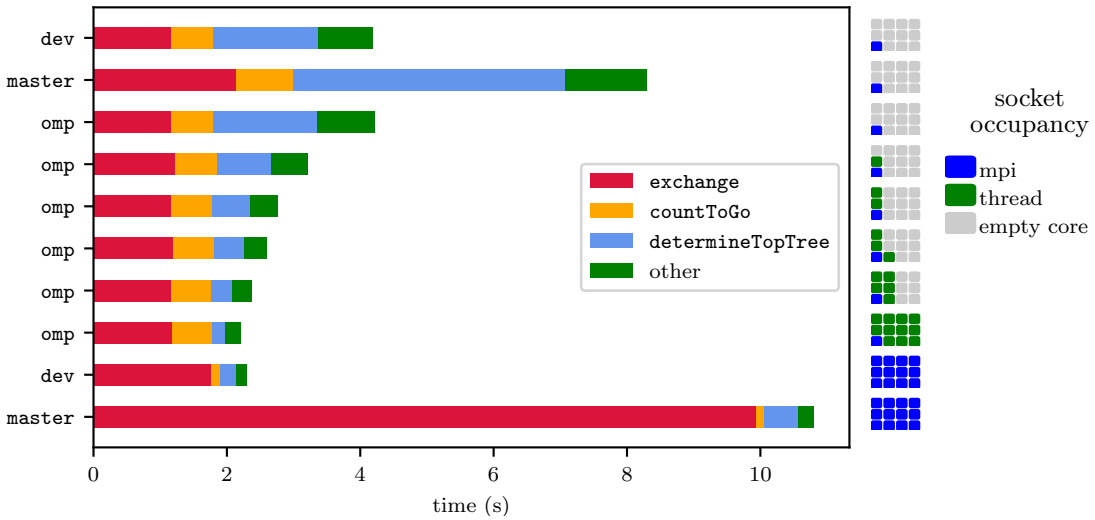

Figure 19. Profile of the domain decomposition for the `master`, `dev`, and `omp` versions, using 1 or 12 MPI processes per socket and different number of threads. These instances were run on 8 computing nodes using an initial condition file consisting of 2×10^8 particles.

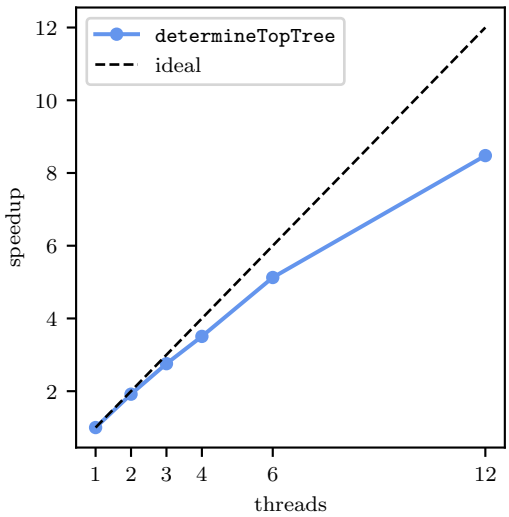

Figure 20. Strong scalability of `determineTopTree` with the number of threads. These instances were run on 8 computing nodes using 4 MPI processes per node for an initial condition file consisting of 2×10^8 particles.

Chapter V

CONCLUSIONS

GADGET CODE FOR ASTROPHYSICAL SIMULATIONS is undergoing a process of re-design for the improvement of its modularity, its capacity to handle challenging problems and to adapt it for the forthcoming exa-scale architectures. The domain decomposition is one of the components of its core, that is most relevant for this effort. This thesis has examinated the individual constituents of the domain decomposition module, their purpose and performance; and it has proposed new robust and efficient algorithms to carry out the tasks they ought to accomplish.

The stability of the construction of the top-tree was one of the key issues that were addressed. Mainly because it has an impact on the quality of the domain decomposition in terms of imbalance. The proposed algorithm `iterative_toptree` (see Listing 3) is able to construct a top-tree which is uniquely determined by the particles' configuration, unlike the original procedure present in PGADGET-3's code. In addition it is slightly faster.

The algorithm `bs_split` (see Listing 5) for the computation of balanced split of Hilbert-key ranges was also presented. Having the advantage over the original greedy version for being perfectly optimal and robust. Which could mean that `bs_split` is able to find a solution to the split problem with memory load constraints whenever the solution exists—a feature not present in the original code—and the result is guaranteed to be optimal. Constraints of additive positive quantities are naturally handled by this algorithm, which means additional requirements can be easily added in the future if necessary; for instance a limit on the number of particle of a specific type per MPI process. The computational cost of `bs_split` was shown to be comparable to the original split algorithm, and negligible for the problem sizes dealt with by GADGET.

Unlike the split, the optimal assignment of domains to MPI processes is a hard computational problem. The proposed algorithm, `greedy_assign` (see Listing 6), is a reasonable alternative to the exact solution. With an algorithmic complexity of $\mathcal{O}(p \log p)$ it becomes a strong candidate for the substitution of PGADGET-3's assign routine, which grows as $\mathcal{O}(p^2)$ where p is the number of MPI processes. In addition, the use of a robust and efficient top-tree and split algorithms could provide a lower bound to the work imbalance of the assignment that if small enough (determined at runtime) it would grant the possibility to perform an assignment of domains to processes to reduce the number of particles exchanges, and thus the communication time, without negative consequences for the work imbalance.

Finally, the whole domain decomposition module was modified for thread parallelism using the OpenMP standard. The routine for the construction of the top-tree proposed in the thesis, which is the second most time consuming component of the module after `exchange`, turned out to be easily parallalelized. As a result, there is almost perfect scalability of the latter with the number of threads, and the execution time of the hybrid MPI-OpenMP full domain decomposition is roughly the same as for the pure-MPI version using the same amount of computational resources.

Future work on GADGET's core should tackle the yet unsolved problem of the modularization. An interesting idea on this regard that should be explored is the `C++` implementation through templated classes of the data structures, separating the fundamental quantities used by the code's infrastructure and any new special physical quantities used by individual physical modules; allowing a developer to interact with the code and the "services" offered by its infrastructure—ie. domain decomposition, tree walking, neighbours finding, communications—without having to modify that infrastructure himself.

Also it should be noted that the version of the domain decomposition as a result of this thesis, relies on MPI's *all-to-all* routines to perform the `exchange` of particles. This gives an upper hand in performance with respect to the PGADGET-3 implementation. But it is not safe to use in a simulation constituted of more than 2×10^9 particles, because MPI routines use 32-bit signed integer counters in its interface. This shortcoming should be resolved before the code is integrated into PGADGET.

Almost blank page.

Appendix A

ALPHABETIC INDEX

Almost blank page.

Appendix B

BIBLIOGRAPHY

[Garey, 1979] *Michael R. Garey, David S. Johnson. Computers and Intractability - A Guide to the Theory of NP-Completeness.* Bell Telephone Laboratories 1979. ISBN 0-7167-1044-7.

[Graham, 1969] *R. L. Graham. Bounds on Multiprocessing timing anomalies.* SIAM J. Appl. Math. Vol. 17, No. 2, March 1969. DOI 10.1137/0117039.

[Haverkort, 2011] *Haverkort H. An inventory of three-dimensional Hilbert space-filling curves.* arXiv:1109.2323v2.

[MPI] *MPI: A Message-Passing Interface Standard, version 3.0.* September 21, 2012. https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[OpenMP] *OpenMP, Application Programming Interface, version 4.5.* November 2015. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[Ravindra, 1993] *Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin. Network Flows - Theory, Algorithms, and Applications.* Prentice-Hall 1993. ISBN 0-13-617549-X

[Springel V. 2005] *Springel V. The Cosmological simulation code GADGET-2.* MNRAS, Vol. 364, Issue 4, 2005, 1105–134.

[Springel. Millenium] *Springel V. Simulating the joint evolution of quasasrs, galaxies and their large-scale distribution.* arXiv:astro-ph/0504097.

[TEXbook] *Donald E. Knuth. The TEXbook.* Addison–Wesley publishing, ISBN 0-201-13447-0.

[TopCoder] *TopCoder - Competitive Programming Tutorials.* https://topcoder.com/community/competitive-programming/tutorials

Almost blank page.

Appendix C

CODE SNIPPETS

C++ implementation of the algorithm presented on Listing 6, to construct the assignment $\mathcal{A}_{\text{good}}$ that minimizes approximately the work imbalance.

```
1.  extern "C"
2.  double domain_assign_minWorkBalance(
3.      /*input*/ const int nproc,const int Ndom,
4.                const double *work,
5.      /*output*/ int *assign)
6.  {
7.      vector<int> proc_capacity(nproc,Ndom/nproc);
8.      vector< pair<double,int> > work_vector;
9.      set< pair<double,int>  > proc_set;
10.     double sumwork=0,maxwork=0;
11.     for(int i=0;i<Ndom;++i)
12.     {
13.         work_vector.push_back( {work[i],i} );
14.         sumwork+=work[i];
15.     }
16.     sort(work_vector.rbegin(),work_vector.rend());
17.     for(int i=0;i<nproc;++i)
18.         proc_set.insert({0,i});
19.     for(int i=0;i<Ndom;++i)
20.     {
21.         auto proc = *proc_set.begin();
22.         proc_set.erase(proc_set.begin());
23.         proc.first += work_vector[i].first; // assignment
24.         assign[work_vector[i].second] = proc.second;
25.         --proc_capacity[proc.second];
26.         maxwork=max(maxwork,proc.first);
27.         if(proc_capacity[proc.second]>0)
28.             proc_set.insert(proc);
29.     }
30.     assert(proc_set.size()==0);
31.     return nproc*maxwork/sumwork;
32. }
```

Almost blank page.

Appendix D

DEMONSTRATA

LEMMA D.1. In the top-tree, a DFS traversal of the subtree starting at vertex with keys $[a, b)$ will find the first leaf has keys $[a, b')$ and the last leaf has keys $[a', b)$ where $a', b' \in [a, b)$.

Proof. If the vertex v with keys $[a, b)$ is a leaf then the Lemma's assertion is true. Otherwise, the first leaf is the first leaf in the subtree starting at the first child of v with keys $[a, a + l_{ab}/8)$, and the last leaf is the last leaf in the subtree starting at the last child of v with keys $[b - l_{ab}/8, b)$ where $l_{ab} = b - a$. By induction one deduces that the first leaf is found at a vertex with keys $[a, a + l_{ab}/8^{n_1})$ and the last leaf has keys $[b - l_{ab}/8^{n_2}, b)$, where n_1 and n_2 are integer numbers. □

Proof of Proposition 2.1: It is not difficult to see why each Hilbert-key is mapped into exactly one leaf of the top-tree. For each vertex v with keys $[a, b)$ and a key $k \in v$ either v is a leaf or there exists a unique child $v' \subset v$ such that $k \in v'$. Then if one starts at the root one clearly has $k \in [0, 2^{3K})$ and then by selecting always the child that contains k one eventually arrives to some leaf that contains k. On the other hand, two different leaves u and u' cannot share a common key; if they had then their lowest common ancestor $\mathrm{LCA}_{uu'}$ will have two different children v and v', such that $u \subset v$ and $u' \subset v'$, but $u \cap u' \neq \emptyset$ implies $v \cap v' \neq \emptyset$ while v and v' for being different children of $\mathrm{LCA}_{uu'}$ they satisfy $v \cap v' = \emptyset$ which is a contradiction. Therefore, for each key k there exists a unique leaf u in the toptree such that $k \in u$.
For the second part of the proposition, one considers two different leaves $u = [a, b)$ and $u' = [a', b')$ with DFS indexes i and $i+1$. Their lowest common ancestor $\mathrm{LCA}_{uu'}$ has two consecutive children v and v' such that $u \subset v$ and $u' \subset v'$. Then u is the last leaf in the subtree of v because the following leaf in the DFS order $u' \notin v$, and similarly u' is the first leaf in the subtree of v'. Therefore $v = [., b)$ and $v' = [a', .)$ due to Lemma D.1. But $v = [., b)$ and $v' = [a', .)$ for being consecutive children of $\mathrm{LCA}_{uu'}$ must have $b = a'$. □

Proof of Listing 4. By construction if $\mathcal{S}_o$ is a solution found by the the previous algorithm, then for every $s \in \mathcal{S}_o$ we have

$$\mathsf{work}(s) \leq w_{\max};$$

thus

$$\sup_{s \in \mathcal{S}_o} \mathsf{work}(s) \leq w_{\max} = \frac{\eta_w}{n} \sum_{d \in L} \mathsf{work}(d);$$

therefore

$$\eta(\mathsf{work}, \mathcal{S}_o) = \frac{\sup_{s \in \mathcal{S}_o} \mathsf{work}(s)}{\frac{1}{n} \sum_{d \in L} \mathsf{work}(d)} \leq \eta_w;$$

and similarly it can be shown that $\eta(\mathsf{load}, \mathcal{S}_o) \leq \eta_l$. Therefore, the prescribed constraints are satisfied.
On the other hand if at least one solution $\mathcal{S}'$ exists we need to prove that the algorithm is able to find any. For that we use induction. If $n = 0$ and $L_0 = \emptyset$, a trivial solution exists $\mathcal{S}_o(0, L_0) = \emptyset$ and the algorithm returns precisely that. If $n = 1$ a necessary and sufficient condition for the solution to exist is $\mathsf{work}(L_1) \leq w_{\max}$ and $\mathsf{load}(L_1) \leq w_{\max}$, and the longest prefix that satisfies the constraints is L_1 itself; again the algorithm returns $\mathcal{S}_o(1, L_1) = \{L_1\}$, which is the correct answer. Assume that for a given value of n the algorithm always finds a solution for any L_n namely $\mathcal{S}_o(n, L_n)$, provided the solution exists. If the algorithm is requested to find a split of size $n+1$ for a given leaf sequence L_{n+1}, in the first iteration it will strip the longest prefix s_{n+1} that satisfies the constraints, and for $L_n = L_{n+1} \setminus s_{n+1}$ there must exists a split of size n, because s_{n+1} cannot be larger without violating the constraints. By the initial assumption the algorithm will compute $\mathcal{S}_o(n, L_n)$ and return $\mathcal{S}_o(n+1, L_{n+1}) = \{s_{n+1}\} \cup \mathcal{S}_o(n, L_n)$. □

Proof of Lemma 2.1: Let $\mathcal{S}$ be a domain split, and denote $\eta_{\mathcal{S}}$ be the imbalance of some non-negative function $X \colon \mathcal{S} \to \mathbb{R}^+$, for instance work or load. Then

$$\eta_{\mathcal{S}} = \frac{\sup_{s \in \mathcal{S}} X(s)}{\frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} X(s)};$$

also for any assignment $\mathcal{A} \colon \mathcal{S} \to \mathcal{X}$, let $\eta_{\mathcal{A}}$ denote the imbalance relative to the function X:

$$\sum_{i \in \mathcal{X}} X(i) = \sum_{s \in \mathcal{S}} X(s),$$

and $|\mathcal{X}|\mathcal{M} = |\mathcal{S}|$. On the other hand for any $i \in \mathcal{X}$:

$$X(i) = \sum_{s \in \mathcal{A}^{-1}(i)} X(s) \leq \sum_{s \in \mathcal{A}^{-1}(i)} \sup_{s' \in \mathcal{S}} X(s') = \mathcal{M} \sup_{s \in \mathcal{S}} X(s);$$

therefore

$$\sup_{i \in \mathcal{X}} X(i) \leq \mathcal{M} \sup_{s \in \mathcal{S}} X(s);$$

and it follows that

$$\eta_{\mathcal{A}} \frac{1}{|\mathcal{X}|} \sum_{i \in \mathcal{X}} X(i) \leq \eta_{\mathcal{S}} \frac{\mathcal{M}}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} X(s);$$

hence

$$\eta_{\mathcal{A}} \leq \eta_{\mathcal{S}}.$$

□