Master in High Performance Computing

Enhancing modularity of simulation software through modern C++ features on the example of GADGET

Supervisors:
Luca Tornatore,
Alberto Sartori

Candidate:
Irina Davydenkova

6th edition
2019–2020

Abstract

This work discusses the challenges faced by the cosmological simulation code GADGET under the prospect of future development. A new data structures engine based on C++17 standard is proposed to allow better code modularity leading to significant simplification in developing new features or redesigning the existing ones. The major part of the new construct is based on compile-time programming and thus no run-time penalties are incurred by employing high-level language features.

Contents

Acknowledgements

I would like to express my gratitude to my supervisors - Luca Tornatore for introducing me to GADGET and guiding me through the course of thesis, and Alberto Sartori for invaluable support and ideas in the technical C++ part of the project. My sincere thanks also go to the referee, Claudio Gheller.

1 Introduction

Numerical simulations have become pervasive in modern science production and their role is projected to only increase in the future. Simulations are especially important in fields where actual experiments cannot easily (or at all) be conducted - climate change models, astrophysics and cosmology, nuclear fusion, etc. As the computational power of the machines increases, leading to exascale computing in the near future, current scientific software needs to be able to adapt both to researching finer physical details and to possibly modifying how it works with memory resources to be able to actually use the increased computational power. In this thesis we will work with one of the most widely used cosmological codes -GADGET [1].

GADGET (GAlaxies with Dark matter and Gas intEracT) is a scientific code developed to solve the gravitational and hydro-dynamical equations in their Lagrangian form for a large ensemble of particles. The code can simulate either the formation and evolution of the cosmic structures in the whole Universe or a limited portion of it, usually a gravitationally-bound object (a cluster of galaxies, a single galaxy, a cluster of stars). In the first case, a significant fraction of the visible Universe is evolved in a cosmological background, starting from initial conditions that reproduce the status of the Universe at very early times. These simulations compute the cosmic evolution from very large scales (i.e. a significant fraction of the whole Universe) down to the single galaxies, spanning a time scale equivalent to the entire lifetime of the Universe. In the second case, the code can simulate the evolution of isolated objects at higher resolution, possibly in an evolving cosmological context.

The fundamental physical interaction that drives the evolution of the Universe and its structures is Gravitation that acts on long-distance ranges and as such couples each particle to each other one in the whole computational domain, involving both the ordinary matter and the dark matter. The ordinary matter properties, which are our observables, are instead mostly shaped at the local scale by Hydrodynamics. Several other physical phenomena (star formation, stellar feedback, gas accretion on compact objects, radiation, magnetic fields, and others), whose effects extend to the galactic scale, are also determining the fundamental traits of ordinary matter and must be modelled appropriately.

GADGET organises data and algorithms upon the central idea of an oct-tree that offers the support for both the domain decomposition and the neighbours' search. The gravitational forces are calculated by a mean-field approximation using FFTs for large scales, while they are computed pairwise for very close particles or using a multipole expansion of the mass distribution for more distant particles. The hydrodynamical forces and all quantities related to local physics are solved using the Smoothed Particle Hydrodynamics technique which averages physical quantities over the neighbouring particles, weighting them by a smoothing kernel with suitable characteristics.

As we have already stated it is important for a scientific code to be easily modifiable and this is where GADGET currently comes short. The details of the tree data structure are deeply intertwined in all parts of the code that use the tree for their calculations. Whenever one needs to introduce a new physical module which requires the tree for its operations, one needs to modify the core parts of the code - add some new variables inside fundamental data structures, modify domain decomposition and tree routines. This leads to a complicated and barely human-readable "`#ifdef` forest" (see App. A) which even the experienced developers have trouble navigating through. Any new developer starting to work on GADGET currently needs to spend weeks learning in thorough detail about the core of the code and then add

modifications to the core, potentially crushing something unrelated in the process and then spending more time on untrivial debugging procedure. The ideal situation, however, should not require every developer to ever touch core functionality and instead let them concentrate on their particular physics problem, implementing new functions in a separate file while following clear instructions on the APIs.

We can now state the main goal of this work - *we aim to develop a proof-of-concept framework to make the modularization straightforward and provide the future developers a convenient set of tools for adding new features to* GADGET. Note that, while we only work with the GADGET code, the engine itself can be used in any new scientific software (or an old one being redesigned) aiming for future flexibility.

Another goal of the thesis is tackling the "fear" of C++ among scientific code developers used to low-level languages such as C and Fortran. Currently GADGET is written in pure C and is a very well optimized code. There have been concerns about the possible performance decrease in case of switching to a more sophisticated high-level language like C++. With this work, we try to present a C++ version which does not lose (much) the performance while providing all the advantages of the high-level language. We claim that most of the complicated additions needed to realize our version do not come with runtime penalty, as we use templates and compile time loops/conditions thus resolving everything "complicated" at compile time. Such an approach does lead to an increase of the compilation time, but this increase is not relevant as it stays at the order of a few seconds. On the contrary, there expected to be a performance increase due to giving up huge structs whose members are never accessed all together and grouping up the relevant variables thus better utilizing the cache.

It should be mentioned that we didn't work on the real GADGET production code, but rather the "kernel extract", a mini application that performs the core operations of domain decomposition, tree building and neighbour search. Throughout the text we are going to refer to the GADGET kernel extract and the real GADGET interchangeably, stressing the difference only when relevant to the context.

The project consists of the `main.cpp` file and a number of header `*.hpp` files, which is justified by the extensive use of templates. We have been using the Meson build system in combination with the Ninja to compile the code.

The thesis is organized as follows: in Section 2 we describe the data structures used in the current GADGET code and explain the changes we made in the C++ version to increase the code flexibility and modularity. We also give a number of practical examples to illustrate the use of the new engine. In Section 3 we present the core GADGET algorithm while giving comments on what changes have been made to accommodate for the new data structures used. And lastly, in Section 5 we discuss the overall results we have achieved by this thesis and the future work needed to properly incorporate the developed engine into the real GADGET code.

2 The Data Structures as an engine

2.1 The GADGET data structures

The fundamental data structures in GADGET are the following:

1. **Domain-decomposition** data, related to which MPI task a particle belongs to (or, more precisely, a tree node belongs to)
2. **Tree data**, related to the building of the tree structure and its quick traversal
3. **Particle data**, used for storing the physical properties of the particles and their evolution along the simulation. These data are *strongly* dependent on what physics is actually activated in a given code realization, i.e. at compile-time (for instance: a Gravity-only run, a Gravity+Hydro run, a Gravity+Hydro+Cooling+StarFormation run, a run with also Black Holes, Magnetic Field and Cosmic Rays, etc).

In this work we focus on the 3rd data type, or, in other words, on how the particle-related data are represented in the code. Although it may sound as a 2nd-order detail, it is actually a crucial point for both the readability/maintenance/evolution of the code and its performance at run-time.

GADGET has 6 types of particles: type 0 (gas particles), type 1-3 (dark matter particles), type 4 (star particles) and type 5 (black holes particles). Each type of particle represents a different type of matter (baryons, type 0 and 4; the unknown dark-matter, type 1-3; the singularities, type 5) and is then subject to different physical processes, leaving aside the gravity by which all the particles are affected.

A "physical process" in a simulation translates to a computational algorithm that involves some variables in a set of computations. Therefore, for a particle type to be involved in a given physical process it is required that the particles of that type contain the relevant variables to work with and store the appropriate physical quantities.

What we called a "particle type" is, in fact, a `C` structure whose fields are involved in different computations and keep the results for the subsequent uses. From which it obviously follows that:

- the structures that define each type have fields different than the other structures for the other types;
- the exact composition of a type's data structure depends on the physical processes it will be involved in during the simulation, although the kernel of variables related to Gravity and the time-integration scheme is common to all the particle types.

All in all, the following `C` structures are defined:

`P`	the structure common to all the particles; typical size is 150-200 B.
`SphP`	the structure specific for gas particles; typical size is 250-300 B.
`MetP`	the structure specific for star particles; typical size is 100-120 B.
`BHP`	the structure specific for Black Holes particles; typical size is 200 B.

The `P` structure harbours several variables that are mandatory for each particle: namely, among the others, a unique ID, the type, the position, the velocity, the mass, the gravitational

acceleration, the gravitational potential, the time-step. *Every* particle has an entry in the `P` array and then a separated entry in the relevant one among the others. An expert reader would immediately spot some critical issues with this setup.

First, the structures' large sizes can easily lead to a catastrophic usage of the cache; in other words, the ubiquitous tension between the AoS (Array of Structures) and the SoA (Structure of Arrays) strategy is still relevant also in our case. Some combined usages of the variables are quite common (for instance: the mass, the position, the velocity and the acceleration) while others are rarer. The optimal grouping of the variables inside the structures is, on one hand, almost impossible, and, on the other hand, such a manual fine-tuning would be both impractical in the long term and unstable under code development. Conversely, breaking down the structures into too many smaller substructures would lead to a large amount of cross-indexing between the fundamental `P` and all the other relevant ones (see Fig. 1).

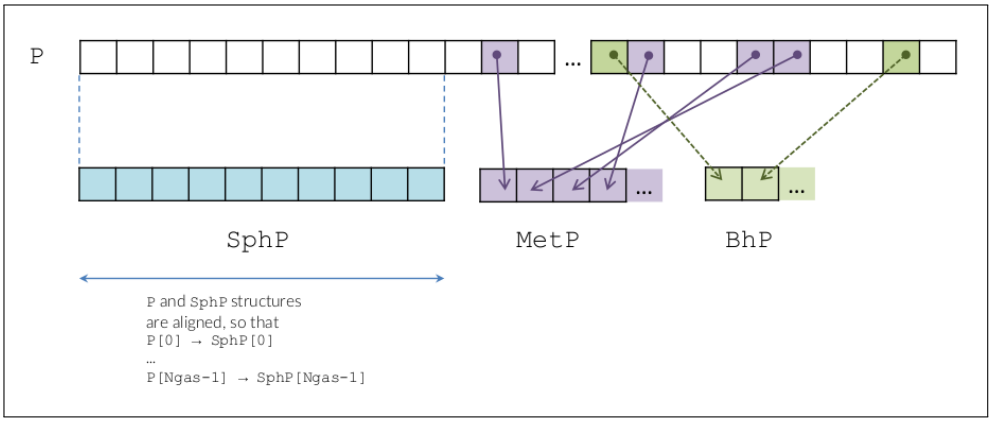


Figure 1: Pictorial view of the cross–indexing among the common P structures and the type-specific structures

Second, not all the physical processes are "active" in every run. For instance, one may want to perform an extremely large simulation of the large-scale structure of the Universe including only the gravitation and, potentially, some relativistic effect. Or to conduct a pure hydro-dynamical simulation in a non cosmological background, or to not include molecular cooling, or to switch off the cosmic rays, or, again, to use diverse hydro-dynamical schemes, etc.

Each of the mentioned "physical modules", and many others not listed here, bring in some dedicated variables into the particle data structure of the relevant type. Since including all the variables would result in already large structures becoming even larger than needed, the current strategy in GADGET is to use the conditional compilation which, thus became heavily ubiquitous throughout the whole code in order to switch on and off entire sections that are not needed (and that would require some unnecessary variables to be defined in the data structures to be compiled).

As it was mentioned in the Introduction, this has led to the so-called "`#ifdef`-catastrophe" that can be visually appreciated from an excerpt of the definition of the `P` and `SphP` structures in the code snippet 18, and a tiny fraction of the density loop in the code snippet 19 provided in the Appendix section A.

2.2 The concept of colonies

To address the problems stated in Section 2.1, we introduce the notion of *colonies*. Each colony corresponds to a different particle type and each type is considered as a collection of *properties*. Each property is defined as a `struct` (a `class` with default public visibility for its members) with fields corresponding to actual physical quantities (like mass or velocity) and functions which can be called specifically for this property. We also template the properties on the type of fields to allow an easy switch between, for example, `double` and `float`. To avoid always having to specify the template argument in the `PROPERTY` each time it is used, we created a separate header file with type definitions of the form `using ID=sID<unsigned int>`. We will return to the property structures in more detail after introducing all of the notions necessary for the proper understanding of how the property member functions are called in our setting.

Furthermore, to allow for future flexibility of the way of storing our colonies, we introduce a helper class `colony_storing_vector` which our colonies inherit from (in fact, we template on the parent class). All the function calls to the colony class should be delegated to the parent storing class (see the example in the code snippet 1). For the moment we choose the most straightforward solution to exploit the cache and store each property in `std::vector`. The number of properties in each colony can vary, but is known at compile time, which allows us to template the vector on all of the properties by using variadic templates (which are supported since the C++11 standard). To store all of the property vectors in one single container, we use the variadicly templated `std::tuple`, a C++ container that can store variables of different type at the same time. Below are two code snippets 1 and 2 from the definition of the classes to illustrate how this construction eventually looks like:

Code snippet 1: Part of the colony class definition

```
template <typename Storing>
class colony : public Storing {
public:
 template <typename T>
 static constexpr bool has() {
   return Storing::template has<T>();
 }
 ...
}
```

Code snippet 2: Part of the colony_storing_vector class definition

```
template <template <typename> typename Allocator, typename... Type>
struct colony_storing_vector {
 using tuple_type = std::tuple<std::vector<Type, Allocator<Type>>...>;
 tuple_type the_tuple;
 template <typename T>
 static constexpr bool has() noexcept {
   return (std::is_same_v<T, Type> || ...);
 }
 ...
 }
```

We have included the function `has<T>()` into the snippets as it's an important function used to quire the colony on if its members actually have some property.

Leaving the explanation of the small implementation details to the code documentation, let us demonstrate a dummy example of how a colony can be created and its data accessed. Please look at the snippet 3 to find the example (where we assume that the operator << is defined for the `field1` of the property structure `PROPERTY2`).

Code snippet 3: Dummy colony creation and member access example

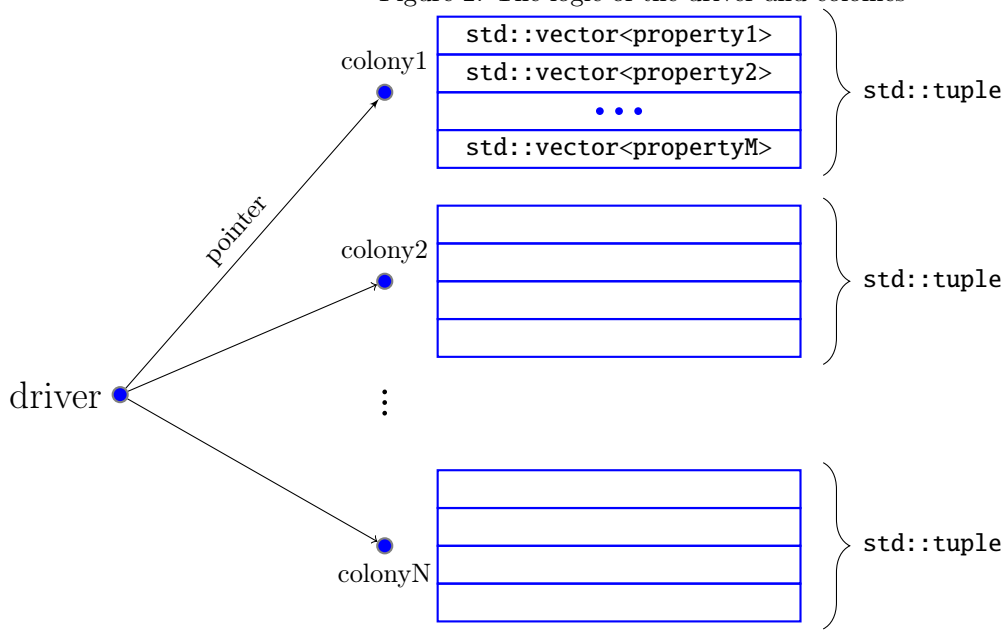
```
auto c1=make_colony<PROPERTY1,PROPERTY2,PROPERTY3>();
...
auto temp=(std::get<1>(c1.the_tuple))[0].field1;
std::cout << temp << std::endl;
```

Note how the `auto` keyword conveniently saves us the trouble of typing the complicated templated type of the variable for the colony or knowing in advance what the type of the `field1` even is. Here `std::get<NUMBER>` needs a `NUMBER` known at compile time to get the access to a given tuple member which corresponds to a property in the order they were given when calling `make_colony`. In fact, to loop over some property data inside a colony, the iterator functions `begin()` and `end()` have also been implemented, but their use will be demonstrated at a later time to avoid overburdening the reader for the moment.

Listing the properties as the template arguments when creating the colony replaces the need to put the `#ifdef`s inside any structures keeping the physical variables. Adding/activating new properties now requires the "user" (that is, a GADGET developer) creating their own header with that property rather than going inside the common particle structure file. Also, note that the "cross reference problem" simply does not exist anymore. As to how the colonies help with the other part of definitions coming from inside the physical functions will be clear in later sections.

One can notice that the syntax of the accessing the data is not very "user friendly" and requires manually looping over all colonies if we want to call some function for every particle. To greatly simplify the handling of colonies for the programmer, we introduce the class discussed in the next section - the `driver`.

Figure 2: The logic of the driver and colonies



2.3 The driver class

As we have stated in the previous section, the driver has been created to manage the colonies. We template it on the colony types using, again, variadic templates as we, in principle, do not know how many colonies there will be. We store the colonies as a tuple of pointers, `std::tuple` container has been chosen because the colonies are of different type. The snippet 4 shows part of the class declaration code with a constructor and an alias for the complicated tuple of pointes.

Code snippet 4: Driver class declaration

```
template <typename... ColonyTypes>
class Driver {
public:
  using tuple_type = std::tuple<ColonyTypes*...>;
  Driver(ColonyTypes&... args) : colonies{&args...} {}
  ...
```

The Figure 2 illustrates the final logic of our data structures with the current choice of the implementation for the `Storing` class.

To be consistent with naming of construction functions we call the function to create a driver `make_driver`. Snippet 5 illustrates how to use it.

Code snippet 5: How to create a driver

```
  auto c1=make_colony<PROPERTY1,PROPERTY2,PROPERTY3>();
  auto c2=make_colony<PROPERTY1,PROPERTY3,PROPERTY4>();
  auto c3=make_colony<PROPERTY2,PROPERTY3,PROPERTY4>();

  auto d=make_driver(c1,c2,c3);
```

Now let us illustrate one of the possible ways to work on colonies data (more cases will be discussed in the next section). Let us suppose we want, for our dummy example, to print all the data of a certain PROPERTY for all the particles that have the property in question. Please look at the code snippet 6 and then read the explanations following it.

Code snippet 6: The first example of caling a function on colonies

```
template <typename T, std::size_t N = 0>
void example_function1() {
  if constexpr (N < sizeof...(ColonyTypes)) {
    auto colony = std::get<N>(colonies);
    if (colony->template has<T>() ){
      T::example_function1(colony->template begin<T>(), colony->template end<T>(),
                           *this);
    }
    example_function1<T, N + 1>();
  }
}
```

First, we need to loop over all of the colonies, but as there is no straighforward way to have a runtime loop over `std::tuple` members, we need to implore compile time loop or compile time recursion. We have chosen to use recursion in this example, please refer to the Section 2.5 to see how compile time loop can be written. The recursion is implemented by templating a function on an extra integer parameter and giving it a default value of zero. The stopping condition is implemented via compile time construct `if constexpr` available since C++17 standart. We only need to call the `example_function1` on the colonies which actually have the property that the function in question has been implemented for, so we put an `if` condition for that using the function `has` which has been introduced in Section 2.2. The property structure acting as a template argument must have the function `example_function1` implemented inside its body. See code snippet 7 for the example of a property struct. Note that the function must be `static` so that we could do one function call for many particles with the `PROPERTY`.

Code snippet 7: A dummy property struct with an example function implemented

```
template <typename T>
struct PROPERTY {
  T field1;
  T field2
template <typename I, typename D>
static void example_function1(I first, I last, D&){
  while (first != last) {
    std::cout << first->field1 <<" "<< first->field2 << std::endl;
    ++first;
  }
}
};
```

Now we if we call `d.example_function1<PROPERTY>()`, where `d` is the driver, we get the fields of the `PROPERTY` printed from all of the particles that have it. Note that, if called from

another function different from the `main`, the syntax for calling any functions from the driver class looks like `d. template example_function1<T>();`. To simplify it, it's possible to define a utility function as show in snippet 8

Code snippet 8: A helper function simplifying syntax

```
template <typename T, typename D>
void example_function1(D& d) {
  d. template example_function1<T>();
}
```

Now we can use the line `example_function1<T>(d);` to call our function. The Section 2.5 will illustrate how to create new functions working with the driver and colonies in detail for various situations.

2.4 An Augmented Index

To conclude the collection of basic tools for working with the colonies data, let us introduce one last structure. For certain tasks, like the search of the nearest neighbours, we do not need to go through all of the particles of the colony in order, but instead we might need to retrieve the various particle information based on its position in the oct-tree. The current GADGET, whose algorithms we want to preserve whenever possible, just uses the indices of the particle array `P[i]` inside the tree, but now our particles are not stored in one array anymore, so we need to create something that would replace the call to `P[i]` to get the data. It is convinient to introduce a special utility structure, which we call an `augmented_index`, that allows us to retrieve all the informations of a given particle. Such index must know the `colony` to which the referenced particle belongs to. As we do not know the type of colony in advance, we store the address of the colony in a `void*`, and then use the driver for performing the right cast (since it knows both the types and the addresses of the colonies). The definition of this struct is very simple, so we provide the whole of it in the snippet 9.

Code snippet 9: The augmented index struct definition

```
struct augmented_index {
  std::size_t idx;
  void*       colony;
  friend bool operator==(const augmented_index& x, const augmented_index& y) {
    return x.idx == y.idx && x.colony == y.colony;
  }
  friend bool operator!=(const augmented_index& x, const augmented_index& y) {
    return !(x == y);
  }
};
```

The idea might not seem very clear to the reader for the moment, but we will see the ease of its use when we describe the treewalk in section 3.3. For now it is enough to know how one can create an augmented index and then use it with the help of the driver class function `get()` as presented in the following code (snippet 10):

Code snippet 10: The augmented index usage example

```
  augmented_index it{NUMBER, &colony};
  auto* something = d. template get<PROPERTY>(it);
  std::cout << something->field1 << std::endl;
```

We provide the function `get` from the `driver` class in Snippet 11 for better understanding of the mechanic. Note that we have already seen the compile time recursion and `if constrexpr` in Section 2.3.

Code snippet 11: The get function to work with the augmented index

```
template <typename T, typename I, std::size_t N = 0>
  T* get(const I& x) {
    if constexpr (N < sizeof...(ColonyTypes)) {
      if constexpr (std::tuple_element<N, std::tuple<ColonyTypes...>>::type::
                        template has<T>()){
```

```
      if (x.colony == std::get<N>(colonies)){
        auto first = static_cast<std::tuple_element_t<N, tuple_type>>(x.colony)
                     ->template begin<T>();
        std::advance(first, x.idx);
        return &*first;
      }
    }
    return get<T, I, N + 1>(x);
  } else {
    return nullptr;
  }
}
```

2.5 Practical guide to writing new functions

2.5.1 A function that does something for each particle having a particular property

We have already shown one of the most important ways to create functions to work with the colonies in the Section 2.3, specifically in code snippets 7 and 6. That case deals with applying some function to all particles in a simulation having a particular property. To summarize, you need to

i) Create `some_function` inside the `PROPERTY` structure file that your function will work with. The function should have the following features:

- it should be static (so that we could do one function call for many particles);
- it should have the `first` and `last` iterators as arguments. We template on the iterator type for simplicity.

That is, the function should look similar to

```
template <typename I, typename D>
static void some_function(I first, I last, ...) {
  while (first != last) {
    do_something(first,...);
    ++first;
  }
}
```

where the dots just stand for whatever other parameters you need.

ii) Add an entry with your function to the `driver` class, as shown below

```
template <typename T, std::size_t N = 0>
void some_function( -your_arguments-) {
  if constexpr (N < sizeof...(ColonyTypes)) {
    auto colony = std::get<N>(colonies);
    if (colony->template has<T>() ){
      T::some_function(colony->template begin<T>(),
```

```
                                colony->template end<T>(),
                                 -your_arguments-);
      }
      some_function<T, N + 1>( -your_arguments-);
    }
  }
```

Note that if you have several functions with similar signatures, you don't have to repeat the procedure for all of them, but can create one general function and pass your different functions as a template argument. We use this method in Section 3 in the code snippet 13.

It is also worth mentioning that, even though we do have to modify the `driver.hpp` file, common for developers working with different properties, this does not lead to adding any `#ifdef`s inside the `driver` class as the templated function in a header that is never actually called in the code gets ignored by the compiler anyway (no template instantiation happens).

2.5.2 A function that does something for each colony as a whole

There are situations when you need to perform some operation on a colony as a whole, for example resize the colony. This will be utility/helper functions unlike the ones in the previous section where the functions will most likely correspond to some physical process. Now there is no need to touch any of the properties files, but instead one needs to modify the `colony.hpp` and `colony_storing_vector.hpp` file (the latter might be replaced by another one should we choose to store our colonies differently in the future). So, the steps to take are the following (we are going to use the resize function as an example):

i) Add an entry with your function to the `driver` class, as shown below

```
  template <std::size_t N = 0>
 void resize(unsigned long int* new_size ) {
   if constexpr (N < sizeof...(ColonyTypes)) {
     auto colony = std::get<N>(colonies);
     colony->resize_v(new_size[N]);
     resize<N + 1>(new_size);
   }
 }
```

Technically, you could keep the same `resize` name everywhere since the functions belong to different classes, but we changed the name of one of them to make the code more human readable. For some functions, this step will actually be the only one needed if your function doesn't need looping over all of the properties inside the colony.

ii) (If you do need to loop over all of the properties) create an entry for the function inside the `colony` class file:

```
  void resize_v(unsigned long int& new_size){
   Storing::vect_resize(new_size);
  }
```

iii) (If you do need to loop over all of the properties) create an entry for your function inside the `colony_storing_vector` class file:

```
1   void vect_resize(unsigned long int& new_size){
2    std::apply ([&] (auto &... vect){(vect.resize(new_size), ...);}, the_tuple);
3   }
```

In this example we have utilized the so-called λ-function to loop over the tuple elements. If the function is too complicated to be turned into a λ, one can always write the same compile-time recursion we have used to go though the colonies in the driver.

3 The Gadget Algorithm

3.1 Domain decomposition

Large astrophysical simulations require the particles to be distributed among many different processors due to memory limits of a single machine. The procedure to distribute the data is usually referred to as domain decomposition. To perform domain decomposition, GADGET uses a scheme based on a space-filling fractal, the Peano-Gilbert curve (see Figure 3). We effectively map 3D space onto a 1D curve, which then can be cut into domains. Such mapping actually preserves locality, that is points close in 3D space are also close along the curve, which produces a small surface-to-volume ratio thus allowing to reduce communication between processors. This method also guarantees that the error of the tree force does not depend on the number of the MPI processes. For more information on space filling curves, we refer the reader to [2].

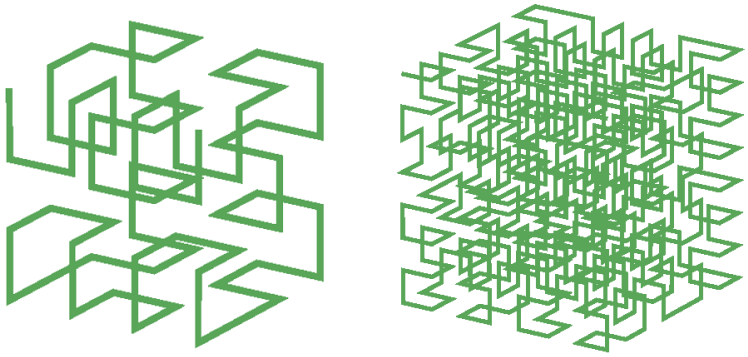


Figure 3: 3-dimentional Hilbert curves or 2nd and 3rd order

GADGET tries to distribute the particles in such a way that not only the memory needed by each MPI process is roughly the same, but also the computational burden is similar, so the algorithm is a bit more complicated than a naive cutting of the curve, but we leave out a detailed explanation as it is not relevant to our work. The general workflow is as follows:

- calculate Peano-Hilbert keys for all particles based on their coordinates;
- construct the global top-level tree. This is done by considering the list of Peano-Hilbert keys and recursively chopping it into pieces of eight segments until each segment holds at most a certain number of particles; for further details, we address the reader to section 3.2.
- find the split of the particles such that the workload is balanced;
- exchange the particles according to the determined domain split.

In fact, to better balance the work, the list of Peano-Hilbert keys is not simply cut into the number of pieces equal to the number of MPI processes, but there is a parameter called `MULTIPLEDOMAINS` governing how many chunks one process should have. See Figure 4 for the visualization of domain decomposition between 4 processors, the left having

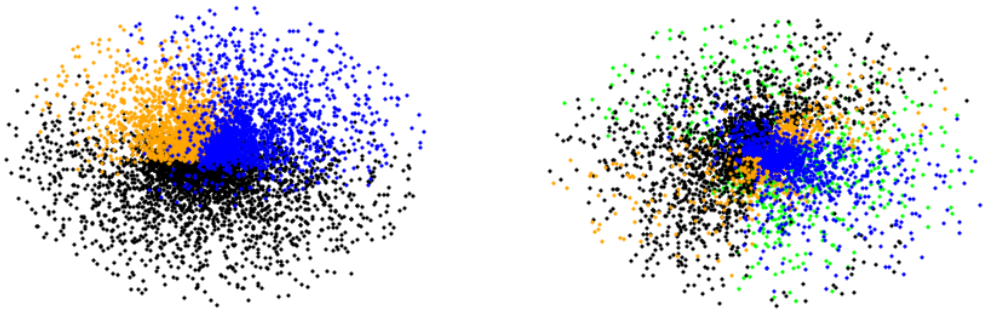


Figure 4: Visualization for domain decomposition for 4 processors with parameters `MULTIPLEDOMAINS=1` and `MULTIPLEDOMAINS=16` respectively.

`MULTIPLEDOMAINS=1` and the right `MULTIPLEDOMAINS=16`. On the left one the "green" process got all of its particles in the center region, thus we cannot see them in the picture.

We have not changed anything in the logic of the algorithm itself, however, we have adopted C++ tools of programming even where it was not strictly required by the changed core data structures. For example, most of the `array`s have been replaced by `std::vector`s and the functions working on the same objects have been grouped into classes together with those objects wherever possible. The main class for domain decomposition has been named `CDomainTree` and roughly corresponds to the file `domain.c` of the initial code. We tried to keep the naming condition as close to the original as possible so eventually the call to the function `domain_Decomposition(...)` is simply replaced by `tree.domain_Decomposition(...)` where `tree` is a variable of `CDomainTree`.Code snippet 12 shows part of the class declaration.

Code snippet 12: Part of the `CDomainTree` class declaration

```
class CDomainTree{
  ...
  std::vector<local_topnode_data> topNodes;
  template <typename D, typename P>
  void domain_Decomposition(int UseAllTimeBins, global_vars& global,
                                global_data_all_processes& All,D& d, P& pex);
  ...
  int local_refine(int i, double countlimit, double costlimit);
  void domain_allocate(global_vars& global,global_data_all_processes& All);
  template <typename D, typename P>
  int domain_decompose( global_data_all_processes& All, D& d, P& pex);
  void domain_findSplit_work_balanced(int ncpu, int ndomain);
  void domain_assign_load_or_work_balanced(int mode, int multipledomains);
  void domain_findSplit_load_balanced(int ncpu, int ndomain);
  void determine_domains(std::vector<std::vector<int>>& dest);
  template <typename D>
  int domain_determineTopTree(D& d);
  void domain_insertnode(std::vector<local_topnode_data>& treeA,
```

```
                          std::vector<local_topnode_data>& treeB,
                          int noA, int noB);
  template <typename D>
  void domain_sumCost(D& d);
  void domain_walktoptree(int no);
  int  domain_check_memory_bound(int multipledomains,global_data_all_processes& All);
  template <typename D>
  void domain_findExtent(D& d);
  void peano_hilbert_order();
  void domain_add_cost(std::vector<local_topnode_data>& treeA,
                       int noA, long long count, double cost, double sphcost);

};
```

The functions that *had* to be changed are those that work directly with the particle data. For computing the Peano-Hilbert keys we followed the route of the code snippet 11, but implemented a more general function in case one needs to perform a similar operation in the future, see snippet 13.

Code snippet 13: A general function to calculate something for all particles

```
template <typename T, typename F, std::size_t N = 0>
  void compute_for_each_particle(std::vector<unsigned long long>& result_vector,
                                 long long int& ind, F f,
                                 std::array<double,3>& p1, double& p2) {
    if constexpr (N < sizeof...(ColonyTypes)) { //recursion through colonies
      auto colony = std::get<N>(colonies); //got a particular colony
      if (colony->template has<T>() ){ //if it has a required property
        T::get_result_vector(result_vector,p1,p2, ind,f,colony->template begin<T>(),
                             colony->template end<T>());
      }
      compute_for_each_particle<T,F, N + 1>(result_vector,ind,f,p1,p2);
    }
  }
```

The more interesting things had to be implemented to exchange the data. So far we were thinking of particles only as living in colonies, not existing as a separate object in our code. In Section 2.4 we have already introduced the `augmented_index` which allows us to retrieve a particular property of a given particle, but this time we need to retrieve *all* of the particle information to send it to the MPI process it belongs to according to our domain decomposition. To solve this in the most structured and clear way, we have created a class `particle` which is essentially "a slice" of a colony at a certain index. This class is variadicly templated the same way as is the `colony_storing_vector`, but instead of having a tuple of pointers to vectors, it simple has a tuple of values. It is initially constructed by passing a reference to a colony to resolve all of the template arguments. The snippet 14 shows part of the definition of the class.

Code snippet 14: Part of a particle class definition

```
template <typename... Type>
struct particle {
  using tuple_type = std::tuple<Type...>;
  tuple_type a_tuple;

  //enough to deduce the types, works for an empty colony too
  particle(const colony_storing_vector<std::allocator, Type...>& c){
  };
  ...
}
```

Its other functions allow to "extract" a particle with a particular index, add a new particle to a colony or replace a particle at a given place (this function is needed since we don't want to move large amounts of data and thus, when a particle was extracted to be sent away, a new one received can be put directly at its place). We think its instructive to put the code snippets for one of the functions here (code snippet 15) because they use the compile-time loop with the help of `std::index_sequence_for` to deal with going over a tuple elements and we have not seen this possibility of dealing with a tuple before.

Code snippet 15: Part of a particle class definition

```
 ...
 template <typename M>
    void slice_assign(M& a, const M& b){
      a=b;
  }
  template<std::size_t ...I>
  void replace_help(int ind, std::index_sequence<I...>,
                        colony_storing_vector<std::allocator, Type...>& c){
    (slice_assign(std::get<I>(c.the_tuple)[ind],std::get<I>(a_tuple)),...);
  };
  void replace_particle(int ind, colony_storing_vector<std::allocator, Type...>& c){
    replace_help(ind, std::index_sequence_for<Type...>{},c);
  };
```

Let us give a brief explanation to what happens in these functions. When the `replace_particle` is called with a reference to a certain colony, the command `std::index_sequence_for<Type...>` is used to general the `std::index_sequence<I...>` for the parameter pack `<Type...>`. The `replace_help` function is basically a compile-time loop which is going to call the `slice_assign` function for every vector inside the tuple of the `colony_storing_vector`.

Coming back from the technical C++ details, as the final point of this section, let us mention that due to how workload balancing of the domains works, there is some freedom in how the domains will be split, so re-implementing it with our data structures does not always guarantee *exactly* the same result, but the difference only concerns a few border particles. For some cases we have reproduced the domain decomposition precisely.

3.2 Gravity Tree

To achieve spacial adaptivity, GADGET is using a hierarchical multipole expansion algorithm, also called a tree algorithm. This algorithm, firstly introduced in cosmological simulations by Katz et al. [5], groups particles into cells allowing their gravity force to be replaced by a single multipole which greatly reduces the number of calculations required compared to a direct summation method. The hierarchical grouping required is achieved by the recursive division of space into 8 daughter nodes starting with the root node encompassing all of the particles. Forces are then calculated by walking the tree and opening the nodes until a stopping criteria is met and the node is small or distant enough to provide a good approximation by its multipole moment. By changing that stopping criteria, the error of the final result can be carefully controlled.

The gravity tree is build by the same principle that has been used before to build the domain tree. We have not actually gone into any details about the tree building when discussing the domain decomposition, so we can do that in this section.

First, let us note that we followed the same general idea when recreating the tree building algorithm as discussed in Section 3.1 - only changed what was necessary for the new data structures access, replaced arrays with `std::vector`s and created a special class for the tree and associated functions - the `CForceTree` class. The excerpt of the class definition is presented in code snippet 16.

Code snippet 16: Part of the `CForceTree` class declaration with the most important functions

```
template <typename MyFloat>
class CForceTree{
public:

  ...
  template <typename D>
  int build_single(int npart, struct unbind_data *mp,
                   global_data_all_processes& All,
                   CDomainTree& domt,global_vars& g,D& d,
                   std::vector<MyFloat>& rndt);
  template <typename D>
  int build(int npart, struct unbind_data *mp,
            global_data_all_processes& All,
            CDomainTree& domt,global_vars& g, D& d,
            std::vector<MyFloat>& rndt);
  NODE<MyFloat>* Nodes;
  extNODE<MyFloat>* Extnodes;
  ...
  std::vector<augmented_index> AugP;

private:
  ...
  void flag_localnodes( CDomainTree& domt);
  void exchange_pseudodata( CDomainTree& domt);
  void update_pseudos(int no, global_data_all_processes& All);
```

```
  void create_empty_nodes(int no, int topnode, int bits,
                          int x, int y, int z, int *nodecount,
                          int *nextfree, global_vars& g, CDomainTree& domt);

  template <typename D>
  void update_node_recursive(int no, int sib, int father,
                             float LocSoft,global_data_all_processes& All,
                             global_vars& g, D& d);
  void insert_pseudo_particles(CDomainTree& domt,
                               global_data_all_processes& All);
  ...
};
```

Now let us briefly walk through the algorithm. First, to build the top-tree a set of nodes is created starting from the root nodes which corresponds to the entire computational box. The number of particles belonging in each node, determined by pure spatial information on their position, is computed and stored in each node. Then recursively, each node is refined in its 8 children sub-nodes each of which is "populated" in the same way. The refinement, i.e. the creation of sub-nodes in each existing node, stops when either the number of particles living in such a final node, or their forecast computational cost, is less than a given parameter. In this way, the tree structure, which we refer to as the "top-tree", contains a broad-level information about the distribution of small-enough bunches of particles. Such bunches, and not the single particles, will then be the actual targets for the domain decomposition. The information of the individual top-trees of each MPI tasks are then exchanged among all the MPI tasks so that every task has a complete picture of the particle distribution at broad level.

The domain decomposition, i.e. the distribution of work and load among all the MPI tasks, is then performed so to respect as much as possible i an even distribution of workload within the memory limits, and ii the spatial contiguity of the top-leaves assigned to each tasks.

Determining who is the target companion task that is candidate to receive and send particles to each other task is easily achieved by using the complete top-tree information. After that all the particles have been exchanged and each MPI task owns all its particles, then all the tasks can continue the construction of the entire tree for their particles. Such a constructions follows exactly the same logic than the top-tree except for the fact that the recursive process stops when a node contains a single particle instead of a bunch of particles.

3.3 Toy physics function

Due to some peculiarities of the domain decomposition algorithm, different order of storing data in memory may lead to slightly different domain decomposition results, thus depriving us of a sure easy way to test our code validity at that stage. To be able to really test our code for correctness, we have to compare with GADGET some physical results we can get from the simulation. The easiest function to do so is the one that finds all the neighbours of a given particle.

To do so, a toy function, `look_around` has been introduced both into the GADGET extract and our version of the code. For this purpose, staying true to the C++ principle of encapsulating tasks into separate classes, a new class `CTestTree` has been created. The code snippet 17 shows part of the class definition with its most important functions.

Code snippet 17: Excerpt of the class `CTestTree` definition

```
 class CTestTree{
public:
 ....

template <typename D>
void look_around(CForceTree<double>& ft, CDomainTree& domt,
                 global_data_all_processes& All, int Nparticles,
                 double radius, D& d);

private:
  ...
  template <typename D>
  int la_evaluate(CForceTree<double>& ft, global_data_all_processes& All,
                  D& d, CDomainTree& domt, int target, double radius,
                  int mode, int* nexport, int *ngblist);

  template <typename D>
  unsigned int la_evaluate_secondary(CForceTree<double>& ft, g
                                     lobal_data_all_processes& All, D& d,
                                     CDomainTree& domt, double R);

  template <typename D>
  unsigned int la_evaluate_primary(CForceTree<double>& ft,
                                   global_data_all_processes& All,  D& d,
                                   CDomainTree& domt, double R);

 ...
}
```

Since the GADGET algorithm uses a lot of calls to the particles structure with some random index `P[i]`, we need to rely on the `augmented_index` presented in Section 2.4 to replace the calls

the the particle structure that does not exist anymore. Obviously, it requires to be renewed before use every time after the domain decomposition has been executed. The introduction of `augmented_index` has been the most significant change to the code, other than that we have completely kept the GADGET algorithm, which can be schematicaly summarized as the following:

- Call the function `la_evaluate_primary(...)` to process the local particles, finding their neighbours and preparing the export list for the other MPI tasks that have neighbour particles;
- Exchange the data between MPI tasks;
- Call the function `la_evaulate_secondary(...)` to process the received "foreign" particles data.

The process of finding the neighbour particles consists of walking through the previously created gravity tree (discussed in Section 3.2) and calling the function `int CForceTree<...>::ngb_treefind_variable(...)` for each particle we need to process. This function, with the help of the gravity tree, finds all the particles within a given radius from the one it has been called for.

The results obtained have been used to check our code for correctness by comparing the lists of neighbours for each particle within a certain distance.

We note that, even though the function `look_around` is, in itself, merely "a toy" for creating a list of neighbours, rewriting it with the help of our engine and analyzing the performance is a very important task as it contains the neighbour search used in various real functions.

4 Performance

We present the visual results of timing our C++ application against the extract of GADGET we have based our code on at Figures 5-10. We have separately timed the key points of the algorithm - the domain decomposition (discussed in Section 3.1), tree building (Section 3.2) and our "toy physics function" called `look_around` (Section 3.3). The analysis have been performed on two "real-life" GADGET snapshot files - let us call them snapshot `A` and snapshot `B` (in fact the domain decompostion picture 4 from Section 3.1 is based on the snapshot `A` data). The first one has the data of 1446538 particles and the second - of 8069212. On the aforementioned Figures, we called our application "new" (blue) and the GADGET extract "old" (green). All the measurements have been taken at the HOTCAT cluster of INAF (Istituto Nazionale di Astrofisica) [3], [4].

Both GADGET and our application have been "crippled" by turning off the "gravity cost" of the particle, basing the domain decomposition on pure geometrical positions only, leading to heavy imbalance of work when distributing the data between many processors as only some of them got the particles from the dense area and thus ended up having the most number of neighbours for their particles. This explains poor scaling when going to 32 processors on our relatively small set of data.

We stress that, as we have worked with only a part of the real GADGET on a toy problem and the main goal of our work resided in creating a viable proof-of-concept for a new engine rather than presenting a fully optimized production code, the results of the relative performance are to be considered only as the demonstration that even without devoting any time to the actual optimization of the code, we were able to obtain only slightly worse timings. The performance analysis conducted has only had the goal of showing that there is no major increase in execution time. Both GADGET and our code are sensitive to various parameters, like `PartAllocFactor` responsible for the extra space allocated for particles to avoid having to move data after the number of particles has changed, reacting to the same changes in a different way, so we would not claim that the pictures represent best possible GADGET performance against our code best performance, but rather we simply demonstrate that the performance stays "around the same".

We see that the worst results have been obtained by the domain decomposition part, but there are two significant reasons to not consider this as a serious shortcoming. First, on the Figure 12, we show the timings of a real-life typical GADGET run, so one can see that domain decomposition takes under 2% of the run, thus having it slow down will not significantly slow down the whole simulation run.

Second, there have been a few decision made in favor of simplicity rather than better performance, which, however, do not present any potential block on the road to turning our code into a real simulation program. For example, one of the GADGET tricks to improve the performance consists of allocating all of the memory pool needed for various arrays at the start of the simulation and then managing this memory, never asking the system to allocate anything again. In principle, this can be translated to C++ which allows the use of a custom allocator for the `std::vector`, but this feature has not yet been implemented in this thesis, leaving fine optimization like rewriting an allocator function for future work.

Another possible important source of optimization is exploring different ways to store colonies. We have mentioned this when introducing the colonies, and that was the reason the colony class is templated on a general storing class in case we later want to change it without affecting the end engine user. We stress that this is a particular strength in our design because *i* it leaves us a total freedom in deciding such a crucial details as the memory pattern, and *ii* even more important, to modify it and experiment with different patterns (for instance: switching between a structures–of–arrays to array–of–structures paradigms) very easily without having to change anything else in the code.

Great optimization potential also lies in reconsidering the pattern of the data exchange of the domain decomposition. To better analyze the performance, we rerun our timing tests on a larger GADGET snapshot file,

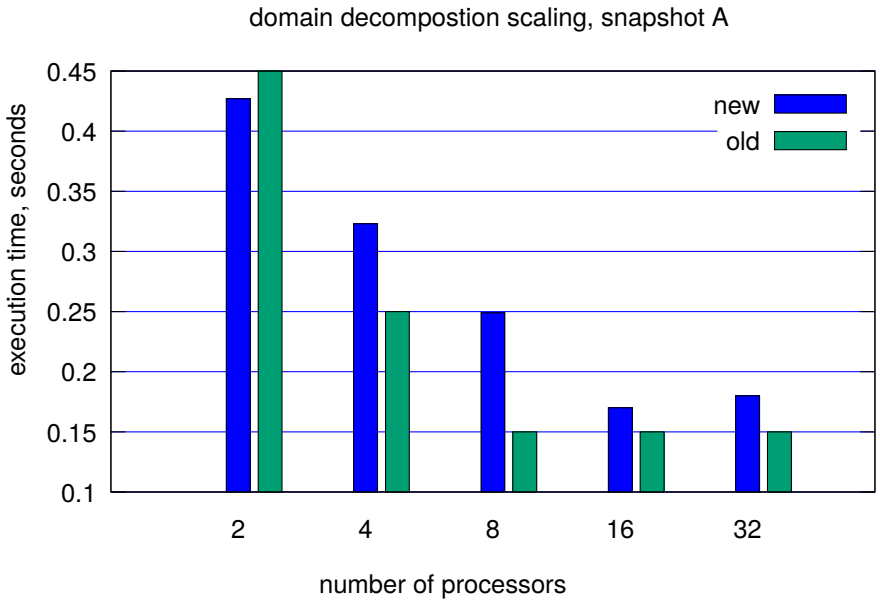


Figure 5: The timings of domain decomposition run on the snapshot file A

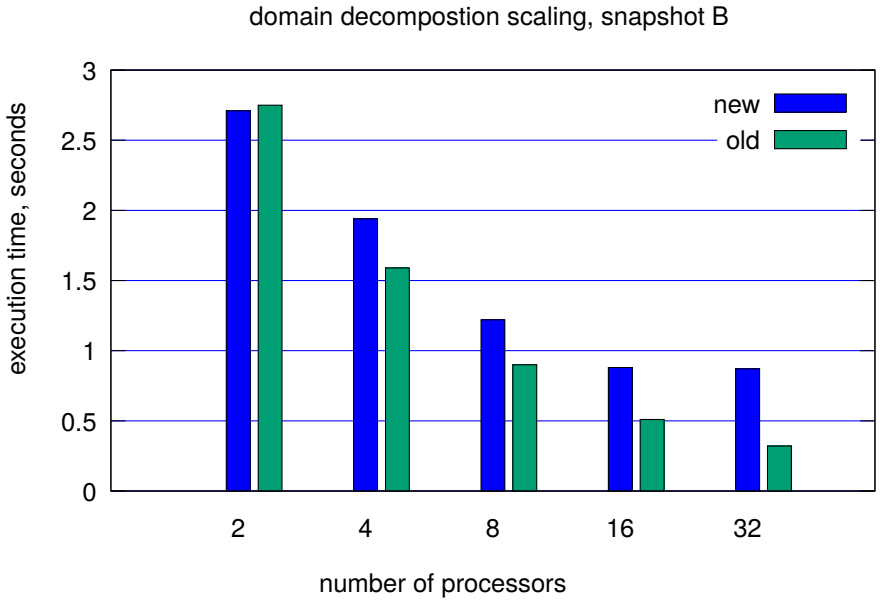


Figure 6: The timings of domain decomposition run on the snapshot file B

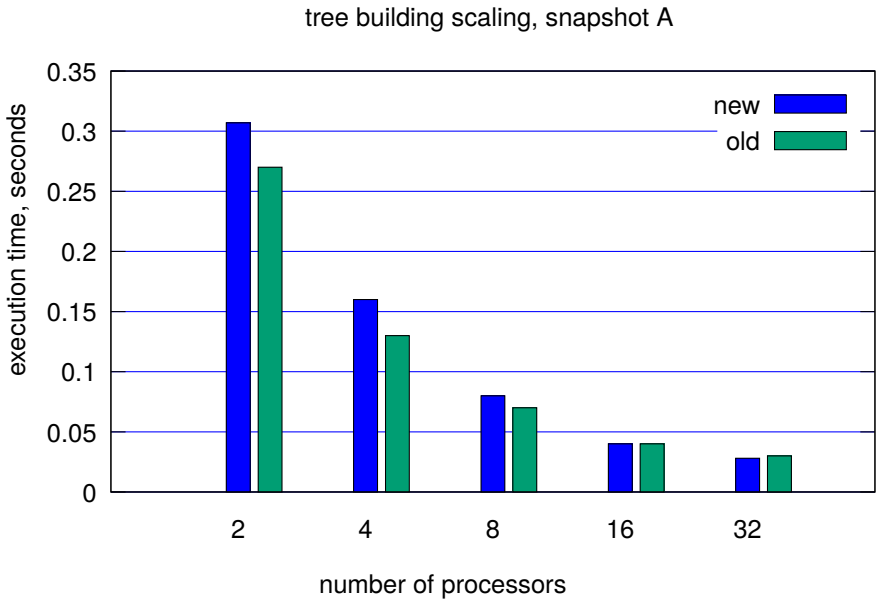


Figure 7: The timings of tree building run on the snapshot file A

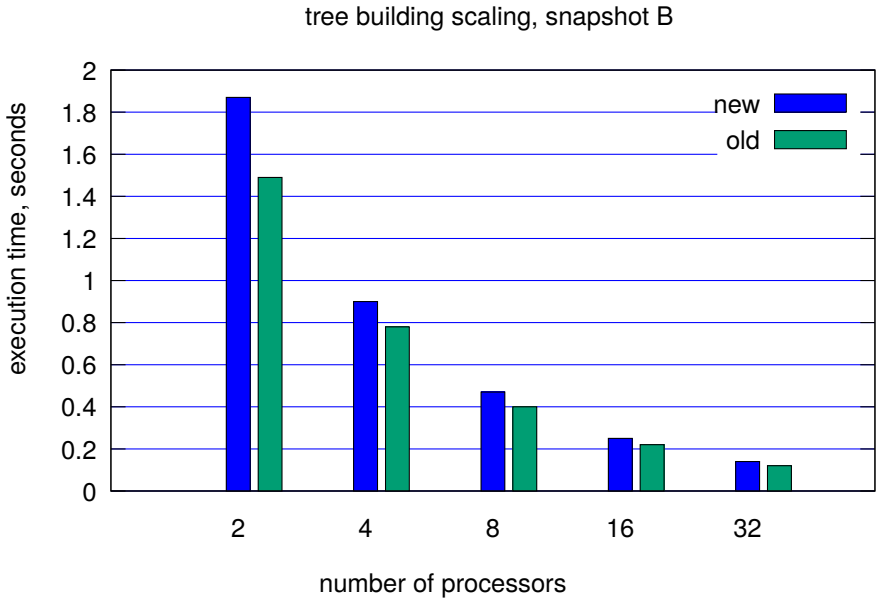


Figure 8: The timings of tree building run on the snapshot file B

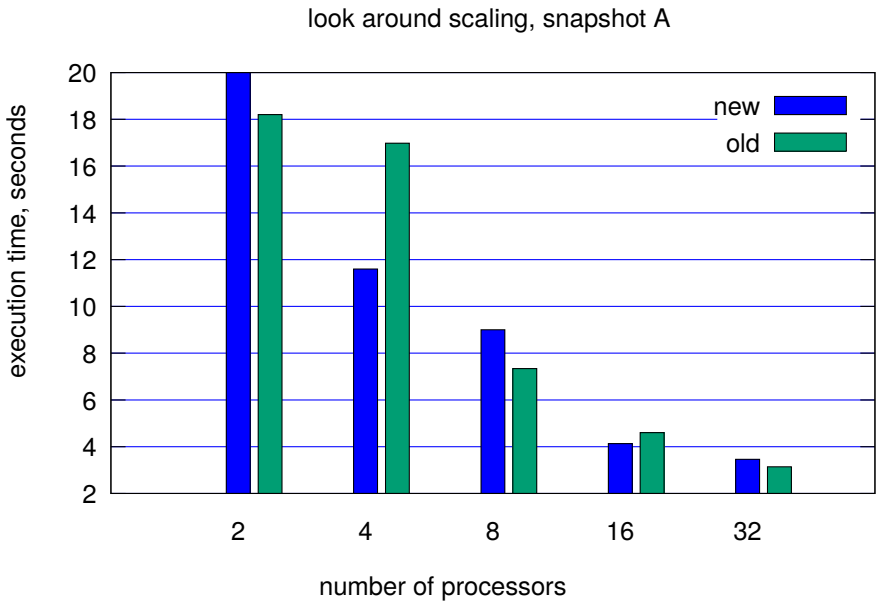


Figure 9: The timings of look_around run on the snapshot file A

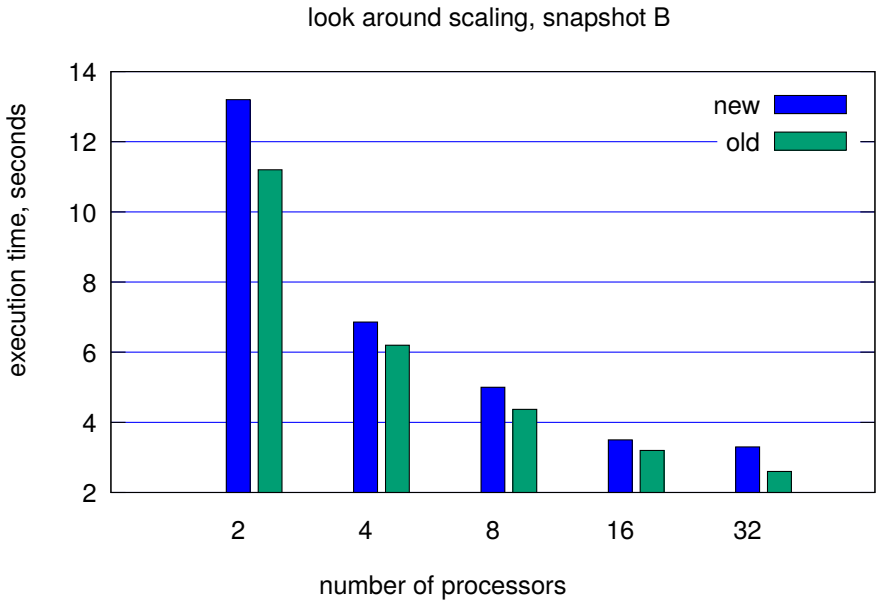


Figure 10: The timings of look_around run on the snapshot file B

which we will refer to as snapshot C, having 33 Millions of particles. We present profile results screenshots from the ARM-Allinea DDT/Map parallel debugger and analyzer[1] on Figures 13-15. By comparing the Figures 13 and 14, analyzing the runs of our application and the extract of GADGET on the same snapshot file C on 32 processes, we see that a significant fraction of the difference in the execution time is taken by the MPI calls. While the I/O and distribution parts behave quite similarly in the 2 codes, being dominated by the MPI communications, most of the discrepancy lies in the Domain-Decomposition. The GADGET code appears to be dominated by CPU-intensive code, that accounts for the creation and population of tree nodes and the setting-up of the tree data structures for linear-time traversal. As opposite, our code appears to be affected by a much larger MPI activity. However, from the Figure 15, we see that what is actually accounted as MPI time is attributed to a `MPI_Barrier` call; that actually indicates a huge imbalance in the computation/communication. In fact, as we deduce from the same figure, that call collects the MPI tasks at the end of a loop of point-to-point communications in which the particles are moved among processors and inserted in the right place, leading to most of the time being spent in memory access (as can be noted in the left gray panel of the same figure). Thus, we expect to improve the performance by adjusting our algorithm to improve the locality of the memory access calls.

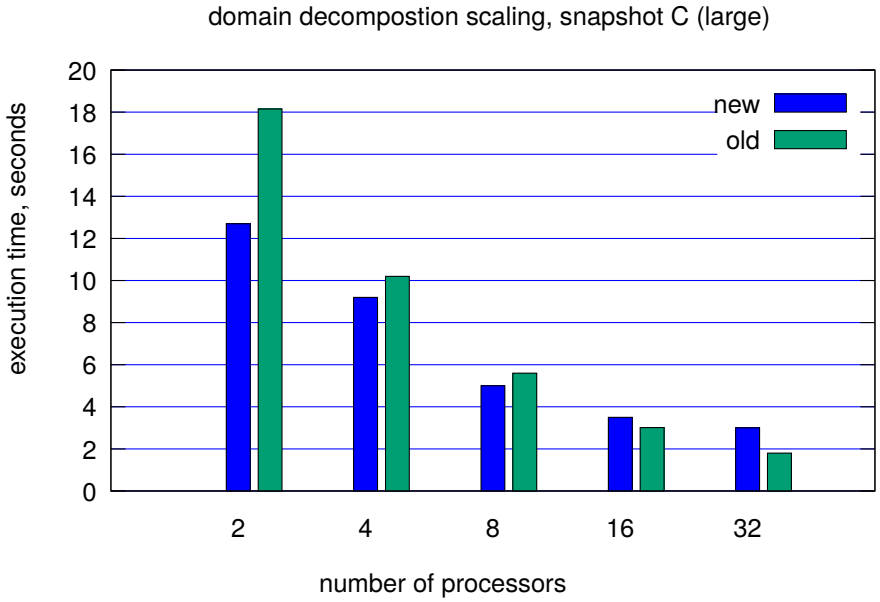


Figure 11: The timings of tree building run on the snapshot file B

[1]The usage of this application has been granted by the fact that L.T. participation in this work is accounted under his activity in the EU H2020 project EuroExa (https://euroexa.eu/) under Grant Agreement no. 754337

```
   treebuild                585.03    1.6%
   treeupdate               129.16    0.4%
   treewalk                4001.80   11.1%
   treecomm                  81.05    0.2%
   treeimbal               6187.90   17.1%
pmgrav                     1690.08    4.7%
sph                       16937.36   46.9%
   density                 6264.30   17.3%
   denscomm                 144.82    0.4%
   densimbal               1900.68    5.3%
   hydrofrc                2647.82    7.3%
   hydcomm                  105.78    0.3%
   hydmisc                 1172.63    3.2%
   hydnetwork                 0.00    0.0%
   hydimbal                1327.43    3.7%
   hmaxupdate                34.29    0.1%
domain                     623.79    1.7%
potential                    0.00    0.0%
predict                     28.64    0.1%
kicks                      291.15    0.8%
i/o                         12.15    0.0%
peano                      211.09    0.6%
sfrcool                   3238.27    9.0%
blackholes                 257.96    0.7%
fof/subfind                368.70    1.0%
smoothing                  513.41    1.4%
misc                       678.70    1.9%
```

Figure 12: The timings of typical real gadget application run

5 Conclusions and Future Work

We have presented the new data structures engine for the GADGET simulation code, rewriting the following core functions:

- domain decomposition;
- gravitational tree building;
- finding particle neighbours by using the tree.

The results have been tested for correctness against the GADGET code by comparing the neighbours found.

We have successfully created an easily modifiable "constructor" for possible code changes. During this work, we have already had an ample opportunity to experience the benefits of the engine. In fact, when we first completed our code prototype, we didn't have any use for `sph_particle_data`, which weights several hundreds of bytes, so it was never included into our work. The GADGET, however, had all of it present, even if as a dead weight. Obviously, to compare the timings to GADGET, especially for the domain decomposition, all the same data had to be present. After having noticed that discrepancy, we introduced the `sph_particle_data` property into the code - and it took us under 3 minutes to do that. The only things we had to do was creating the property file, including it and then adding the property to the list of templates when making the colony for the particles that have this property. Granted, this was never used in the "real calculations", but even for performing domain decomposition, the GADGET has SpH data array deeply intertwined inside the code and adding/removing it would

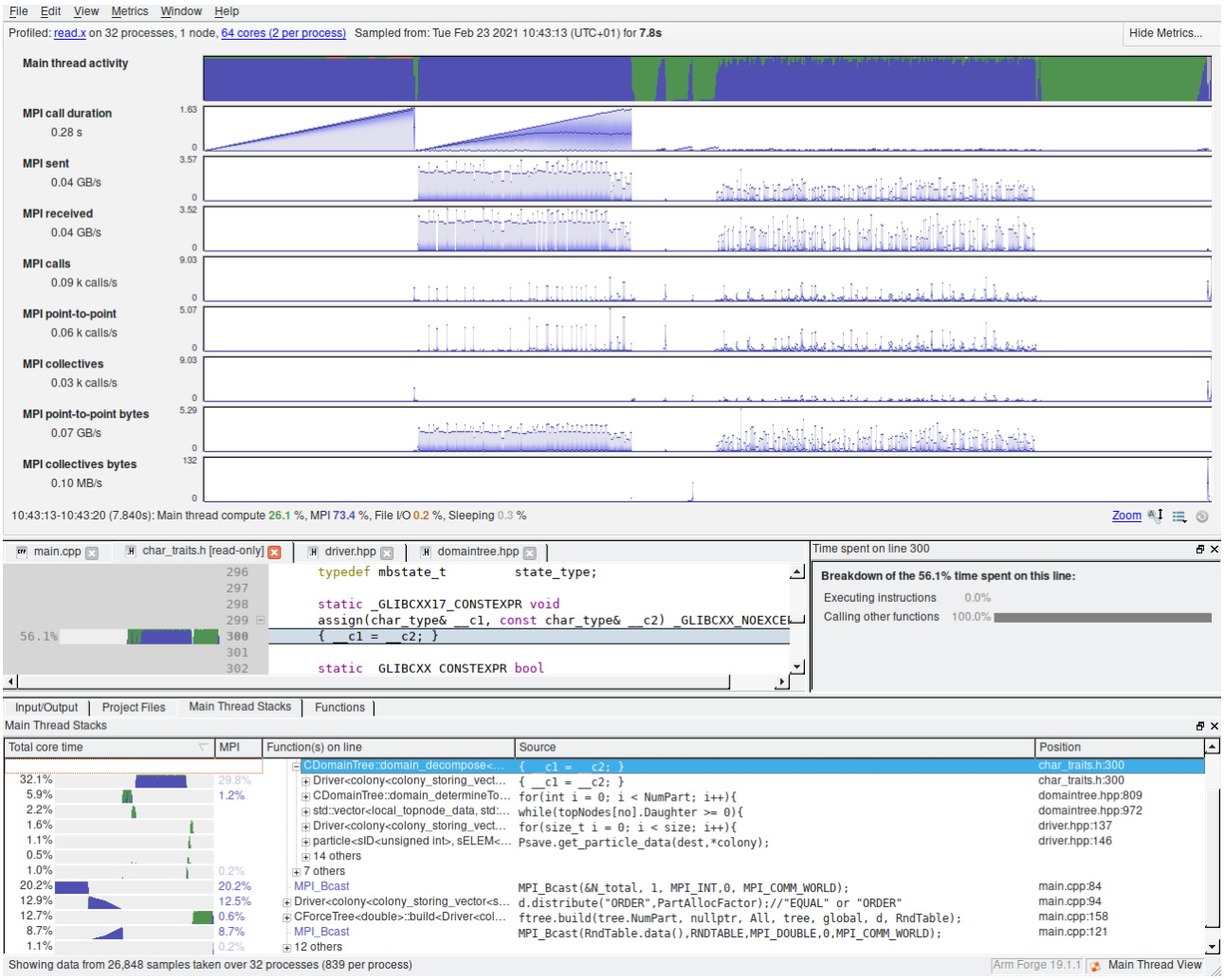


Figure 13: ARM/Allinea DDT/Map results for our code on the snapshot file C

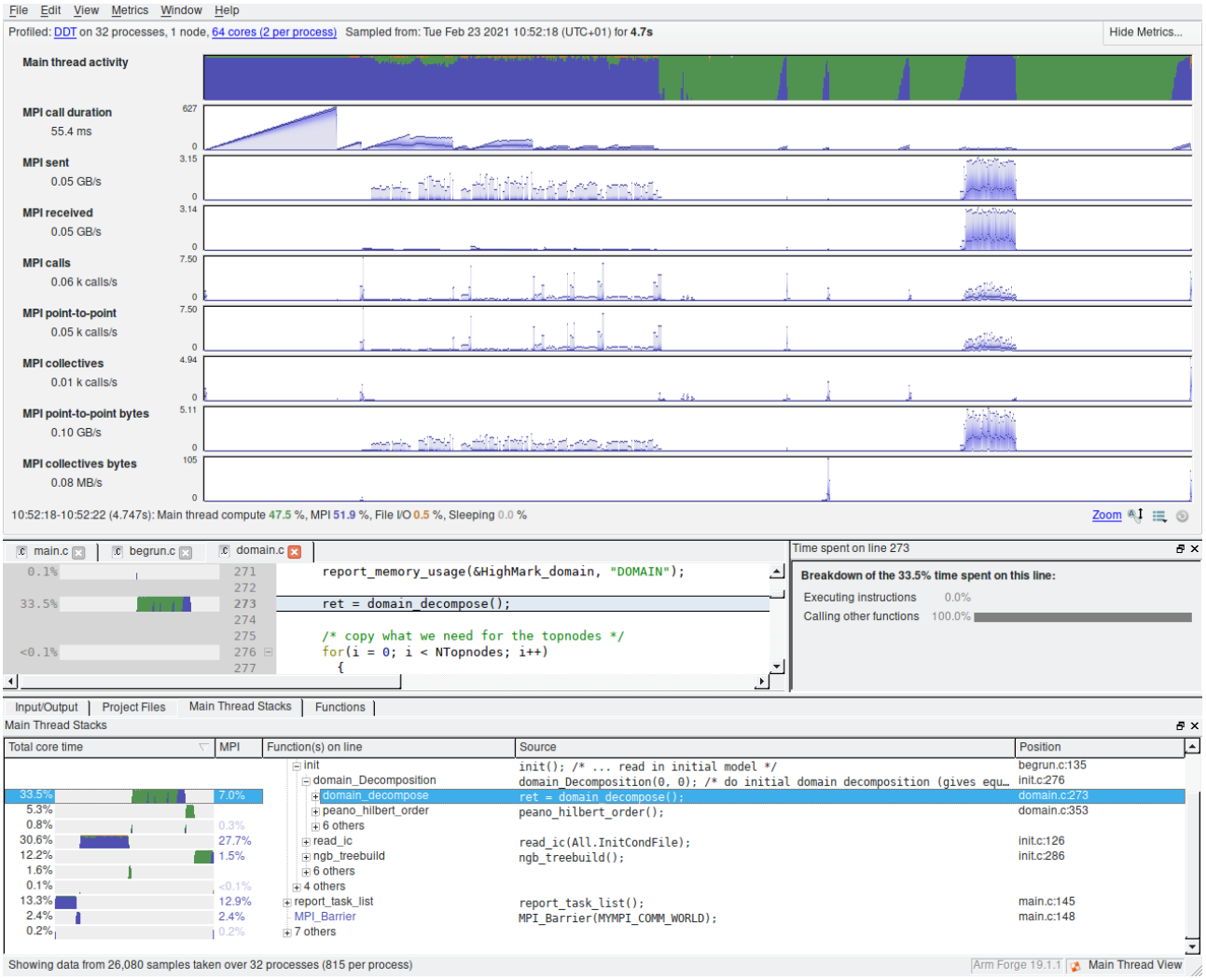


Figure 14: ARM/Allinea DDT/Map results for GADGET on the snapshot file C

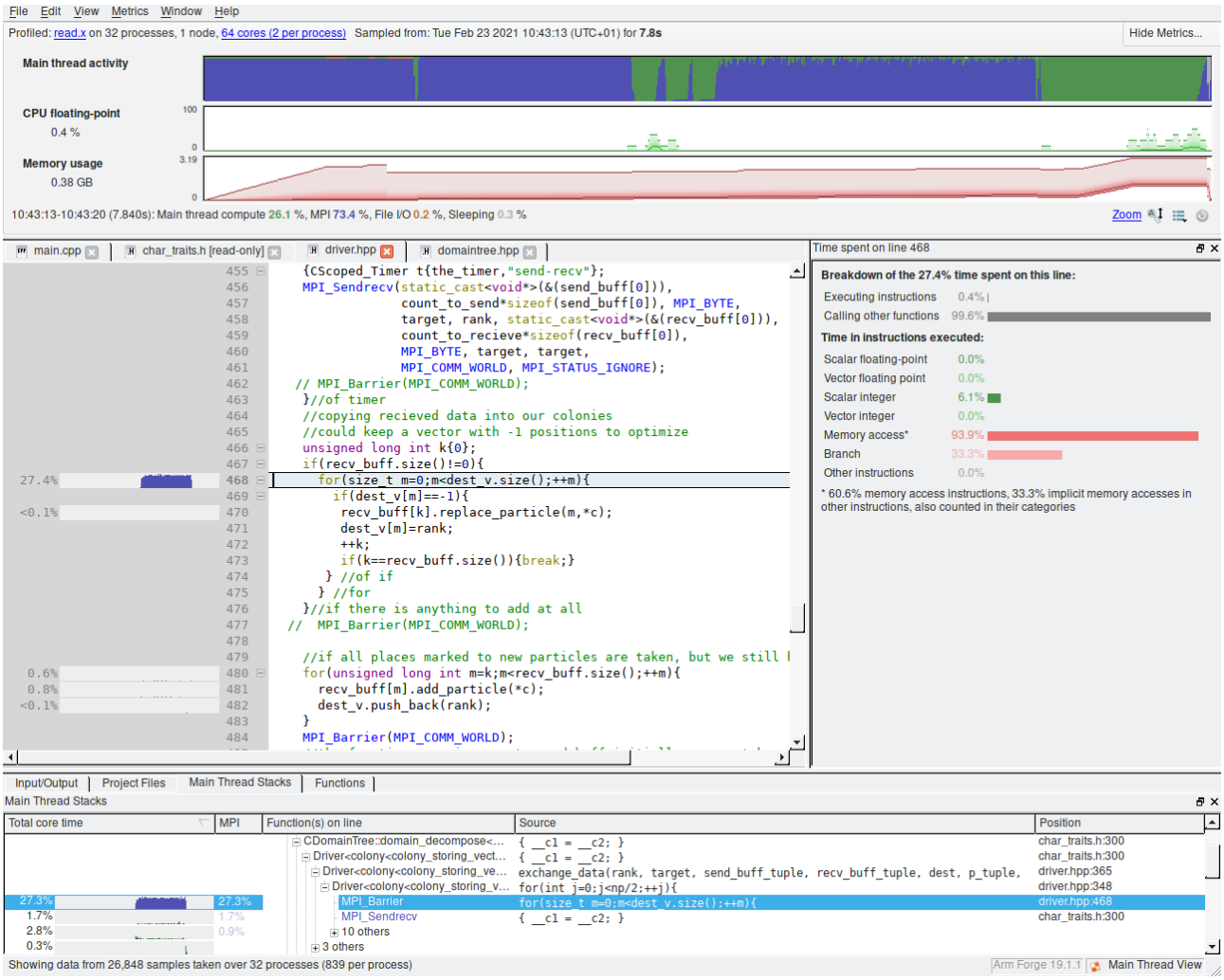


Figure 15: ARM/Allinea DDT/Map results for our code on the snapshot file C; details about the part discussed in the text.

require a serious effort. Thus, *we have demonstrated that the main goal of our proof of concept has been successfully achieved - modifying the key physics data structures has become a very easy task.*

However, much work remains to be done before our engine can be accepted for the real life production code.

First, we have only rewritten a toy function for finding particle neighbours, we need to expand this function into the real GADGET physics computation - the `density` loop. This, in principle, is similar to our `look_around` function, but requires a more general way of looking at things. While not presenting any fundamental obstacles, it, however, requires some serious work for completion.

Second, GADGET production code is huge with many modules and rewriting the whole thing in C++ might not be feasible in the near future or at all. As such, an interface for the old functions needs to be provided to allow incorporating the new core into the old codebase.

And last, but not least, we need to explore the optimization potential discussed in Section 4. Oftentimes, we strictly followed the GADGET initial code when rewriting it with our data structures, aiming to simply reproduce the same results rather than solve the computational task the best possible way with the tools given.
To summarize, the future work outline can be presented as the following scheme:

- Rewriting the density loop;
- Providing the interface for replacing the real GADGET core code;
- Optimization:
  - exploring different ways to store colonies;
  - adjusting algorithms to better fit with the new data structures;
  - adding a pool allocator.

References

[1] V. Springel, *The cosmological simulation code* GADGET-*2,* ArXiv:astro-ph/0505010v1

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

[3] G. Taffoni, et al. *CHIPP: INAF pilot project for HTC, HPC and HPDA,* ArXiv abs/2002.01283 (2020)

[4] S. Bertocco et al., *INAF Trieste Astronomical Observatory Information Technology Framework*, ArXiv abs/1912.05340 (2019)

[5] N. Katz, D.H. Weinberg, L. Hernquist, *Cosmological Simulations with TreeSPH*, Astrophysical Journal Supplement v.105, p.19

A The #ifdef catastrophe illustrations

Code snippet 18: Excerpt of the P and SphP structures definitions

```
extern ALIGN(32) struct particle_data{
  MyLongDouble Pos[3];   /*!< particle position at its current time */
  short int Type;   /*!< flags particle type.  0=gas,.. */
  short int TimeBin;
  MyIDType ID;
  integertime Ti_begstep; /*!< marks start of current timestep.. */
  integertime Ti_current; /*!< current time of the particle */
  MyFloat Mass;     /*!< particle mass */
  MyFloat Vel[3];   /*!< particle velocity .. */
  MyFloat GravAccel[3]; /*!< particle acceleration due to gravity */
#ifdef PMGRID
  MyFloat GravPM[3];  /*!< particle acceleration due to PM.. */
#endif
#ifdef FORCETEST
  MyFloat GravAccelDirect[3];  summation */
#endif
#if defined(EVALPOTENTIAL) || defined(COMPUTE_POTENTIAL_ENERGY) ||
defined(OUTPUTPOTENTIAL)
  MyFloat       Potential;    /*!< gravitational potential */
#endif
...
}

extern struct sph_particle_data{
  MyLongDouble Entropy;   /*!< entropy ... */ */
  MyLongDouble EntropyPred;
  MyFloat VelPred[3];   /*!< predicted SPH particle velocity... */
  MyFloat MaxSignalVel;     /*!< maximum signal velocity */
  MyLongDouble Density;
  MyLongDouble DtEntropy;
  ...
#ifdef TIME_DEP_ART_COND
  MyFloat Calpha, GradA[3];
#ifndef NOGRAVITY
  MyFloat Climit, Cgrav[3];
#endif
#endif
#ifdef AB_SHEAR
  MyFloat Chi[3][3];
  MyFloat Xix[3], Xiy[3], Xiz[3];
#endif
#ifdef MAGNETIC
  MyFloat B[3];
#ifdef WINDS
  MyFloat WindB[3];
#endif
  MyFloat BPred[3];
#ifdef MAGNETIC_SN_SEEDING
  MyFloat MagSeed[3];
```

```
#endif
...
#endif      // closes MAGNETIC
 #if (defined(SMOOTH_DIVB) || defined(DIVBCLEANING_DEDNER) ||
 defined(BSMOOTH) || defined(SMOOTH_ROTB) ||
 defined(LT_USE_DENSITY_IN_WEIGHT) ||
 defined(LT_SMOOTH_Z) ||
 defined(LT_SMOOTH_XCLD) || defined(LT_TRACK_WINDS) ||
 defined(VSMOOTH))
  MyFloat DensityNorm;
#endif
#ifdef TIME_DEP_ART_VISC
  MyFloat alpha, Dtalpha;
#ifdef AB_ART_VISC
  MyFloat visc_R, visc_OldDvel;
#endif
#endif
...
}
```

Code snippet 19: Excerpt from the density loop routine

```
#if defined(LT_STELLAREVOLUTION)
    if(TimeBinActive[P[i].TimeBin])
      {
#endif
#if GADGET_HYDRO == HYDRO_PESPH
        SphP[i].Density = SphP[i].NumDens*P[i].Mass;
#elif GADGET_HYDRO == HYDRO_MFM
        SphP[i].Density = SphP[i].NumDens*P[i].Mass;
#endif
    if(SphP[i].Density > 0)
      {
#if GADGET_HYDRO == HYDRO_SPH
      SphP[i].DhsmlDensityFactor *= P[i].Hsml / (NUMDIMS * SphP[i].Density);
      if(SphP[i].DhsmlDensityFactor > -0.9)
        SphP[i].DhsmlDensityFactor = 1 / (1 + SphP[i].DhsmlDensityFactor);
      else
        SphP[i].DhsmlDensityFactor = 1;
#elif GADGET_HYDRO == HYDRO_PESPH
      SphP[i].DhsmlPressureFactor *= P[i].Hsml / (NUMDIMS * SphP[i].NumDens);
      SphP[i].DhsmlNumDensFactor *= P[i].Hsml / (NUMDIMS * SphP[i].NumDens);
      if(SphP[i].DhsmlNumDensFactor > -0.9)
        SphP[i].DhsmlNumDensFactor = 1 / (1 + SphP[i].DhsmlNumDensFactor);
      else
        SphP[i].DhsmlNumDensFactor = 1;
#elif GADGET_HYDRO == HYDRO_MFM
      SphP[i].DhsmlNumDensFactor *= P[i].Hsml / (NUMDIMS * SphP[i].NumDens);
      if(SphP[i].DhsmlNumDensFactor > -0.9)
        SphP[i].DhsmlNumDensFactor = 1 / (1 + SphP[i].DhsmlNumDensFactor);
      else
        SphP[i].DhsmlNumDensFactor = 1;
#endif

...
```