

OPENFPM: A SCALABLE ENVIRONMENT FOR PARTICLE AND PARTICLE-MESH CODES ON PARALLEL COMPUTERS

A dissertation submitted to
TECHNISCHE UNIVERSITÄT DRESDEN
FAKULTÄT INFORMATIK

to attain the degree of

DOCTOR RERUM NATURALIUM
(DR. RER. NAT.)

presented by

M.Sc. Pietro Incardona
born on 30.01.1983 in Genova, Italy



Technische Universität Dresden
Max Planck Institute of Molecular Cell Biology and Genetics
Center for Systems Biology Dresden

Dresden, 2021

Part I

Abstract

(Abstract re-adapted from: OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers. *Comput. Phys. Commun.*, 241:155– 177, 2019.)

Scalable and efficient numerical simulations continue to gain importance, as computation is firmly established tool of discovery, together with theory and experiment. Meanwhile, the performance of computing hardware grows with increasing heterogeneous hardware, enabling simulations of ever more complex models. However, efficiently implementing scalable codes on heterogeneous, distributed hardware systems becomes the bottleneck. This bottleneck can be alleviated by intermediate software layers that provide higher-level abstractions closer to the problem domain, hence allowing the computational scientist to focus on the simulation. Here, we present OpenFPM, an open and scalable framework that provides an abstraction layer for numerical simulations using particles and/or meshes. OpenFPM provides transparent and scalable infrastructure for shared-memory and distributed-memory implementations of particles-only and hybrid particle-mesh simulations of both discrete and continuous models, as well as non-simulation codes. This infrastructure is complemented with frequently used numerical routines, as well as interfaces to third-party libraries. This thesis will present the architecture and design of OpenFPM, detail the underlying abstractions, and benchmark the framework in applications ranging from Smoothed-Particle Hydrodynamics (SPH) to Molecular Dynamics (MD), Discrete Element Methods (DEM), Vortex Methods, stencil codes, high-dimensional Monte Carlo sampling (CMA-ES), and Reaction-Diffusion solvers, comparing it to the current state of the art and existing software frameworks.

Part II

Acknowledgment

I truly believe that the person I am now come from meeting persons that directly or indirectly left important pieces of teaching and experiences in my life. In this thesis I want to mention the persons that contributed in who I am and left something I will continue to carry for the future of my life.

I first want to acknowledge my supervisor Ivo Sbalzarini to give me the possibility to do this thesis in his group. I started my experience in MPI-CBG together with him, and through out this long journey of more than 9 years he gave me the thrust to work in his group, and satisfaction to reach the goal of a PhD thesis.

I also want to thanks Rajesh Ramaswamy for pushing me to start this journey, despite my initial skepticism.

I also want to thanks Abhinav in making me believe more in DC-PSE. Him sharing his work enlarged my knowledge in numerical methods, drove the development of the expression in C++ to solve PDEs. With him I shared my interest in PC parts, hardware and gaming.

Surya for the long walks along the Elbe and bikes to Pirna in which many of the discussion about life, future and prospective, gave me courage to take certain decisions.

I want to thanks Justina, she indirectly introduced me to the world of running, and indirectly pushed me in finding alternatives goals. I also want to thanks her for her contribution to OpenFPM for the numerics on level-set.

Aryamann to be the person with which you can talk about everything, ask honest opinions and keep me with the feet on the ground. He value friendship most and I value his friendship most. I also thanks him for spending a lot of time in proofreading this thesis.

Sachin for teaching me how to do numerics on level-set in an efficient and elegant way. I deeply respect his humbleness and definitively something in which I reflect myself.

Nandu Gopan, despite our life crossed for only six months sharing the same office, discussion with him open up new prospective. I also thanks him for spending a lot of time in proofreading this thesis.

Stefan Adami for being the first external person to believe on this project and use it. The satisfaction of having people to use what I created is always rewarding and key component to motivate what I am doing. Additionally discussions with him and his students sparkle novel directions and challenges in which develop OpenFPM.

I also thanks Yuah Gong and Omer Demieriel. They have been the first two persons I meet when I started in the MOSAIC lab. They introduce me and they have been the initial guide to this fantastic and always evolving journey.

All the MOSAIC group, for the nice atmosphere during all these years. In some way everyone has left something important, but I wanted to mention the one than has a slithly more special place in this journey.

Part III

Published works

This thesis partially presents work that has been published in the following papers:

- P. Incardona, A. Leo, Y. Zaluzhnyi, R. Ramaswamy, and I. F. Sbalzarini. **OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers.** *Comput. Phys. Commun.*, 241:155–177, 2019.
- P. Incardona, T. Bianucci, and I. F. Sbalzarini. **Distributed sparse block grids on GPUs.** In *Proc. International Conference on High Performance Computing (ISC)*, volume 12728 of *Lecture Notes in Computer Science*, pages 272–290, Cham, Switzerland, 2021. Springer.
- A. Singh, P. Incardona, and I. F. Sbalzarini. **A C++ expression system for partial differential equations enables generic simulations of biological hydrodynamics.** *Eur. Phys. J. E*, 44:117, 2021. (Publisher Download, PDF (Open Access))

Chapter 1

Introduction

1.1 Motivation

Computer and numerical simulations are at the base of studies to understand complex physical phenomena, described through equations or more in general mathematical models. With the rapid explosion of computational power provided by modern many-core CPUs and GPUs, numerical simulations became more valuable. This increase of power came at the price of new complex programming paradigms and techniques to take advantage of such new computational devices.

Because of these ever-growing new paradigms and techniques, it is of considerable interest to understand how we can construct and design environments able to separate expertises. In particular, while internally dealing with the hardware's complexity, this environment must offer a simplified way to program numerical methods or algorithms without or with minimal knowledge of hardware details. Abstract hardware details are fundamental because, as we have seen in the last two decades, knowledge to program on modern heterogeneous supercomputing platforms efficiently becomes more complex with time. New and continuously evolving hardware, computer systems, and platforms require many years of experience found only in a small group of people [1].

To address such a problem, a design able to abstract and hide the details of the wide variety of hardware, computer systems, and platforms becomes vital. Creating abstractions able to couple generality and flexibility with performance is challenging. It is well known in high performance computing field that, having abstraction more general and flexible reduces the possibility of doing certain types of optimization. Most of the time, optimizations work in a narrow set of cases imposing restrictions and code specialization. For this reason, a big challenge and effort reside in providing an environment where abstractions work orthogonality to performances.

Many attempt has been made and approaches have been extensively stud-

ied in order to achieve this goal. These are typically provided as *programming language extensions*, *high-level programming languages*, *software libraries*, or *framework*. The most basic level to abstract hardware is a programming language and a compiler. For example, it has been shown that a programming languages, and compilers can abstract many architectures. A compiler translates an architecture-independent file into a binary for multiple CPUs architectures likes: x86, AArch64, ARM, Itanium, Lanai, MIPS, PowerPC, RISC-V, SPARC, and accelerators like Nvidia and AMD GPUs. More recently, architecture independent languages like C/C++ has also been used to drive FPGA development.

While programming languages have been used successfully to abstract the architecture, they provide very little for parallelization paradigms needed for multi/many-core CPUs or massively parallel architectures like GPUs. Numerous programming language extensions and libraries have been introduced to fill this gap. Among them it is possible to cite OpenACC [2] and OpenMP [3] that provide a directive-based parallel programming model, CUDA [4] and OpenCL [5] for GPGPU and accelerator programming, and High-Performance Fortran (HPF) [6]. Examples of higher-level programming languages for parallel computing include Linda [7], providing a model for coordination and communication between parallel processes, Vectoral [8] for direct vector-processor programming. Julia [9] designed for high-performance numerical analysis, Rust, a multi-paradigm programming language, focused on performance and safety [10]. While general-purpose programming languages with their extensions in modern days cover multi/many-core CPUs and different types of accelerator, for the most they does not cover distributed computing.

Many other libraries and computational paradigms have been created and studied deeply to fill this gap for distributed computing. We can find implementations for the Message Passing Interface (MPI) [11] like OpenMPI [12] and MPICH [13], as well as runtime systems for parallel and distributed applications like HPX [14] StarPU [15] and Legion [16], or various implementation of Partitioned global Address Space (PGAS) models like co-array Fortran (CAF) [17], Unified Parallel C (UPC) [18], SHMEM [19], DASH [20] and NVSHMEM[21]. In contrast, as an example of a framework for distributed parallel programming, we have Charm++ [22]. These libraries generally focus on a generic programming style in which the user has to take care of decomposing data and computation explicitly.

Toward fully abstract the data decomposition and computation, there is an enormous number of specialized libraries and solvers for problem-specific cases. In molecular dynamic, there is LAMMPS [23], Gromacs[24], NAMD [25], HOOMD-blue [26], and Espresso[27]. In the case of SPH-only solvers coupled with DEM there is DualSPH [28], and GPUSPH[29]. In Finite Elements Methods (FEM) libraries and environments, there is Trilinos[30], DUNE[31], and Deal. II[32]. For DEM, there is Rocky-DEM[33] with its GUI/Workbench. For more adaptive mesh resolution cases, there is the SAMRAI[34] library and AMREx [35] framework. For Finite Volume Methods (FVM) and stencil-based code there is OpenFOAM [36] and LibGeoDecomp [37].

Beyond HPC, and to a higher level than libraries, several languages and problem-solving environments [38] for simulation exist that focus on sequential processing, including the equation-based simulation language Modelica [39], a Matlab-based compiler [40], and the scientific computing environment FALCON [40]. In the area of multi-physics solver environments, we can find ANSYS [41] that provide a DSL and a GUI/workbench to solve partial derivatives equations as well as COMSOL [42].

The libraries and environment cited before reduce or, in particular cases, eliminate the need for code development, narrowing down the problem to a more specific one. LAMMPS, for example, provides a simple DSL to set up a molecular dynamics simulation with few lines of coding in their DSL before running it. GROMACS narrows the problem to protein folding in water, providing a simple interface based on configuration files, and eliminating completely the need for coding. However, there is a trade-off between flexibility and simplicity to use. While the first provides flexibility and a steep learning curve, the second provides less flexibility but a gentle learning curve. Choosing a general-purpose language gives even more flexibility, but the learning curve can become even steeper.

This thesis will explore the idea of constructing a library called Open Framework for Particle Mesh (OpenFPM) for distributed particle-only, mesh-only, and particle-mesh methods without specializing to a particular numerical method. The thesis focus on the goal of providing distributed data structures for simulation in N -dimensional space with $N > 3$. Such a library has to work on a broad type of hardware, accessing low-level hardware functionality like GPUs, and communication libraries to connect computational nodes, mainly written in C/C++. This mostly force the choice of the programming language for OpenFPM to C/C++.

Despite its design not being theoretically bound to a particular language, all the internal layers presented in the thesis are written in C++ 2011 for the most, and 2014 in few cases. The C++ template meta-language engine is useful to do source code generation at compile-time and as already mentioned the availability of high-performance computing tools written in C++ made it suitable to develop the library's lower level layers. Although C++ is a very complex programming language, simplicity of coding, even at a high level, can be achieved by restricting the OpenFPM API to expose features that require knowledge of only a subset of the C++ language. For example, OpenFPM API avoids forcing users to use pointers, new/delete operations, indirection operators, pointer arithmetic, and general memory management, hence eliminating its intrinsic complexity.

Another way to reduce the complexity in a general-purpose language is to construct a higher level abstraction that narrows the problem to a more specific case where it is possible to expose few degrees of freedom. For example, we could embed an entire particle simulation with n -lines of code into a function exposing only the possibility to change the particle-particle interaction kernel. The only limit resides in the syntax to expose such degrees of freedom and how many it is possible to expose in a reasonably compact form. The choice of what we want to hide and what we want to expose reside on how we design our abstraction.

However, the capabilities of the base language to redefine operators and create specific constructs are fundamental to create such abstractions. C++ up to now remain one of the most flexible languages reinforcing our choice as base language.

If abstractions are constructed vertically like in figure 1.1, the design will benefit from having fewer dependencies between layers. Ideally, the change of one layer should at the most influence the layers up. OpenFPM pursues this direction, creating a hierarchical level of abstractions bottom-up, starting with the multi-hardware memory management to distributed data structures on multi-hardware. On top of them, we have numerical methods and higher abstractions, like expression-based computation and generic PDE expression-based computation. The entire development stack remains mainly in the C++ language. Although some top-level development like generic PDE solvers could be done in a higher-level language like Python or a DSL, currently, the development is still in C++.

In constructing such environment, the thesis use distributed particle only and particle-mesh methods. Such methods are particularly appealing from a software-engineering viewpoint because they require various data structures in interaction with each other. This thesis will show how this requirement creates several challenges in terms of the design of the library.

Many of the libraries already cited for mesh-based simulation like AMREx and Samrai support a hybrid approach, where particles works as an extension of the mesh-based structure. Others like LAMMPS,FDPS and many of the particles only based libraries cited, support some mesh computation as an extension to support long-range interactions. While few libraries like POOMA [43], and the Parallel Particle Mesh (PPM) library [44, 45] with its domain-specific Parallel Particle Mesh Language (PPML) [46, 47] support both particles-only and mesh-only within the same library without having a privileged base data-structure. While these libraries have successfully provided abstractions for rapid development of scalable parallel implementations of particle and particle-mesh methods, both seems to be discontinued. Additionally, these libraries do not support dimensionality bigger than 3, modern accelerators and complex properties.

1.2 Contribution

Many separated libraries exist able to handle particle-only particle-mesh and mesh-only simulations. To the best of our knowledge, none of them expose general data structures supporting CPU and GPUs natively at the same time for particle-only mesh-only and particle-mesh methods. The level of flexibility given by the design and abstractions of OpenFPM cannot be found in any other library. None of the libraries can do N-dimension or have particle or mesh nodes with an unlimited number of variables, where each variable can also be a nested structure. We will see how by design, OpenFPM gives the possibility to construct data-structures with natively the feature of switchable layout arrays,

such as array of struct (AoS) and struct of array (SoA) or reshape memory coming from an arbitrary source. Another characteristic not seen in others libraries is having distributed data structures where the layout switching is not limited to AoS or SoA or fundamental types like floating-point or integers. In OpenFPM, the C++ type extensions like array the operator `[]` is recognized and supported by the layout switching mechanism. In particular, an array of `double[4]` or, in general a compile-time M-dimensional matrix `double[4]...[6]` can be re-arranged in memory at the granularity of the single components. This thesis will show how such a mechanism by design is transparent to the concept of being a double an int or a general type.

In addition to compile-time matrixes and general types, OpenFPM supports general N-dimensional distributed data structures where N can be bigger than 3. The possibility to have ghost areas with unlimited size and disentangled from the space decomposition. Where ghost areas are intended as the area overlapping two or more processors like in figure 3.2. This feature makes cases like having a grid 2×2 and a ghost of 1 with 1024 processors possible in OpenFPM. The result will be that four processors will have one point each, and the other processors will be able to access reading or writing by the ghost area. The final result is 4 points shared across all processors to exchange information or implement tree-based long-range reductions without employing or developing unique data structures for such tasks.

In addition to unlimited ghost areas, another novel point of OpenFPM is its design of having distributed templated containers that automatically make distributed shared memory or non-distributed containers. In order to do so, the non-distributed containers must respect a specific interface; such design open-up to the possibility to use the distributed containers to external non-distributed containers with different implementations. With this feature, it is possible to potentially distribute external library containers that were not designed to be distributed, with an adapter interface.

Another essential feature of OpenFPM is to support CPU and GPU natively. In this regard, some libraries like AMREx recently also started to provide multi-hardware data structures on GPU. To the best of our knowledge, this is the only library to support such a feature. In particular, the thesis will show how OpenFPM has been designed to have low-level modules specialization to handle multiple types of hardware. The thesis will show how many components has been modularized in order to reduce hardware specialization to fewer modules. As an example it will show how to code one distributed mesh structure able to handle uniform grid and sparse grids with few changes.

Another contribution resides in providing a successor to the discontinued PPM Library [44, 45], based on this past success. In showing its design, this thesis will explain and use advanced scientific software design and engineering techniques. This thesis will extend the capabilities by adding transparent dynamic load balancing, support for accelerator hardware, and automatic memory layout optimization in comparison to PPM. Using OpenFPM is further facilitated by complete and up-to-date documentation and a series of tutorial videos and example codes. It is actively supported in the long term, with new func-

tionality continuously being added.

1.3 Particle Methods

Particle methods give a unified framework able to simulate discrete and continuous models. In the discrete case, particles model real physical entities like atoms in molecular dynamic simulations or granules in the discrete element method. A particle can also represent complex entities like an animal, human, or car in agent-based simulation. In the continuous case, particles represent, combined with their kernel function, mathematical collocation points. The set of points represent the finite-dimensional space of functions. In both cases, a particle p has position in space $\mathbf{x}_p \in \mathbb{R}^n$ and properties \mathbf{w}_p where each property can have a different data types.

Particles can interact with each other and move in space. Several particle method interactions are limited to be pairwise and restricted to be additive, ensuring that the overall result is independent of the particle indexing order. Once the interactions are computed, particles evolve their positions and properties according to pre-defined rules. The update rules can come from a cellular automaton, or from the discretization of continuous differential operators like integration schemes.

In formulas, particle methods restricted to pairwise interaction can be written as

$$\frac{d\mathbf{x}_p}{dt} = \sum_{q=1}^{N(t)} \mathbf{K}(\mathbf{x}_p, \mathbf{x}_q, \mathbf{w}_p, \mathbf{w}_q) \quad (1.1)$$

$$\frac{d\mathbf{w}_p}{dt} = \sum_{q=1}^{N(t)} \mathbf{F}(\mathbf{x}_p, \mathbf{x}_q, \mathbf{w}_p, \mathbf{w}_q) \quad (1.2)$$

Where $N(t)$ indicate the total number of particles at time t , \mathbf{K} is the interaction kernel \mathbf{K} and \mathbf{F} contain the evolution rules of the model being simulated or the numerical method used. During a simulation, the number of particles can change. In particular, to implement adaptive resolution methods, we have to add or remove particles locally.

In the above equation, the summation extends to all particles; this leads to a computational cost of $O(N^2)$ where N is the number of particles. In many cases, the interaction kernel is local or compact. In these cases, it is possible to use efficient algorithms to retrieve neighborhoods like cell lists or Verlet lists [48]. Such algorithms are able to reduce the computational cost to $O(N)$ on average, if the density of the particles remains constant. When the kernel is non-local, the interaction is decomposed into short- and long-range components. While the short-range component is evaluated using Verlet and cell-list algorithms, the long-range interactions are evaluated on a uniform Cartesian background mesh [49], like remeshed vortex methods for solving the incompressible Navier-Stokes equations [50]. Additionally for long-range interaction approximation

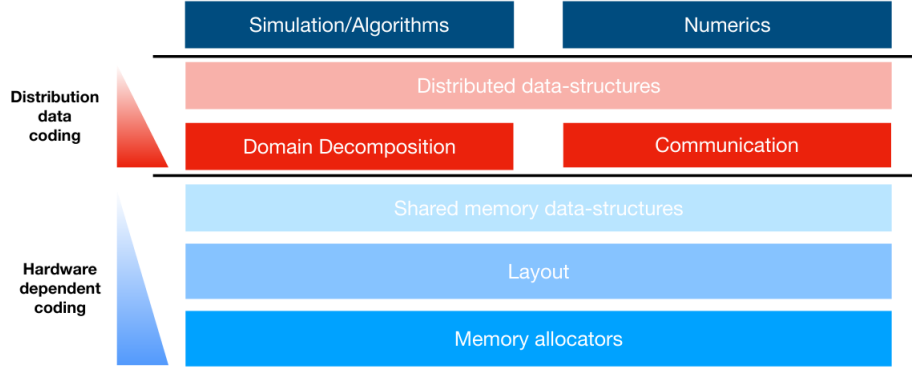


Figure 1.1: Diagram of the OpenFPM software stack. The top-level modules in the dark blue provide scalable numerical algorithms implemented using the transparently distributed data structure of the lower level. The distributed data structures are implemented based on efficient single-core data structures through a domain-decomposition and inter-processor-communication layer. The memory layout of the single-core data structures is parametrically decided at compile-time and managed at runtime by the lowest layer: the memory allocators.

algorithms, there are Barnes-Hut [51], and fast multipole methods [52] and the Ewald method for including electrostatic interactions in molecular dynamic simulations [53]. Hybrid particle-mesh methods allow each computational step to be performed in the best-suited formulation, moment-conserving particle-mesh and mesh-particle interpolation are used to translate between the two discretizations [54].

1.4 The OpenFPM Library

As stated in the introduction, the OpenFPM project aims to build an open-source software library to implement scalable particle and hybrid particle-mesh simulations on shared-memory and distributed-memory parallel computer systems. OpenFPM is written entirely in C++, and at the user level, provides a scalable infrastructure to implement custom particle and particle-mesh codes. In figure 1.1 the layers below in red and light red mainly feature methods for domain decomposition, dynamic load balancing, communication abstractions, checkpoint/restart, asymmetric and symmetric (i.e., “action-reaction law”) interactions, and iterators for particles and meshes.

Under the red layers, there are efficient single-core data structures (light blue layers), which rely on compile-time memory layout module and runtime memory allocators. On top of the distributed data structures, this infrastructure is complemented by a set of frequently used numerical solvers indicated in the dark blue (Numerics). Typically these solvers are built on top of other numerical

libraries like PetSc [55] and Eigen [56], for linear algebra and matrix inversion.

One step below the numeric layer, the user level is built on top of the distributed data structures. OpenFPM provides the flexibility through templated parametric data types. A data structure parameter can define the dimensionality of the space or the floating-point precision used. On the other hand, it can also add or modify the internal code of the data structure. The thesis will show how this mechanism is used in OpenFPM to design layers to fit very different requirements, like those coming from supporting multiple types of hardware. We will see how designing each layer, separating each concern, and stacking them together to create flexible code that can fit different requirements using most possible compact code.

OpenFPM, in addition to provides a modular and compact code, also aims to give a public available long-term supported tool. For this reason, OpenFPM is developed under test-driven development concept and code quality assurance. The code is checked every commit by the continuous integration, and overnight the code is automatically tested on a local HPC cluster with a different number of processes, nodes, and hardwares (AMD CPU, ARM CPU, Nvidia GPUs), with different compilers (clang, gcc, intel), MPI implementations (OpenMPI, MPICH), and different debugging tools (libsanitizer, Valgrind, Cuda-memcheck). Static analyzers, coverage reports, and automated performance test graphs give a reference if a commit introduces a coverage or performance regression.

The installer/CMake scripts are also tested overnight cross-platform on different Linux distributions (Mint, OpenSuse, Fedora, Debian, CentOS, Ubuntu) and OS (Windows with MSYS2 and macOS) to guarantee a working cross-platform installation process. The docker images with OpenFPM installed, generated during testing are provided to the user as an alternative way to use the OpenFPM library. We will see in the last chapter of the thesis the OpenFPM build and release pipeline.

1.5 Design of the OpenFPM library

In this section, is going to explain the design of the OpenFPM library. Figure 1.1 shows an overview of the OpenFPM design at a global level. These modules are grouped into six repositories. Memory allocation stays in `openfpm_devices`. Layout and single-core data structures stay in `openfpm_data`. Communications stay in `openfpm_vcluster`, Domain decomposition and Distributed data-structures live in `openfpm_pdata` and Numerics live in `openfpm_numerics`. There is an additional module called `openfpm_io` that lives external to this vertical design, and its purpose is to provide functionality to write data to VTK, HDF5, OpenPMD [57] files, and other output formats. Each repo has its tests and can run independently from the repo stacked above.

One advantage of having different repositories is that each functionality can be isolated, and the OpenFPM project can be redistributed with only a subset of functionalities. For example, when we need only the memory allocation, we use

only `openfpm_devices`. If we want single-core data structures on CPU and GPU without any further functionality, we use `openfpm_devices` + `openfpm_data`. If we also want to write data, we can redistribute the OpenFPM project to use `openfpm_devices` + `openfpm_data` + `openfpm_io`. If we go parallel and we need to communicate data, we can repack `openfpm_devices` + `openfpm_data` + `openfpm_io` + `openfpm_vcluster`. If we also want the distributed data structure, we incrementally add `openfpm_pdata`, and the same reasoning goes for `numerics` and `openfpm_numerics`.

Repacking the project with a smaller number of modules/repositories reduces the number of dependencies and consequently simplifies the installation process. A table of the external dependencies for each of the repositories can be found in table 1.5

openfpm_io	Boost,HDF5, ADIOS2, OpenPMD
openfpm_numerics	Boost,PETSc,SuiteSparse,Algoim,Blitz,Eigen, OpenBLAS
openfpm_pdata	Boost, HDF5, Metis, ParMetis, LibHilbert, MPI
openfpm_vcluster	Boost, MPI
openfpm_data	Boost, Vc
openfpm_devices	Boost,

Repacking the project means that OpenFPM can be used in non-parallel projects without requiring MPI dependencies or the complete set of dependencies. In particular, separate repositories are convenient if a project does need or require parallelization across multiple nodes.

Another reason to use separated repositories is the possibility to enforce a particular design and modularity. For example, complex data-structures require memory objects to allocate memory, but memory allocations do not require layout concepts or complex data structures to allocate memory. With separate repositories and separate tests, it is possible to enforce the memory allocation objects to work in a self-contained way without any need for concepts defined in the upper modules. With a single repository, we do not have a mechanism to restrict what we are allowed to include. It would be easy to fall into the case where we have nested inclusions that spread across the entire project violating the software stack design in figure 1.1.

Chapter 2

Hardware independent data-structures

The following subsections will explore the three lowest modules: memory allocation, memory layout, and single-core data structures. These layers are shown as the three blue layers in figure 1.1. This chapter outlines how the memory allocation module creates memory buffers for different types of hardware. The memory layout module and the single-core data structures have the duty to shape and give meaning to the allocated memory.

2.1 Memory Allocation

As previously mentioned, OpenFPM, at the base of its design, has memory allocators. Memory allocators are objects that expose a common interface to create memory. Internally each of this object implement different memory hardware sources, alignment or organization of the allocated memory.

The standard C++, define to have allocator objects that allocate memory and return pointers leaving the duty to deallocate them at the end manually. In comparison, OpenFPM allocates memory and wraps it around objects, making the lifetime of the memory linked to the object's lifetime. This design has the advantage of being easier to maintain and less error-prone to memory leaks because the memory object automatically performs deallocation. Memory destruction is automatically managed when the memory object goes out of scope.

In few scenarios, having a memory object imposes some limits. One is when there are memory objects shared across resources. Although attempts are made to avoid such conditions whenever possible, they cannot be avoided entirely. For this reason, all the memory objects expose an interface for explicit reference counting. The reference counter can be increased, decreased, or queried to understand if the memory is shared and indicate how often this memory chunk is referenced. Explicit destruction of the memory and linked memory object happens only if the reference counter reaches zero.

Memory objects also introduce the concept of single and double-buffering. Double buffering means that one buffer exists on the host and the other exists on the device, where the device can be an accelerator like a GPU or even the system memory. In the case of single buffering, there is only one buffer, either device or host, but both pointers point to the same buffer. The possibility of using a device buffer on the host or a host buffer on the device depends on the limitations imposed by the hardware vendor. In CUDA, host pointers and device pointers can be created with unified virtual memory, making them accessible by both CPU and GPU hardware.

Here is shown the hardware-independent memory interface the objects must follow to be valid memory allocators for OpenFPM.

Listing 2.1: "Interface of a memory allocator object"

```

1
2 bool flush()
3 bool allocate(size_t sz)
4 bool resize(size_t sz)
5 void destroy()
6 bool copy(const memory & m)
7 size_t size()
8 void * getPointer()
9 void incRef()
10 void decRef()
11 long int ref()
12 bool isInitialized()
13 void * getDevicePointer()
14 void deviceToHost()
15 void hostToDevice()
16 void hostToDevice(size_t start, size_t stop)
17 void deviceToHost(size_t start, size_t stop)
18 void fill(unsigned char c)

```

The most important functions are **allocate**, **getPointer**, **getDevicePointer**. Respectively they allocate memory, retrieve the pointer either on host (**getPointer**) or device (**getDevicePointer**). The function **destroy** explicitly delete the internal memory. The call to this function is optional but useful to deallocate the internal memory while the memory object is still alive; in all the other cases, the memory is automatically deallocated when the object goes out of scope.

In order to facilitate deallocation in the case of shared memory resource, the functions **incRef**, **decRef**, and **ref** are methods to increase/decrease and get the reference counter to know how many resources are using this memory object. The function **size** returns the size in bytes of the internal memory allocated, and the **resize** function makes the previously allocated memory bigger or smaller. A resize function at a low level allows us to use virtual memory addressing to remap new memory consecutively to the already allocated memory without physically moving it.

Out of resizing memory at this level, as already mentioned, memory allocators introduce the abstraction of double buffering. Because of double buffering, two methods to transfer memory from device to host **deviceToHost** and from host to device **hostToDevice** are provided. It worth noticing that in single buffer objects like **HeapMemory**, **getPointer()** and **getDevicePointer()**

return the same pointer, and the function **deviceToHost** and **hostToDevice** does not perform any operation. Such characteristic gives the option to avoid a full implementation of double buffering, despite being forced to expose a double buffering interface.

Memory objects also hide the mechanism of on-demand allocatio. The allocate function does not actively allocate host or device memory until **getPointer** or **getDevicePointer** is called to return the pointer explicitly to the allocated memory. This feature allows the creation of device-only or host-only memory even when double-buffering memory objects are used.

The type of object identifies the type of memory and the policy adopted like single or double-buffering. The freedom to implement the memory interface gives the possibility to have an object like **HeapMemory** to allocate system memory from the heap in single buffering or **CudaMemory** for NVIDIA/AMD GPU memory in double buffering. This freedom also gives the possibility to construct objects with particular policies. One such object is **PtrMemory** which can wrap a memory object around externally allocated memory. Its role becomes relevant when we want to re-shape preallocated and filled memory. **PtrMemory** can be used to force a high-level data structure to absorb external raw memory and re-shape it to a multi-dimensional array. In the case of **PtrMemory**, the resize function is limited to accept a size smaller or equal to the size of the external buffer. New classes can be created to construct new memory allocation policies.

Among particular allocation policies classes there is ExtPreAlloc. This memory object can be seen as a wrapper that maps a preallocated memory pool into a subsequent series of **allocate()** method calls. For example, we can allocate 200 bytes of memory and call for the first time the method **allocate** with size 100. The subsequent **getPointer()** method will return a pointer to the first 100 bytes. A second call to **allocate** with size 100 and subsequent call to the function **getPointer()** will return a pointer to the second 100 bytes. This type of functionality is beneficial if we want to have two distinct data structures contiguous in memory. This feature is important to serialize multiple data structures into a contiguous array to be sent over the network. Consider, for example, constructing a packet containing information about a collection of multiples data structures. Such a mechanism enables the possibility to remap multiple data structures over a contiguous portion of allocated memory ready to be sent over the network. Memory policies can also be combined with internal debugging features.

HeapMemory and CudaMemory for debugging purpose can initialize the allocated host and device memory with all bits flipped to one when the pre-processor macro GARBAGE_INJECTOR is enabled. This debugging policy because initialize floats to nan and integer to their maximum values, helps in locating errors generated by reading uninitialized data.

2.2 Shared memory data structures and layout

The previous section introduced general objects providing a standard interface to allocate memory from different sources like heap memory or Cuda GPU memory. Organizing data in OpenFPM is always done with an increasing set of higher-level data structures. Because data require memory, data structures are built on top of memory objects presented before.

Some of the features from memory allocators, like double buffering, are inherited and exposed by design, while others remain hidden. This thesis will show how features like double buffering are inherited and exposed across all openfpm layers, up to the user level of the distributed data structures.

In OpenFPM, data structures are implemented as a composition of layers; where each layer introduces a feature to organize data. We will see how these layers can be reused across multiple data structures and how the implementation of one layer does not interfere with the others. We will see that some maps determine complexity in accessing or performance like cache friendliness with specific access patterns, while others define the structure of the container.

The first layer converts multi-indexes into one-dimensional linear memory. The thesis will show how additional mapping layers can be combined to create increasingly more complex data structures.

2.2.1 Multi-dimensional array of primitives

The first layer we are going to present is the multi-dimensional array. In particular, the layer takes as an argument a multi-index and transforms it into an address in memory. In the code 2.2 we construct a multi-dimensional array map in openfpm, and the following parameters characterize it

- **dimensionality** or how many indexes we have in our multi-dimensional array.
- **type** Type stored by the multi-dimensional array. For example, integer or float if we want to construct a multi-dimensional array of floats or integer.
- **test_mem**: While in general a multi-dimensional array internally allocate memory on heap, stack or other types of memory. Here our multidimensional array can be seen as a pure map and accept external memory that has been pre-allocated and is big enough to store all the elements.
- **type of index** The type of the index indicates if an index is runtime `boost::mpl::int<-1>` or known at compile-time `boost::mpl::int<3>` or `boost::mpl::int<7>`
- **order** The order of the indexes indicates how the indexes are ordered in the array. This parameter controls the stride units of the indexes. Consider for example the case of `float[3][10][7]` the index [7] has normally stride unit 1 while the index [10] has stride unit 7 and the index [3] has $7 \times 10 = 70$. This order is indicated as 0,1,2. We could reorder the indexes as 2,1,0 like

in Fortran. In this case, the stride units would be 1 for [3] and 3 for the index [10], and 30 for the index [7].

Listing 2.2: Code to construct a multi-dimensional array

```

1  openfpm::general_storage_order<3> so = openfpm::ofp_storage_order();
2
3
4  openfpm::multi_array_ref_openfpm<float,
5      3,
6      boost::mpl::vector<
7          boost::mpl::int_<-1>,
8          boost::mpl::int_<3>,
9          boost::mpl::int_<7>>
10 ar((float *)test_mem,10,so);
11
12 ar[5][2][5] = 8.0;
13 ar[9][2][6] = 4.0;
```

The code 2.2 define a multidimensional array of floats with sizes [10][3][7] in which [10] is determined at runtime. Note that the number for the runtime dimension is given in the constructor and not as a template parameter. Because this structure can be seen as a view over the memory we pass in construction, it does not allocate memory, but shapes preallocated memory. The constructor's first parameter is a memory pointer (test_mem) big enough to contain all elements. Despite its complex definition, the multidimensional array can be used easily with the operator[].

There is no limitation on which type of memory the pointer test_mem is pointing to, it can be memory allocated either on heap,gpu,stack or any other valid memory type. In order to guarantee this structure works on accelerators, it has been tested to compile and run on different specialized accelerator compilers like Nvidia Cuda for Nvidia GPUs, and AMD hip-clang for AMD GPUs. The implementation is based and inspired by a heavily modified boost::multi::array version. There is a distinct separation between indices known at compile-time and those known at runtime, which was not present in the boost original version. The implementation has been also simplified and removed of features that impact performance.

The structure presented here map multi-indexes and accept primitives like floating-point types or integers of any size. It can also work with general objects like a C++ struct but cannot remap the internal elements of a struct. In order to achieve the flexibility to remap the elements of a struct, we should, in theory, be able to parse the struct and get all its elements. Unfortunately, this is not possible in standard C++. The following section will see how this feature can be achieved when structures are expressed as tuples and the equivalent OpenFPM wrapper aggregate.

2.2.2 Tuples as compile-time parse-able structures or aggregates

A tuple can be seen as a fixed-size collection of heterogeneous values. This definition also fits the concept of a simple C++ struct. From C++ standard 2011,

the language officially introduces the variadic template syntax. This construct can be used to express a variable list of heterogeneous types. The C++ standard library provided an implementation called `std::tuple`, but because `boost::fusion` provides an implementation with more functionalities at the beginning of this project, we mainly wrapped around the boost implementation. Across this entire thesis, we will see how in general, OpenFPM try not to expose a library directly, and always wrap its functionality in order to have a more convenient way to replace the implementation if necessary. For this reason, in OpenFPM, there are aggregates as equivalent to tuples. Aggregates are implemented around `boost::fusion` and work in a very similar manner to tuples. If we want to have a structure that contains, for example, a scalar and a static array of 3 elements, we write

```
1 aggregate<double,double[3]> data;
```

We can access each element with its positional number with `at_c`

```
1 at_c<0>(data) = 1.0
2
3 at_c<1>(data)[0] = 1.0
4 at_c<1>(data)[1] = 2.0
5 at_c<1>(data)[2] = 3.0
```

or if we define some constant expression

```
1
2 constexpr int scalar = 0;
3 constexpr int vector = 1;
4
5 at_c<scalar>(data) = 1.0
6
7 at_c<vector>(data)[0] = 1.0
8 at_c<vector>(data)[1] = 2.0
9 at_c<vector>(data)[2] = 3.0
```

The code presented above with tuples can be written in an equivalent way with a struct

```
1 struct Data
2 {
3     double scalar;
4     double vector[3];
5 };
6
7 Data data;
8
9 data.scalar = 1.0
10
11 data.vector[0] = 1.0
12 data.vector[1] = 2.0
13 data.vector[2] = 3.0
```

The first form is parseable at compile-time, while the second form is slightly more readable at the cost of being non-parseable. In particular, using the tuples, we can construct algorithms that transform a tuple into another tuple, this property open to the possibility to transform a struct into a different struct for layout or alignment. Suppose we want to construct a transformation of an aggregate, where all the properties must homogenize to an 8 byte equivalent primitive for alignment. More formally construct an algorithm 1.

Algorithm 1 Convert the types into equivalent 8 byte

```

1: for all attribute in aggregate do
2:   if attribute == float then
3:     transform into double
4:   if attribute == int then
5:     transform into long int
6:   else
7:     Do not transform

```

It also possible to produce a version able to transform vectors types like float[3] or matrices like float[4][5], but for the sake of simplicity, algorithm 1 use only scalars. In order to write the algorithm above at compile-time, it is necessary to introduce the C++ template engine that can be seen as a touring complete functional language at compile-time. The thesis in appendix A briefly introduces how this language works and how to map constructs of a standard imperative language into the C++ functional meta-programming language. The meta-code equivalent to algorithm 1 is reported in listing 2.3

Listing 2.3: meta code to tranform a tuple with types into a tuple with types alligned by eight bytes

```

1
2 template<typename T>
3 struct if_float_or_int // No tranform
4 {
5     typedef T value;
6 }
7
8 template<>
9 struct if_float_or_int<float> // transform float to double
10 {
11     typedef double value;
12 }
13
14 template<>
15 struct if_float_or_int<int> // tranform int to long int
16 {
17     typedef long int value;
18 }
19
20 template<typename T>
21 struct align_to_8_byte
22 {};
23
24 struct align_to_8_byte<typename ... props>
25 {
26     typedef aggregate<if_float_or_int<props>...> value;
27 }

```


2.2.3 Toward multidimensional array and aggregates, limits of `openfpm::multi_array_ref_openfpm`

This section will glue together the concepts introduced in 2.2.2 and 2.2.1, and explain how to construct a general multidimensional array working on aggregates (tuples) with a high degree of freedom in layout, ordering, and memory type.

Section 2.2.2 introduced tuples and the C++ template engine as a meta-language to manipulate them. The thesis also presented in section 2.2.1 the first structure, `openfpm::multi_array_ref_openfpm`, that can be used to create a multidimensional array of primitives or objects like a class on external memory. This structure could re-arrange indices and have both static- and runtime-sized arrays or a mixture of them.

While it provides great flexibility and a way to switch array striding, it does not provide any mechanism to do layout switching at compile-time, like array of struct or struct of array. Consider the case where the type for `openfpm::multi_array_ref_openfpm` is a struct defined in listing 2.4

Listing 2.4: standard C structure

```

1
2 struct A
3 {
4     float scalar;
5     double vector[3];
6     int matrix[6][7];
7 };

```

If we now pass struct A to the map `openfpm::multi_array_ref_openfpm`, we will have a multidimensional array of struct A. In order to do a compile-time layout switch into a struct of arrays, we need a method to input structures that are compile-time parse-able. In section 2.1 we have shown how it is possible to achieve this with tuples. The second step would be to extend the previous multidimensional array map to also work on tuples.

Because we want to maintain a component-based system instead of extending the previous code, another additional map layer is created on top of the previous one to maintain the code development vertical and reusable. Two requirements are necessary for this map.

- A parameter indicating the type of layout we want to use.
- A way to access the data-structure independent from the layout chosen.

Consider an example in which we want to construct a multi-dimensional array of size 10×10 of a struct A defined in listing 2.4. As we said before, we have to convert our struct into a parse-able tuple like show in listing 2.5

Listing 2.5: struct A converted into aggregate

```

1
2 constexpr int scalar = 0;
3 constexpr int vector = 1;

```

```

4 constexpr int matrix = 2;
5
6 typedef aggregate<float,double[3],int[6][7]> A;

```

In the case we want an array of structures A, we can create a single `openfpm::multi_array_ref_openfpm`, and pass the aggregate defined in listing 2.5.

Naively, if one tries to implement a struct of arrays based on `openfpm::multi_array_ref_openfpm`, he could create manually three `openfpm::multi_array_ref_openfpm` like in listing 2.6. Although the final goal is to have an automatic way, rather than manual, to construct an SoA object from the tuple.

Listing 2.6: Manual implementation of a SoA

```

1
2 // for the scalar
3 openfpm::multi_array_ref_openfpm<float,
4     1,
5     boost::mpl::vector<
6         boost::mpl::int_<-1>,
7     >
8 ar_scalar(...100...);
9
10 // for the vector
11 openfpm::multi_array_ref_openfpm<double,
12     2,
13     boost::mpl::vector<
14         boost::mpl::int_<-1>,
15         boost::mpl::int_<3>,
16     >
17 ar_vector(...100,so);
18
19 // for the matrix
20 openfpm::multi_array_ref_openfpm<int,
21     3,
22     boost::mpl::vector<
23         boost::mpl::int_<-1>,
24         boost::mpl::int_<6>,
25         boost::mpl::int_<7>,
26     >
27 ar_matrix(...100...);

```

In the manual case shown in listing 2.6 the 10×10 elements has been linearized to 100 and mapped into one runtime index of `openfpm::multi_array_ref_openfpm`, the compile-time indices of vector and matrix are instead specified at compile-time into `openfpm::multi_array_ref_openfpm`.

At this point, someone could have argued why not map the two runtime indexes into two runtime indexes of the multi-dimensional array. The reason reside in the limited number of indexes linearization `openfpm::multi_array_ref_openfpm` provides. Later this thesis will introduce an additional layer on top of `openfpm::multi_array_ref_openfpm`, to provide a more generalized way to linearize multi-indices. For now, we consider that the linearization follows the standard C++ striding linearization.

Reading the manual implementation in listing 2.6 we realize that the algorithms written on top of a manual SoA restructuring depend from the layout chosen. The ideal case would be constructing an abstraction to define a standard API to access data independently from the layout. Before introducing this

solution, we must fuse the map `openfpm::multi_array_ref_openfpm`, that accepts external memory, with allocators, that create a memory. The thesis will explain this in the next section introducing the object `memory_c`.

2.2.4 Introducing memory allocation `memory_c`

The previous sub-section, introduced the object `openfpm::multi_array_ref_openfpm` an object that shapes memory and does not handle memory on its own. This section introduce the class `memory_c` that glues a memory object allocator like `CudaMemory` or `HeapMemory` with the object `openfpm::multi_array_ref_openfpm` that shape memory.

We start observing that the destruction of the object `openfpm::multi_array_ref_openfpm` does not destroy the memory. By design `openfpm::multi_array_ref_openfpm` only map indices to memory, and does not handle its construction, destruction or reallocation.

This separation is convenient for modularity and specific scenarios where we want to shape or reshape memory. On the other hand, many times, we want the memory to live together with its representation (`openfpm::multi_array_ref_openfpm`). For this reason, at this level is introduced the object `memory_c` that encapsulates both a representation and a memory object to allocate memory. This encapsulation guarantees that when `memory_c` is destroyed, both the representation and the memory object resources are deallocated.

Because there is both, memory object and representation, it is possible to add additional functionalities to the object `memory_c` that require memory and representation together. The first functionality involves double buffering, in such case, there is always a device pointer and an host pointer, but the representation can only work on one at a time. For this reason, we create the possibility to switch the representation pointer to device pointer with **`switchToDevicePtr()`** or to host memory with **`switchToHostPtr()`**. This operation does nothing if the memory object does not support double buffering because the two pointers are equal.

Another functionality is the possibility to have two different representations `openfpm::multi_array_ref_openfpm` acting on the same memory. In order to do this, `memory_c` have a method called **`bind_ref`**. Consider the case where there are two `memory_c` objects `m1` and `m2`, and each `memory_c` encapsulates a memory object (`mem`) and a representation (`mem_r`). We can use the function `m2.bind_ref(m1)` to reshape the memory pointed in `m1` with the representation in `m2`. The advantage of using `memory_c` rather than using directly `openfpm::multi_array_ref_openfpm` reside in the automatic memory management. The method **`bind_ref`** makes the `m2` memory object (`m2.mem`) point to `m1` memory object (`m1.mem`) and increment the reference of the `m1` memory object to ensure the memory is released only when both `m1` and `m2` get deleted. Waiting for the reference counter reaching zero, ensures the memory is not referenced by any other `memory_c`, making safe to destroy the memory. As in the method **`switchToDevicePtr`**, **`bind_ref`** can be implemented because `memory_c` glue memory allocation with representation.

In exceptional cases, `memory_c` can accept external memory setting it with the function `setMemory`. If the reference counter of the external memory object is initialized to zero before calling the method `setMemory`, the object will be automatically managed and deleted when `memory_c` is destroyed. If the initial reference counter differs from zero, the memory object will not be automatically managed, and it must be destroyed explicitly. The mechanism of using `setMemory` is essential to force a `memory_c` to shape an already filled external memory. Consider, for example, the case where we receive data stored in a memory object. If we know it has a particular layout, we can fit a representation to it.

The last method is `swap`. Given `m1` and `m2` with the same type of memory and representation, we can swap both the memory and the representation with `swap`. Doing so, `m1` becomes `m2`, and `m2` becomes `m1`. Such functionality is generally helpful to implement a function where the input is changed in-place. While in principle, the input should be overwritten by the result, it can happen that the operation requires a separate buffer to be completed. In this case, internally we have to create another `memory_c` object and use it as an output. In order to convert the output in input, and maintain the in-place interface, as final step, we call `m1.swap(m2)`.

2.2.5 Array of struct (AoS) and Struct of Array (SoA)

This section will introduce another abstraction layer to further evolve our concepts about data ordering and data structuring. One limitation of `multi_array_ref_openfpm` presented in section 2.2.3, is that there is no automatic mechanism to switch from "array of struct" to "struct of array" (SoA/AoS). If we want to do it for a structure like the aggregate `aggregate<float, double[3], int[6][7]>`, until now we have to create three `multi_array_ref_openfpm` manually.

Additionally, general or more complex reordering than indices reordering is not present. The thesis also mentioned that the runtime-index of the structure `multi_array_ref_openfpm` hide more complex indexing strategies, but the thesis did not provide any mechanism to abstract it.

In order to do this, we start introducing a new abstraction layer to control the layout of the data. Controlling the layout requires analyzing the tuple with fully compile-time algorithms. We know from the previous section that creating a compile-time algorithm means creating meta-functions, that in C++ translate in creating types containing the compile-time algorithm. In `openfpm`, there is the `memory_traits_lin` metafunction to implement the AoS restructuring and `memory_traits_inte` to implement the SoA restructuring.

For `memory_traits_lin` the meta-algorithm is given in 2, while for `memory_traits_inte` is given in 3. They both process an aggregate and create an object with respectively AoS or SoA structuring using the object `memory_c` object.

For example when `memory_traits_lin` is chosen, and the aggregate is `aggregate<float, float[3], double[3][4]>` the object in listing 2.7 is constructed at compile-time.

Algorithm 2 memory_traits_lin**Input:** aggregate as list of types **Output:** A memory_c object

- 1: create a type a memory_c <aggregate> where aggregate is the the input aggregate
- 2: return the memory_c type

Listing 2.7: AoS object constructed by memory_traits_lin

```

1
2 memory_c<aggregate<float,float[3],double[3][4]>>

```

We know memory_c<object> is a wrapper to multi_array_openfpm_ref<object> and multi_array_openfpm_ref<object> is, in its simple form, an array. While the aggregate aggregate<float,float[3],double[3][4]> is the equivalent of a struct. Using this reasoning memory_c<aggregate<float,float[3],double[3][4]>> is an array of struct (AoS).

In the case of memory_traits_inte, we construct at compile time a SoA object. Given the tuple aggregate<float,float[3],double[3][4]> the meta-algorithm construct the object in listing 2.8. Listing 2.8 omit several namespaces for readability Int_ is boost::mpl::Int_ and vector is boost::mpl::vector. In this case, we have first an aggregate, so a struct, of memory_c, or array, leading to a struct of array. Because memory_c have control of the array indices, the linearized runtime index is selected to have step-size 1 using the flexibility of the multi_array_ref_openfpm representation specified in listing 2.9.

Algorithm 3 memory_traits_inte**Input:** aggregate as list of attributes

- 1: create an empty tuple A
- 2: **for all** attribute in aggregate **do**
- 3: **if** attribute is a scalar **then**
- 4: append to A memory_c<attribute>
- 5: **else if** attribute is an array of rank 1 with dimension N1 **then**
- 6: append to A memory_c<attribute,int_<N1>>
- 7: **else if** attribute is an array of rank 2 with dimension N1,N2 **then**
- 8: append to A memory_c<attribute,int_<N1>,int_<N2>>
- 9: **else if** attribute is an array of rank 3 with dimension N1,N2,N3 **then**
- 10: append to A memory_c<attribute,int_<N1>,int_<N2>,int_<N3>>
- 11: ...
- 12: return the type generated A

```

1
2 aggregate<
3     memory_c<float>,
4     memory_c<vector<float,Int_<3>>>,
5     memory_c<vector<double,Int_<3>,Int_<4>>>
6 >

```

Listing 2.8: SoA object

Listing 2.9: Internal multi array object for the last attribute

```

1
2 openfpm::multi_array_ref_openfpm<double,
3                                     3,
4                                     boost::mpl::vector<
5                                     boost::mpl::int\_$_<-1>$,
6                                     boost::mpl::int\_$_<3>$,
7                                     boost::mpl::int\_$_<4>$ >

```

At the time of writing this thesis, in OpenFPM there are implemented the two meta algorithms presented above. However, because `memory_traits_*` are free to contain any compile-time meta-algorithm, we can have different implementations of meta-algorithms and consequently transformations of the aggregates.

2.2.6 Access homogenization

The thesis explained in the previous section how to construct at compile-time SoA and AoS objects starting from an aggregate.

Listing 2.10: AoS and SoA

```

1
2 //AoS
3 memory_c<aggregate<float,float[3],double[3][4]>> object;
4
5 //SoA
6 aggregate<
7     memory_c<float>,
8     memory_c<multi_array<vector<float,Int_<3>>>,
9     memory_c<multi_array<vector<double,Int_<3>,Int_<4>>>
10 > object;

```

The two constructed objects in listing 2.10 despite containing the same information, do not expose the same way to access the information. For example, suppose we want to access the matrix component `[1][2]` for the last element. In the AoS case we access such element with `at_c<2>(object[25][1][2])`, while in the case of SoA we access the element with `at_c<2>(object)[25][1][2]`. In particular, the order in which `at_c` is acting is different in the two cases. `Memory_traits_inte` and `memory_traits_lin` abstraction objects can be used to homogenize the access.

In order to homogenize the access, while any `memory_traits_*` objects encapsulate the meta code to transform an aggregate into an AoS or SoA object, they also contain the specialized code for a get function able to access the compile-time object they construct, with a uniform interface. These functions are defined for each `memory_traits_*` object in Listing 2.11, where they expose a get function with the same signature but different implementation. The code in Listing 2.12 shows how from the outside, we can construct generic code valid for both objects. This code shows accessing the same element `([25][1][2])` independently from T being `memory_traits_inte` or `memory_traits_lin`.

Listing 2.11: get function

```

1
2 template<typename T>
3 struct memory_traits_inte
4 {

```

```

5      //! inter_memc is a metafunction encapsulating the
6      //! meta-algorithm to transform the aggregate object T
7      typedef typename inter_memc<typename T::type>::type type;
8
9      // getter function for the object constructed
10     __host__ __device__
11     static inline auto get(data_type & data_,
12                           const key_type & v1)
13     -> decltype(boost::fusion::at_c<p>(data_).mem_r.operator[] (v1))
14     {
15         return boost::fusion::at_c<p>(data_).mem_r.operator[] (v1);
16     }
17 };
18
19
20
21
22 template<typename T>
23 struct memory_traits_lin
24 {
25     //! memory_traits_lin_type is a metafunction encapsulating
26     //! the meta-algorithm to transform the aggregate object T
27     typedef typename memory_traits_lin_type<T>::type type;
28
29     __host__ __device__
30     static inline auto get(data_type & data_,
31                           const key_type & v1)
32     -> decltype(boost::fusion::at_c<p>(
33                 data_.mem_r.operator[] (v1))) &
34     {
35         return boost::fusion::at_c<p>(
36                 data_.mem_r.operator[] (v1)
37                 );
38     }
39 };

```

Listing 2.12: get function

```

1
2 T::type data;
3
4 // ... some initialization
5
6 // access
7
8 auto ele = T::get(data,25)[1][2];

```

The code shown here is still inconvenient, but at least the abstraction constructed homogenizes the access. It is worth mentioning that the parenthesis [1][2] in the code also hides the generic code of the `multi_array_ref_openfpm`. The two layer compose vertically giving not only flexibility in AoS and SoA layout switching, but also in striding reordering for the parenthesis []. This again re-connects on the main OpenFPM design principle of having independent vertical layers. The top layer inherits natural properties from the down-layer without creating specialized code. All associative data structures in openfpm are build on top of this access homogenization pattern.

2.2.7 Multi-indices ordering

In section 2.2.1 the thesis mentioned how the runtime index of a `multi_array_ref_openfpm` is at most one. The reason behind this choice was to separate compile-time indices with runtime indices. This section of the thesis will present the abstraction on the linearization of the runtime indices into a single number. Like before, this abstraction is again an object, exposing a common interface to linearize runtime multi-indices.

In listing 2.13 is shown an example of this object.

Listing 2.13: Example of object to linearize multi-indices

```

1  template<unsigned int N, typename T>
2  class grid_sm
3  {
4  {
5
6
7  public:
8
9      // info methods
10     inline Box<N,size_t> getBox() const
11     inline const Box<N,size_t> getBoxKey() const
12     inline void setDimensions(const size_t (& dims)[N])
13
14     inline size_t size() const
15     inline size_t size(unsigned int i) const
16     inline const size_t (& getSize() const)[N]
17
18     // multi-index linearization methods
19
20     template<typename check=NoCheck, typename ids_type>
21     inline mem_id LinId(const grid_key_dx<N,ids_type> & gk,
22                        const char sum_id[N]) const
23
24     template<typename check=NoCheck,typename ids_type>
25     inline mem_id LinId(const grid_key_dx<N,ids_type> & gk,
26                        const char sum_id[N],
27                        const size_t (&bc)[N]) const
28
29     inline mem_id LinIdPtr(size_t * k) const
30
31     inline mem_id LinId(const size_t (& k)[N]) const
32
33     template<typename ids_type>
34     inline mem_id LinId(const grid_key_dx<N,ids_type> & gk) const
35
36     template<typename a, typename ...lT,
37             typename enabler =
38                 typename std::enable_if
39                     <sizeof...(lT) == N-1>::type
40                     >
41     inline mem_id Lin(a v,lT...t) const
42
43     template<typename a, typename ...lT, typename enabler = typename std::
44         enable_if<sizeof...(lT) == N-1>::type >
45     __device__ __host__ inline mem_id Lin(a v,lT...t) const
46
47     inline grid_key_dx<N> InvLinId(mem_id id) const
48
49     inline mem_id LinId(mem_id * id) const

```



```

50 // utility functions
51
52 inline void swap(grid_sm<N,T> & g)
53     std::string toString() const
54 };

```

The object can be decomposed into three sections. The first section contains methods to get information about each index range on each dimension or set the range of each index on each dimension. The second section is relative to the methods to linearize multi-indices into a single number of type `mem_id` (in 64-bit a long unsigned int). The last section contains utility functions like the **swap** method acting similarly to the one shown in section 2.2.4, and the method `toString` to convert the information contained in this object into a human-readable string.

In the linearization section of the object, all the **LinId** methods are different overloads of the same function. They are all designed to linearize multi-indices with different arguments. For the n-number linearizer case, the enabler parameter uses `SFINAE` (Appendix A) to ensure the method is valid only when we input exactly n-numbers. The method is disabled in all the other cases.

Other linearizers implemented in OpenFPM are `grid_smb`, `grid_zmb`, and `grid_zm`. Figure 2.1 shows all four space-filling curves. In general, using these curves can have benefits in more cache hits depending on our access pattern. In other cases, a different linearization better fits the hardware type like the GPU, where the computation is organized in blocks. This type of linearization will be used with `SparseGrids` later in this thesis.

2.2.8 Combining everything into grids

The thesis introduced `memory_traits_*` as a general way to create objects such that given a tuple, they can create AoS/SoA or an even more complex layout. Additionally, has been introduced an abstraction to linearize multi indices and create general reordering of multi indices. The thesis now combines these two abstractions to create a more convenient multi-dimensional array inheriting all the features of previous layers. In particular, this new layer or object will inherit layout switching AoS and SoA, compile-time parenthesis `[][]` reordering, multi-indices reordering, and capability to work on device or host memory.

This object is again a templated defined in listing 2.14. The templated parameters can be explained as follows:

- **dimensionality** or how many indices you have in your multi-dimensional grid.
- **type** Type stored by the multidimensional array. It must be an aggregate as a wrapper of a tuple. The tuple specifies the properties each point contains. The tuple can contain multi-dimensional static arrays like `float[3]` or `double[4][5]`

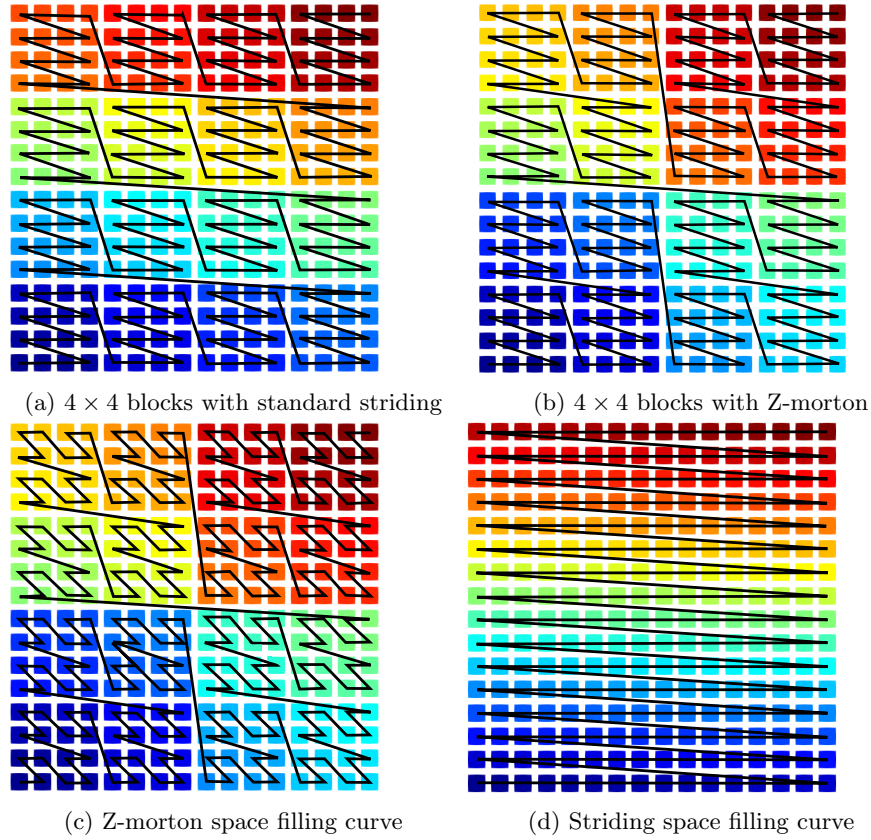


Figure 2.1: 4 different space filling curves, the color indicate the offset in memory of the element from the beginning of the buffer. The scale goes from 0 bytes (blue) to 1024 bytes (dark red)

- `layout_base` The memory layout specifications explained in section 2.2.5. In particular, it gives the possibility to switch between SoA or AoS and index striding.
- `ordering` specify the ordering or linearization of the n-indices into one index (section 2.2.7).

Listing 2.14: Grid object

```

1
2 template<unsigned int dim, typename T, typename S,
3         template<typename>class layout_base,
4         typename ord_type >
5 class grid_base_impl
6 {
7     //! memory layout
8     typedef typename layout_base<T>::type layout;

```

```

9
10      ///! Memory layout specification + memory chunk pointer
11      layout data_;
12
13      ///! This is a structure that store all information related to the grid
14      and how indices are linearized
15      ord_type g1;
16
17      ...
18      template <unsigned int p, typename r_type=decltype(layout_base<T>::
19          template get<p>(data_,g1,grid_key_dx<dim>()))>
20      __device__ __host__ inline r_type get(const grid_key_dx<dim> & v1)
21      {
22          return layout_base<T>::template get<p>(data_,g1,v1);
23      }

```

Many of the parameters come from the layers already explained before in the thesis. `template<typename>class layout_base` provides the meta-algorithm to transform the aggregate into a SoA or AoS object, explained in section 2.2.5. Possible choices are either `memory_traits_lin` or `memory_traits_inte`. As explained in 2.2.7, ordering is instead the function used to linearize the runtime multi-indices. This parameter can be any of the objects `grid_sm` `grid_zm` `grid_smb` `grid_zmb`. While the memory type, select the type of underlying memory the grid create, as introduced in section 2.1. The grid exposes double buffering functionality by providing the function **hostToDevice** and **deviceToHost**. At this level, because `aggregate` is used, these methods now accept an additional template parameter to indicate which indices must be transferred from device to host or from host to device. The type of the object `data_` is constructed at compile-time from the meta-algorithm explained in section 2.2.5. Operatively contains the raw data of the grid. The `g1` object instead is used for multi-index linearization and is instantiated once together with `grid_base_impl`. The code shown for `grid_base_impl` does not include full API of the data structure. Unfortunately, going exhaustively across the entire set of methods is out of the scope of this thesis. The thesis only shows the **get** method coded agonistically from the parameters of the data structure, thanks to the fact that it is built on top of the layers already explained. This concept applies to all the other API functions.

The thesis limit to mention some of the general functionalities of `grid_base_impl`

- **copy_to:** A function to copy a portion of one grid into another grid. Require a grid from where the data must be read, a source box, and a destination box. This function also comes in different flavors, and the thesis will explain later the reason.
- **resize:** As the name suggest, it resizes the grid. The data are preserved when the grid resizes bigger. Data is cropped when the grid resizes smaller.
- **duplicate:** This function duplicates the grid, creating another grid containing the same data.

- **swap:** Given two grids A and B, it swaps the data of the two grids. after A.swap(B) A will contain the data of B, and B will contain the data of A. The grid are forced to be the same type

2.2.9 Vector or 1D array of aggregates

The thesis has shown in the previous section that `grid_base_impl` implements a multi-dimensional array of arbitrary dimensions, storing a list of types on each point with interchangeable layout and ordering. The particular case of a grid in one dimension can be seen as a vector or contiguous array. The reordering loses meaning in this case, but layout switching and the selection of memory types like `HeapMemory` or `CudaMemory` remain valid. For this reason, the construction of a vector data-structure object is on top of `grid_base_impl`. The sketch of the implementation can be seen in listing 2.15

Listing 2.15: Vector implementation

```

1
2 template<typename T,
3         typename Memory,
4         typename layout,
5         template<typename> class layout_base,
6         typename grow_p,
7         unsigned int impl>
8 class vector
9 {
10     size_t v_size;
11
12     //! 1-D grid
13     grid_base_impl<1,T,Memory,typename layout_base<T>::type> base;
14
15     ...
16 }
```

As we can see the implementation on top is based on the special case of a one dimensiona grid `grid_base_impl<1,T,Memory,typename layout_base<T>::type>`. We have two additional parameters; one is `grow_p` that select the growing policy, the other is `impl` that select the internal implementation. To explain the parameter `grow_p`, consider the case of a vector with 100 elements that lead to the internal structure `grid_base_impl` base having 100 1D elements. Once we require 101 elements, we have to resize the one-dimensional grid. We could, in theory, reallocate a new 1D grid with 101 elements, but in the eventuality of further subsequent new elements, it would force a reallocation to every addition. It is, in general, better to pre-allocate more elements. The growing policy defines the strategy of how many additional elements must be created at each reallocation. For example, `grow_double` indicates to double the vector's size every time we have to grow the internal one-dimensional grid. The `grow_identity` indicates to increase the size exactly the amount required. The object `grow_p` is again an object exposing a well-defined interface.

In listing 2.16 we can see an example for `grow_policy_double`, in particular the function accept two-arguments, the first is the actual size of the one-dimensional grid. Because reallocations always come from the user requesting a certain size,

the second argument indicates the new number of elements for the vector. The function in the object tries to increase the previous size of the vector by a factor of 2 until we have enough elements to store the requested size. Special care is given when the previous size is zero. In this case, the variable "grow" is initialized to 1; otherwise, the variable would remain zero in the multiplication. The function returns the total number of elements for resizing the internal one-dimensional grid. Having a growth policy based on some factor n from the previous size guarantee the average insertion time to be constant.

Listing 2.16: Grow policy

```

1
2     class grow_policy_double
3     {
4     public:
5
6         static size_t grow(size_t original, size_t requested)
7         {
8             size_t grow = (original == 0)?1:original;
9             while (grow < requested) {grow *= 2;}
10            return grow;
11        }
12    };

```

The number of elements in `grid_base_impl` indicates the number of elements reserved, or alternatively, how much the vector has to grow before a reallocation is triggered. `v_size` gives the actual number of elements in the vector.

```

1
2     template<typename T>
3     using vector_gpu = openfpm::vector<T,
4                                     CudaMemory,
5                                     typename memory_traits_inte<T>::type,
6                                     memory_traits_inte>;
7
8     template<typename T>
9     using vector_gpu_single = openfpm::vector<T,
10                                           CudaMemory,
11                                           typename memory_traits_inte<T>
12                                           >::type,
13                                           memory_traits_inte,
14                                           openfpm::grow_policy_identity>;

```

The parameter "impl" is a compile-time condition and selects the implementation based on input T. In case T is not an aggregate the meta-function `vect_isel<T>::value` will return `STD_VECTOR`, while if T is an aggregate the implementation in listing 2.15 will be chosen. As a grid, a vector exposes similar functionalities like resizing, duplicate, swap and get functions. Because the vector is a special case, it also implements additional functionalities more specific to the dynamic array and list concepts: sorting elements, single element removal, bulk remove of elements, or merge two lists together.

2.2.10 From nested structures to aggregate

When the thesis introduced aggregates, it was explained the possibility of converting any structure into an aggregate. On the other hand, the thesis never explained how a nested structure could be converted into an aggregate.

Let consider a structure like in listing 2.17, a typical way to convert it, would be to create a nested aggregate like the one in listing 2.18

Listing 2.17: structure

```

1
2 struct nested
3 {
4     int n_ele1;
5     double n_ele2;
6 }
7
8 struct A
9 {
10    int ele1;
11    double ele2;
12
13    nested ele_n;
14 }
```

Listing 2.18: structure

```

1
2 constexpr int ele1=0;
3 constexpr int ele2=1;
4 constexpr int ele_n=2;
5 constexpr int n_ele1=0;
6 constexpr int n_ele2=1;
7
8 aggregate<int,double,aggregate<int,double>>
```

Nested aggregates work with all openfpm data structures, but the layout switch is limited to the first-level aggregate. The meta algorithms `memory_traits_*` just read the first aggregate list and consider the second level a single element of the first aggregate.

We could, in theory, create a meta-algorithm considering also the second level, but it has been decided to approach the problem differently and more effectively. We start to observe that any nested aggregate can be flattened into a non-nested aggregate. For example, the nested structure in listing 2.17 following a depth-first traversal of the tree, generates the following one-level aggregate `aggregate<int double, int, double>`. In general, we have a combinatorial number of ways we can flatten the aggregate. This flexibility can be synthesized with a map where every field, nested or not, maps to a number. Listing 2.19 show an example of the definition of such a map. For example `A::ele1` map to 0 `A::ele2` map to 1 `A::nested::n_ele1` map to 2 `A::nested::n_ele2` map to 3 leading to the standard depth-first traversal algorithm flattening. The possibility of rearranging the numbers allows us to re-map any field in the nested struct into a flattened aggregate. The flattening map also ensures that whatever permutation of numbers we choose to convert names to number, this choice does not interfere in accessing data. Another interesting feature of having an external numerical maps is the possibility to have multiple maps and names to numbers for the same data structure. For example, we can define a second map with a different tree structure like in listing 2.20. This feature is not available when we use nested aggregates because defining a nested aggregate fixes how we will access

the elements. For example having a container of an aggregate defined like in listing 2.18 accessing the field double in the nested aggregate at position p , it is forced to be `data.get<nested>(p).get<n_ele2>`. On the other hand in the case we use a numerical map on a flatten tree, the elements can be accessed with `get<A::nested::n_ele2>(p)` or alternatively with `get<A_s::n_ele2>(p)`.

Listing 2.19: structure

```

1
2 struct A
3 {
4     typedef std::integral_constant<int, 0> ele1;
5     typedef std::integral_constant<int, 1> ele2;
6
7     struct nested
8     {
9         typedef std::integral_constant<int, 2> n_ele1;
10        typedef std::integral_constant<int, 3> n_ele2;
11    }
12 }
13
14 typedef aggregate<int,double,int,double> A_aggr;
```

Listing 2.20: structure

```

1
2 struct A_s
3 {
4     typedef std::integral_constant<int, 0> ele1;
5     typedef std::integral_constant<int, 1> ele2;
6     typedef std::integral_constant<int, 3> n_ele2;
7
8     struct nested
9     {
10        typedef std::integral_constant<int, 2> n_ele1;
11    }
12 }
13
14 typedef aggregate<int,double,int,double>
```

The possibility to flatten the tree and map fields/names into numbers give us also the possibility to construct functions to copy fields from one structure into another that work across the boundaries of the trees and sub-trees.

object_creator and functions

An example of such functionality is the possibility to construct from an aggregate another aggregate composed by a subset of properties from the first aggregate. More in the specific the meta-function `object_creator<typename prop::type,A::ele2,A::nested::n_ele>::type` using the positional numbers `A::ele2` and `A::nested::n_ele` create a tuple object `aggregate<double,int>` grouping two elements across different tree levels. This type can be used to construct a completely new container over a sub-set of the selected properties like shown in listing 2.21. The function `object_si_d<...,A::ele2,A::nested::n_ele>` is able to copy elements from a container `openfpm::vector<A_aggr>` to the container with a subset of fields

constructed in listing 2.21. The same goes for the function `object_s_di<..., A::ele2, A::nested::n_ele>` is able to copy from the container with a subset of properties to the original container.

Listing 2.21: create an object from a subset of fields

```

1
2 typedef object<typename object_creator<typename A_aggr::type, A::ele2, A::nested::
  n_ele>::type> prp_object;
3
4 openfpm::vector<prp_object> data;
```

2.2.11 Gpu and multi hardware

We saw in many code snippets how all the code written is compatible with accelerator compilers like the Nvidia Cuda compiler, because many of the methods were marked `__device__` `__host__`, in order to generate GPU and CPU code. On the other hand, in order to work on GPU, the data structures have to work on device pointers or host unified memory pointers rather than host-only pointers.

CudaMemory contains a device pointer and a host unified pointer. While a GPU kernel can access both memories, we always prefer to use device memory in a kernel for performance reasons. The data structure `grid_base_impl` can be passed to a GPU kernel using the method `toKernel()`. This function creates a proxy object of type `grid_base_ker` that internally use the device pointer instead of the host pointer. The proxy object is passed by value to a kernel, and acting on the proxy object inside a kernel means working with the device memory. All containers provide this function to write algorithms on GPUs, and all containers are able to work on nested containers like `openfpm::vector_gpu<aggregate<int, openfpm::vector_gpu<aggregate<int, double[3]>>>`. In this case the `toKernel()` method parse at compile-time the entire tree and create an equivalent proxy object on GPU with nested containers `openfpm::vector_gpu_ker<aggregate<int, openfpm::vector_gpu_ker<aggregate<int, double[3]>>>`. There is no limit on how many containers are nested and which container, as soon as it expose a `toKernel()` function returning a proxy, it is possible to nest-it. Although this feature provides flexibility on GPU, it must be used with care. Fragmenting the memory on GPU creating micro allocation and deallocation are very expensive on GPU. A `cudamalloc` function used to allocate GPU memory on Nvidia is in general order of magnitude slower than heap allocation on CPU.

Using a proxy object, rather than the original structure allows us to first reduce the amount of information to pass to a kernel, hence the usage of the parameter space, and second restrict the interface to methods that work on GPU. If we pass the original data structure to the kernel, all fields inside the class will move to GPU, whenever they are used or not. It is common for classes to contain information that makes no sense to be transferred on GPU. This problem mostly does not occur for simple single-node data structures like a vector or a grid presented until here. However, this problem becomes predominant for more complex cases like a distributed data structure or sparse grids presented later in this thesis.

2.2.12 Performance

The thesis showed how the design of each layer gives modularity, flexibility, and compact code. It remains to check if the compile-time design using template meta-programming interferes with the compiler's optimization stages, avoiding performance penalization. To check that the assumption holds, the thesis compare the high level data-structure explained until here with plain array.

The benchmark creates two vectors containing 2097152 elements on CPU and 16777216 on GPU. The two vectors contain scalars types like double, vector properties like double[2], and tensorial properties like double[2][2]. The code in listing 2.22 use OpenFPM data-structures to move and shuffle data from one vector into another involving all properties and every element. The code in listing 2.22 is clearly memory badwidth limited and is valid for any type of layout like explained in this chapter. An equivalent code for plain array is also created in listing 2.23 valid only for array of struct and listing 2.24 valid only for the structure of array. The performance of the openfpm code listing 2.22 is than compared against the multiple version of codes needed for different layout in the case of plain arrays (listing 2.23 and 2.24). In table 2.1 we report the performance results with different architectures. With the exception of the GTX1650 in memory_traits_lin reporting a 1 percent loss in performance; the table shows no performance impact, within the margin of error, using the OpenFPM data structure against arrays, neither on Intel or AMD CPUs nor on Turing or Ampere Nvidia GPUs in accessing time using the `get` functions explained before.

	memory_traits_lin	memory_traits_inte
Intel i9 9900	1.02 ± 0.07	1.02 ± 0.037
AMD ThreadRipper 3990X	1.01 ± 0.02	1.01 ± 0.02
GTX 1650	1.023 ± 0.01	1.000 ± 0.003
RTX 3090	1.00 ± 0.037	1.01 ± 0.02

Table 2.1: Performance on different CPUs (Intel/AMD) and GPUs (Nvidia Touring/Ampere) with different layouts. The number indicate the ratio between the time to complete the data-movements in OpenFPM listings 2.22 against plain arrays listing 2.23 (left column) or listing 2.24 (right column)

Listing 2.22: OpenFPM data structures working for any layout

```

1
2 template<typename vector_prop_type, typename vector_pos_type>
3 __device__ __host__ void read_write(vector_prop_type & vd_prop, vector_pos_type &
4   vd_pos, unsigned int p)
5 {
6   vd_prop.template get<0>(p) = vd_pos.template get<0>(p)[0] + vd_pos.template get
7     <0>(p)[1];
8   vd_prop.template get<1>(p)[0] = vd_pos.template get<0>(p)[0];
9   vd_prop.template get<1>(p)[1] = vd_pos.template get<0>(p)[1];

```

```

10 vd_prop.template get<2>(p)[0][0] = vd_pos.template get<0>(p)[0];
11 vd_prop.template get<2>(p)[0][1] = vd_pos.template get<0>(p)[1];
12 vd_prop.template get<2>(p)[1][0] = vd_pos.template get<0>(p)[0] +
13                               vd_pos.template get<0>(p)[1];
14 vd_prop.template get<2>(p)[1][1] = vd_pos.template get<0>(p)[1] -
15                               vd_pos.template get<0>(p)[0];
16
17 vd_pos.template get<0>(p)[0] += 0.01f;
18 vd_pos.template get<0>(p)[1] += 0.01f;
19 }

```

Listing 2.23: Plain array AoS

```

1
2 struct ele
3 {
4     double s;
5     double v[2];
6     double t[2][2];
7 };
8
9 __device__ __host__ void read_write_lin(double * pos, ele * prp,
10                                         unsigned int p)
11 {
12     prp[p].s = pos[2*p] + pos[2*p+1];
13
14     prp[p].v[0] = pos[2*p];
15     prp[p].v[1] = pos[2*p+1];
16
17     prp[p].t[0][0] = pos[2*p];
18     prp[p].t[0][1] = pos[2*p+1];
19     prp[p].t[1][0] = pos[2*p] +
20                     pos[2*p+1];
21     prp[p].t[1][1] = pos[2*p+1] -
22                     pos[2*p];
23
24     pos[2*p] += 0.01f;
25     pos[2*p+1] += 0.01f;
26 }

```

Listing 2.24: Plain array SoA

```

1
2 __device__ __host__ void read_write_inte(double * pos,
3                                           double * prp0,
4                                           double * prp1,
5                                           double * prp2,
6                                           unsigned int p,
7                                           unsigned int n_pos)
8 {
9     prp0[0*n_pos + p] = pos[0*n_pos + p] + pos[1*n_pos+p];
10
11     prp1[0*n_pos + p] = pos[0*n_pos + p];
12     prp1[1*n_pos + p] = pos[1*n_pos + p];
13
14     prp2[0*n_pos*2+0*n_pos + p] = pos[0*n_pos + p];
15     prp2[0*n_pos*2+1*n_pos + p] = pos[1*n_pos + p];
16     prp2[1*n_pos*2+0*n_pos + p] = pos[0*n_pos + p] +
17                                     pos[1*n_pos + p];
18     prp2[1*n_pos*2+1*n_pos + p] = pos[1*n_pos + p] -
19                                     pos[0*n_pos + p];
20
21     pos[0*n_pos + p] += 0.01f;

```

```

22     pos[1*n_pos + p] += 0.01f;
23 }

```

2.2.13 Graphs

Starting from basic data structures like a multi-dimensional array and vector able of reordering elements, changing layout, working on host, and device memory, it is possible to construct more complex data structures like graphs with the same features.

In OpenFPM, graphs store information about the decomposition but have been designed to have a more general interface. Its definition follow listing 2.25

Listing 2.25: Graph definition

```

1
2 template<typename V, typename E = no_edge,
3         typename Memory = HeapMemory,
4         template<typename> class layout_v_base = memory_traits_lin,
5         template<typename> class layout_e_base = memory_traits_lin,
6         typename grow_p = openfpm::grow_policy_double>
7 class Graph_CSR
8 {
9
10     //! Structure that store the vertex properties
11     openfpm::vector<V, Memory, layout_v_base,grow_p> v;
12
13     //! Store the number of edges for each vertex
14     openfpm::vector<size_t, Memory, layout_v_base,grow_p> v_l;
15
16     //! Edge properties information
17     openfpm::vector<E, Memory, layout_e_base, grow_p> e;
18
19     //! For each vertex store the adjacent vertex and
20     //! the edge id
21     openfpm::vector<e_map, Memory, layout_e_base , grow_p> e_l;
22
23     ...
24
25 }

```

- **V** Is an aggregate and indicate what is stored as information on the vertices.
- **E** Is an aggregate and indicate what is stored as information on the edges
- **layout_v_base** layout to use for the vector storing information about vertices
- **layout_e_base** layout to use for the vector storing information about edges
- **grow_p** Grow policy to be used for the vector to store vertices and edge information

The graph defined in listing 2.25 can store an aggregate on the vertices and the edges. At the same time, because the graph is shaped using the OpenFPM vectors, inherit the properties to switch layout and use GPU memory.

2.2.14 sparse grids on shared memory

Sparse grids are data structure useful for level-set problems and complex geometry, we will see examples of these type of problems later in this thesis. Sparse grids are grid data structures where points/nodes can be dynamically inserted and removed at runtime. A simplified view is to see a sparse grid as a hash map, where the key is the grid index (i.e., discrete position in the grid) and the value is the data stored at that point. An empty sparse grid does not allocate any memory for the grid data, only for the access and bookkeeping data structures. The sparse grid is then populated using an insert function. This method is in contrast to a dense grid, where all the memory is allocated at instantiation. The usual get function is present in sparse and dense grids to read or write the value at an allocated grid point. Another critical difference between dense and sparse grids is how they are traversed or iterated over. In dense grids, iterators usually proceed to neighboring points until all points have been visited. In sparse grids, iterators can visit all existing points or any subset. Apart from the specific limitations of a numerical method, this interface allows for easy conversion of dense-grid codes to sparse-grid codes, valid for comparative benchmarking.

The changes required to convert a code from sparse to dense are: (1) inserting all existing points at the beginning and (2) changing the iterator to an all-node sparse-grid iterator. In most applications, the inserted points are not randomly scattered in a sparse grid but concentrate in dense sub-regions. Sparse block grids exploit this to reduce random and neighborhood access time. Instead of storing each inserted point individually, they store a list of blocks of points. A common approach is to divide the grid into regular blocks of equal size (size set at compile-time) and allocate a whole block as soon as one point is inserted. These blocks are called chunks, and a globally unique chunk ID identifies each chunk of a sparse block grid. Maintaining this information for a sparse-grid requires a few bookkeeping data structures, as illustrated in figure 2.2. From the characteristics of the sparse grid, it is observable that this structure lies in between particles and standard dense grids. This chapter will see how to implement a sparse grid based on the abstractions presented before.

A standard dense grid allocates all the elements at construction. In the case of a 5×5 grid of integers, the memory is allocated at once ideally as 25 integers array. A sparse grid 5×5 of integers does not allocate memory other than the bookkeeping data structures. The sparse grid is populated incrementally using the insert function `sparse_grid.insert<0>(point) = 5.0`. The instruction inserts a point in the position indicated by the index `point` and assigns to the property zero the value 5.0. A subsequent `sparse_grid.insert<0>(point) = 6.0` will just overwrite the value of the previous insert. The old `get` function present in the dense grid is still valid but now it return a read-only value so `sparse_grid.get<0>(point) = 7.0` would end up in a compile-time error. Doing a `get` in an empty grid is valid and will return a background value. Every property can set a predefined background value. Background values for the sparse grids come in handy in case we want to set some boundary value outside the existing points or some out-of-bound value.

Another difference for the sparse grid compared to dense lies in the iterators. In the standard grid, iterators are constructed to run across the grid points using the method `getIterator(start, stop)` where the start and stop indicate the section of the grid in which we want to iterate. In the sparse grids there are two methods, `getGridIterator(start, stop)`, and `getDomainIterator(start, stop)`. The first iterate across all grid points between the start and stop points, independently from their existence. The second method iterates across the existing points between the start and stop.

This type of interface allows smoothly converting a code that does a calculation on a dense grid to a sparse, simply replacing the **get** method into the **insert** method, and changing the first iterator that initializes the grid from `getDomainIterator` to `getGridIterator`.

Implementation

As mentioned before, sparse grid memory is not allocated immediately but incrementally while the data is inserted. Additionally, data is organized in blocks. Blocks are called chunks and divide the grid into pieces that contain at least one or more points. Figure 2.2 show the implementation of a 2D sparse grid that stores a scalar and a two-dimensional vector. This design choice comes from two observations: the first is that most applications do not have a truly random scattered set of points but regions with points, so it is better to aggregate points into blocks. The second is that blocks have a natural way to map data into computation blocks, a common paradigm for massively parallel accelerators.

The blocks in the following will be called chunks, and a globally unique chunk ID identifies each chunk of a sparse block grid. In order for this data structure to work, only a few bookkeeping data structures are needed, as illustrated in figure 2.2 and explained in the following.

- **chunks:** An `openfpm` vector that contains the data stored by a chunk. In the case of figure 2.2 4×4 chunks are used. This case requires 16 scalar elements and 16 `double[2]` elements to store a chunk. The size of the chunks is set and known at compile-time. Because `openfpm::vector` is used, there is the freedom to choose the internal layout of the data and memory type.
- **headers:** The header array in figure 2.2 contains for each chunk information of the position of the chunks and a bitmask that indicates if a point is filled or not. In the case of a 4×4 chunks the mask need 16 bit equivalent to a `shortint` in C++. In the figure, red points in the chunks mean they exist and are indicated with 1 in the mask, while black points are indicated with 0. Note that chunks and header arrays contain the same amount of elements.
- **linearization.** It is indicated in figure 2.2 as an arrow from positional information into a number. It linearizes the position in coordinates of a

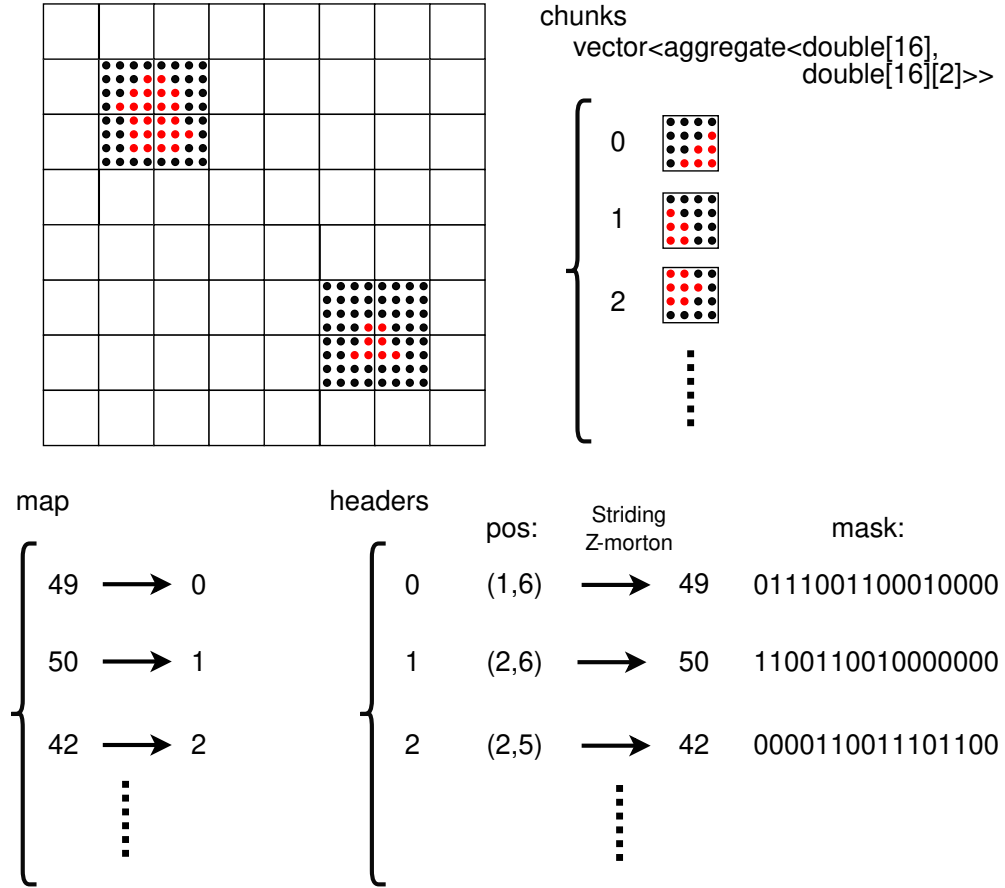


Figure 2.2: Implementation of a Sparse-grid 25x25 in 2D with chunks 4x4

point into a chunk number. The code is injected into the structure using a template parameter. For example, in the case of a chunk at position (1,6). The linearization formula for normal striding is $(1,6) \rightarrow 6 \times 8 + 1 = 49$ while for Z-Morton curve the bits of the x,y coordinates are interleaved $1 = 001, 6 = 110 \rightarrow 010110 = 22$.

- **map:** The map converts from a chunk id to its position in the vector of **chunks**. This map has been implemented either using a sorted array or a hash-map

The data-structure is defined in the code listing 2.26

Listing 2.26: Definition of a sparse grid

```

1
2 template<unsigned int dim,
3         typename T,

```

```

4      typename S,
5      typename grid_lin,
6      template<typename> class layout_base,
7      typename chunking>
8  class sgrid_cpu

```

Only the first two parameters are mandatory, and the others have default values for CPUs (S=HeapMemory,layout_base=memory_traits_lin) and accelerators (S=CudaMemory,layout_base=memory_traits_inte)

- **dim** is the dimensionality of the grid.
- **T** Type stored by the multidimensional array. It must be an aggregate.
- **S** Type of memory used to store data CudaMemory or HeapMemory.
- **chunking** Dimensions of the chunk in each dimension, because is given as template parameter its dimensionality is known at compile-time.
- **grid_lin** This is the parameter containing the code to linearize a multi-index into two numbers: chunk id and offset within the chunk.
- **layout_base** is the specifier for the internal layout of the internal array/vectors like SoA or AoS.

Performance tuning with physical chunk tiling

Most of the operations employed on a grid are local convolutions like second-order 7 points finite-difference Laplacian stencil. Physical tiling of the grid into chunks allows us to allocate only one part of the grid. However, this layout complicates operations like stencil computation. This section analyzes the performance achieved using this layout, the challenges, and how it has been solved. The performance tests use a 7-point second-order Laplacian stencil with finite differences, a SoA layout, and a Z-Morton linearization. The Vc library is used to have explicit vectorization, and the computation is performed in a block-wise way.

The Z-Morton curve increase cache hit in the L3 cache level of the CPU. The profiler measured that this linearization improves L3 cache hit by 30% and the performance by 15%. Calculating the stencil on a block requires the surrounding points depending on how wide the stencil is. A possible solution would be to create a temporary block with borders and load the data on the temporary block before calculating the stencil on the inside points. Assuming that the temporary block is small enough to live in the L1 cache, the data would travel (Memory - Register - L1) to load the block (L1 - Register - L1) for the computation and (L1 - Register - Memory) to store the data. This approach does not lead to the best performance. Despite having a low latency L1 cache, the CPU has to fetch and execute a lot of operation/movement that involves the L1 cache. It turns out that avoiding the temporary block and land on the register, leading to (Memory - Register - Memory) is a factor two faster.

In order to reach the (Memory - Register - Memory) path, the addresses to load are initially computed and subsequently used to do an aligned vectorized load. Special care is required to load data shifted on direction x because loading the four numbers in the purple rectangle of figure 2.3 is split across two chunks. Figure 2.3 shows an example, where an AVX load is able to load four number (double type) on the register (blue rectangle). Loading the four number in the purple rectangle requires splitting the single AVX load instruction into multiple operations. The first operation loads the four doubles in the blue rectangle, followed by a right shift on the register by 64 bits, and finally a load of a single double in the last 64 bits of the register. Figure 2.3 shows how the chunk layout does require additional operations to calculate addresses of points across chunks.

An overhead from sparse grids compared to dense ones comes from the necessity to check the existence of each point. Fortunately, because the information about the existence resides in a structure with the same layout and order of the point-data, calculating the address in memory follows the same rule explained in figure 2.3. The mask is loaded in the same way. In the case of 4 doubles, loading the mask mean loading 4 bytes for a total of one integer. A simple check of the integer to zero checks in one hit if one of the four points is filled.

Checking the mask for filled points is not enough. In many cases, it is necessary to restrict the stencil calculation to a subset of grid points. In order to restrict the computation, the chunk is intersected with the area where to calculate the stencil. The intersection is used to calculate the iteration limits on y and z within the chunk. On x or along the contiguous dimension, the iteration always starts from zero to guarantee aligned vectorized load. A mask is calculated on x to indicate which points are part of the iteration. For example, in case it is needed to calculate the stencil only on the first five elements in the x direction of the chunk, the mask will be *true, true, true, true, true, false, false, false*. The mask is used to perform an optimized masked store to guarantee that only the points within the valid range are written.

Manually performing all the optimizations explained above is complex and error-prone and requires machine code analysis. The sparse grid provides a lambda-based interface where the lambda contains the stencil core computation.

results

The first performance test compares the performance of a sparse grid with the standard dense grid. The benchmark uses a simple second-order finite difference Gray-Scott simulation on sparse grid and dense grid. In the sparse case, all points are filled. Analysis of the machine code shows that the compiler produces vectorized instruction in both cases. Performance tests are conducted on an Intel CPU i7-8700. In table 2.2 the dense grid is faster than the sparse grid. As explained in section 2.2.14 a sparse grid has to handle the mask to check if a point exists or not. In the column SparseGrid-NM (No Mask), the instructions relative to the mask handling are commented to check its impact on this test. It is possible to ignore the mask, in this case, because all points in the grid are filled. A closer look at the machine code shows that if the user-supplied function

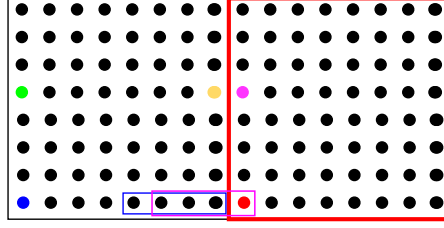


Figure 2.3: Two 8×8 adjacent chunks on direction x. The figure shows how it is possible to calculate the addresses of every point for two near chunks, given the address of the first point of each chunk (blue and red point). Considering the position in memory for the point in red $address_{red}$ and the address of the blue point $address_{blue}$. The double is 8 bytes in size, and the address of the violet point is $address_{red} + 32 * 8$. The green dot is calculated as $address_{blue} + 32 * 8$, while the yellow is calculated as $address_{blue} + (32 + 7) * 8$. From the yellow address, it is possible to calculate the violet point address summing to the yellow address the offset $address_{red} - address_{blue} - 7 * 8$

does not contain instructions involving the mask, many of the mask loading instructions and processing are eliminated by the compiler because of dead code elimination. The compiler performs this type of optimization because, as explained in section 2.2.14 a memory - register - memory gives the compiler the possibility to see such optimization. The 0.005 seconds improvement comes from disabling the few remaining mask load and processing instructions. The significant overhead comes from the additional complexity produced by the chunks layout to calculate the memory addresses before loading the operands on the register.

Dense-Grid	Sparse-Grid	Sparse-Grid (NM)
0.063 s	0.083	0.077

Table 2.2: Time required to compute the right-hand side of the equation and update the fields U, V for the case of the standard grid and Sparse-Grid where has been commented out all the instructions relative to the mask handling (NM)

Extension to GPU or massively parallel hardware

In order to extend the sparse-grid to GPU or massively parallel architectures, it is necessary to address the problem of inserting points in parallel. This operation is challenging in parallel for mainly two reasons: an insertion can generate memory relocation, two threads can try to write the same point generating a race condition. Consider the case where two threads try to add 2 and 3 into the same accumulator variable. If we do not perform an atomic operation, the accumulator can result in different results depending on how the threads access

the accumulator. In order to avoid such conflicts, the operation is spitted into two phases. In the first phase, collect all the changes while the second merge the changes into our structure.

In order to explain how the resolution of the conflicts is performed, it is necessary to introduce basic parallel operations for massively parallel architecture.

Basic operation for massively parallel architecture

- **scan/exclusive prefix sum:** Given an array [1,6,9,10] the exclusive scan operation produce on each entry the sum of all the previous entries [0,1,7,16,26]
- **merge:** Given two sorted array [1,6,7,10,26] and [0,3,7] this operation produce a single sorted array with merged entries [0,1,3,6,7,7,10,26]
- **sort:** for an array [1,10,2,6,26] an ordered buffer is produced [1,2,6,10,26]

It has been shown and proven that these operations can be parallelized on massively parallel architectures [58] [59] [60]. For our purpose, we limit to observe that when we are able to decompose our algorithm into a combination of parallel kernels and a set of basic operations listed above, the algorithm scale to massively parallel architecture.

The following two sections will show how to add data with potential conflicts in the case of a massively parallel architecture. This operation is split into two phases.

First phase

In the first phase, two buffers collect all changes requests. One buffer stores the added chunk ids, and the other buffer stores the data. Rather than keep a reserved space for each thread, a block of thread has a reserved space to add ids and data. The reason is to reduce the allocated memory for the two buffers. If we consider the typical case where the full block of thread fills only one chunk, having the granularity to specify the number of reserved chunks for one block of thread gives the possibility of having one chunk entry for the entire block of thread. If we choose the granularity of the threads, if the block has 256 threads, we would be forced to have at least 256 entries. The adding buffer is allocated before running the kernel that makes the insertions.

Second phase

The second phase runs the function flush. This function requires choosing an operation for each property. The operation indicates how to merge and solve conflicts. Suppose there are two insertions with values 3 and 4 on an existing point containing the value 5. In order to have consistent results, one parameter specifies how the data will be merged. For example, choosing maximum as operation will produce 5, minimum will produce 3, and summation will produce

12. Our final goal is to obtain the buffer of updated chunks ids and updated data chunks from the list of added chunks. Algorithm 4 perform this task resolving conflicts.

Algorithm 4 FlushMethod

Input: Ids of added chunks and Data of added chunk

- 1: Make the add chunks ids contiguous
 - 2: Sort the added chunks ids
 - 3: Merge the list of the old indices with the list of the added unique indices into a merged list
 - 4: From the merged list construct the set of unique indexes (This is our list of sorted indices for the updated structure)
 - 5: Merge the add data buffer using the merged list and list of segments to construct the final list of merged data (The result is the list of updated data chunks)
-

figure 2.4 illustrates the algorithm 4 . Is good to notice that the data reduction and copy are made at the very end. Moving chunks is very expensive in terms of memory movement, for each chunk id (4/8 bytes), there is a data chunk. For example, in the case of a scalar field in 2D containing doubles with chunks 16×16 , the data chunk has 2048 bytes attached and an additional 16×16 bytes for storing the mask for a total of 2304 bytes. It is fundamental to move the least amount of chunks data. For this reason, the flush algorithm work on chunk ids for most of the time constructing maps to retrieve the chunks data during the final reduction and data merge.

Generalizing data-structure to a sparse vector on GPU

The algorithm 4 presented above is encapsulated into a more general structure called sparse vector data structure working on GPU. The concept is the same as a standard vector or a resizable array with only a few elements filled. Such structure can be seen as a hash map where the key is either a 32-bit or 64-bit integer. The aggregate does not necessarily contain chunks but can also contain simple scalars, and the ids are not necessarily connected to the position of the data in an N-dimensional space.

Listing 2.27: Map on gpu

```

1
2 template<typename T,
3         typename Ti = long int,
4         typename Memory=HeapMemory,
5         template<typename> class layout_base=memory_traits_lin,
6         typename grow_p=grow_policy_double,
7         unsigned int impl=vect_isel<T>::value,
8         unsigned int impl2=VECTOR_SPARSE_STANDARD,
9         typename block_funcor = stub_block_funcor>
10 class vector_sparse
11 {
12     ...
  
```

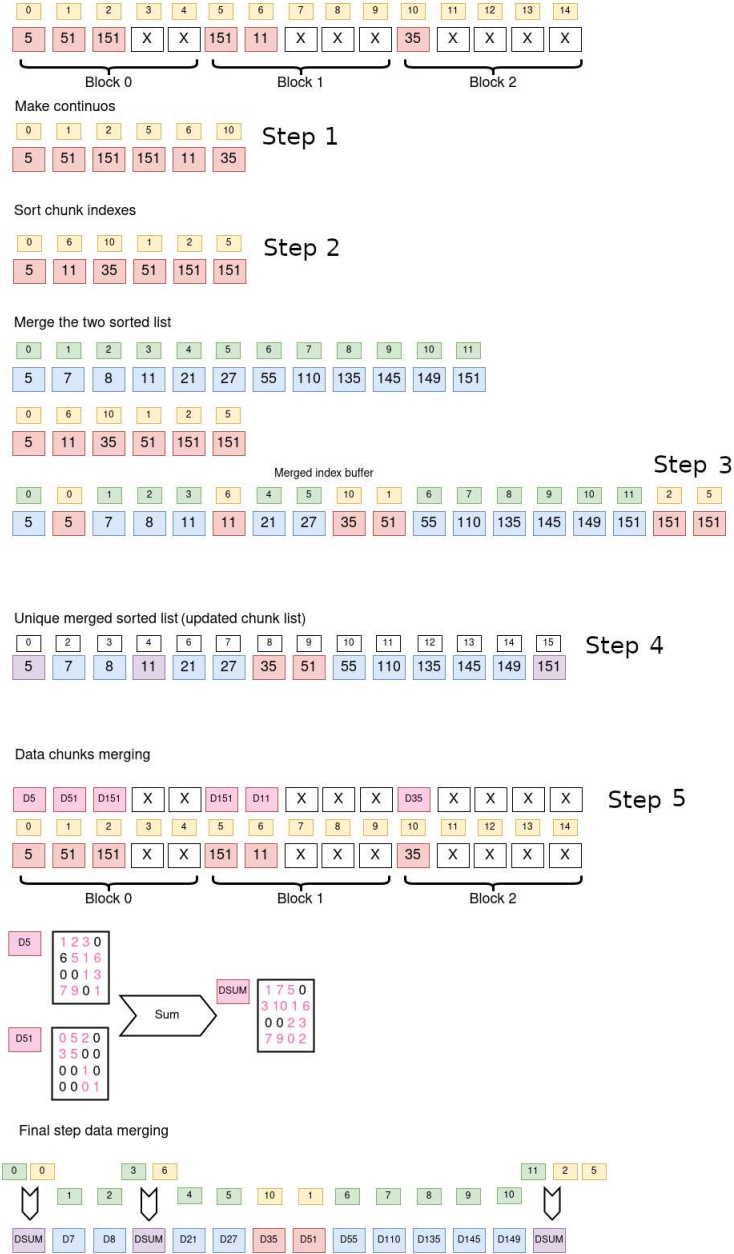


Figure 2.4: Step-by-step example of the **flush** operation from Algorithm 4. Yellow boxes show positions of the chunks in the original array. Red boxes are the corresponding chunk IDs with “X” marking nonexistent chunks. Curly braces indicate how chunks are grouped in GPU thread blocks (five chunks per thread block) for parallel execution. Green boxes are the positions of existing data in the pre-existing chunks. Purple boxes indicate chunk IDs that require data merging. Boxes labeled D_i represent the data contained in the chunk with ID i . The example in the penultimate row shows how D_5 is merged using the reduction operation **sum**, creating a new chunk $DSUM$ containing the element-wise sums of the data from the two input chunks. Black numbers indicate unused grid points containing invalid data, while magenta numbers indicate valid data. The green and yellow boxes are used to track *how* to merge the data in the final step. These maps avoid moving data chunks when determining correspondence groups.

13 }

- **T** Type stored by the sparse vector. It must be an aggregate
- **Ti** precision of the index used for the chunk ID. By default the parameter select 64 bit integers. This means the map can store indices up to $2^{64} - 1$, while in case of int it can store up to $2^{32} - 1$.
- **Memory**: type of memory used to allocate arrays to store data.
- **layout_base** Layout to use to store data AoS or SoA 2.2.5
- **grow_p** is an object exposing a well defined interface to control the policy to allocate memory of the internal arrays. It is explained in section 2.2.9
- **impl** implementation of the internal arrays explained in section 2.2.9
- **impl2** indicate the type of implementation of the sparse vector VECTOR_SPARSE_STANDARD is the standard implementation where the value is an aggregate, VECTOR_SPARSE_BLOCK is the block implementation required when the value are chunks and the implementation has been explained in section 2.2.14
- **block_func** in case of VECTOR_SPARSE_BLOCK is a function to handle the merging of chunks for the last step of the flush. This function is provided as a functor

The sparse grid specialized on GPU depends on the sparse vector presented here. In particular, values are the chunks, while the keys are 64/32 bit integers of linearized chunks indices.

Performance on GPU

When running the sparse block grid code on a single Nvidia GTX 1080 GPU, the runtime reduces by a factor of about 35 compared to the single-thread CPU implementation on the 3.2 GHz Intel i7-8700 (see Table 2.3). Since the grid in this benchmark is dense, all chunks of the sparse block grid are fully occupied, leading to the best thread efficiency on the GPU. The *density* of a sparse block grid is defined as the average (overall allocated chunks) occupancy of the chunks, i.e., the average fraction of grid points in each existing chunk that are allocated/inserted. The lower the density, the lower is the thread efficiency of our GPU implementation. This is confirmed in the measurements reported in Table 2.4 for a 512^3 grid of `float` valued on a single GTX 1080. The thread efficiency is normalized at density 1.00 and from there reduces approximately linearly with the grid density. This is expected because, in our implementation, the density of a chunk directly determines the fraction of busy versus idle threads in each GPU threads block (cf. figure 2.4).

CPU and GPU implementations are compared with the widely used OpenVDB [61] library for sparse volumetric data structures. In this comparison, the

plain array CPU	sparse grid CPU	sparse grid NM	sparse grid GPU
0.063	0.083	0.077	0.0024

Table 2.3: The table gives the runtime in seconds required to compute one time step of the Gray-Scott simulation on a dense regular Cartesian 256^3 grid. The table compare plain-array and sparse block grid implementations on the CPU. Commenting out the mask handling functions in the sparse block grid code (“NM”) quantifies their overhead. When run on the GPU, the sparse block grid code is about 35-times faster than on the CPU (last column).

density	insert	stencil
0.10	15%	24%
0.25	36%	45%
0.50	59%	62%
1.00	100%	100%

Table 2.4: Thread efficiency on one GTX 1080 GPU for inserting new grid points and for evaluating the 3D finite-difference stencil at different grid densities. The grid density is the average fraction of grid points that are allocated/used. Efficiencies are computed relative to the dense case where all chunks are fully occupied. The measurements were done on a 512^3 grid of `float` values using the present OpenFPM implementation.

time to insert and fill all 512^3 grid points of an initially empty sparse block grid of size 512^3 is measured. The points are inserted sequentially from $(0, 0, 0)$ to $(512, 512, 512)$. On the GPU, the entire procedure described in algorithm 4 is measured, consisting of (1) resetting and creating the insert queue, (2) collecting the insertions, and (3) flushing them. The performance test, runs multiple cycles of inserting all points, removing all points again, and inserting them again to control the retention of the internal data structures and memory allocation overhead. The results are given in Table 2.5 for the first insertion cycle, the second cycle, and all subsequent cycles.

As expected, the first insertion cycle is always the slowest because all queues and buffers are initially allocated. This overhead is independent of the speed of the GPU used, as can be seen by comparing the times on the Nvidia GTX 1080 with those on the lower-tier GTX 1650 Ti. The higher speed of the GTX 1080 only shows after three or more cycles of insertions. The table also shows that two cycles are required to retain all buffers, whereas, on the CPU, they are retained after the first cycle. Once all buffers are allocated, the OpenFPM implementation of the present sparse block grid design is about a factor of two faster than OpenVDB on the same CPU.

Insertion cycle:	1	2	3 or more
OpenFPM CPU	0.803	0.295	0.295
OpenFPM GPU (GTX 1080)	0.34	0.17	0.012
OpenFPM GPU (GTX 1650)	0.30	0.19	0.037
OpenVDB (<code>setValue</code>)	0.86	0.68	0.68

Table 2.5: Runtime in seconds for inserting 512^3 points of a sparse block grid in sequential order. We compare the current OpenFPM implementation on the CPU and two different GPUs with the CPU implementation in OpenVDB [61]. over three complete cycles of insertions. Inserting points in OpenVDB is done using the function `setValue`.

2.2.15 GPU code running on CPU

This section shows, how given the abstractions presented before and a few additional ones, the GPU code can run on many-core CPUs.

Cuda’s calls for memory management are all hidden in the allocators, as seen in section 2.1. While data structures allocate and destruct memory on the device (GPU) and host (CPU) transparently. It remains to abstract kernel launch. In order to do this, there is the macro `CUDA_LAUNCH`. Listing 2.28 shows how a regular CUDA kernel launch is transformed using this macro. The first argument is the kernel’s name, and the second is the GPU iterator indicating the number and size of the workgroups. The subsequent arguments are the arguments of the kernel. The macro gives the possibility to hide the CUDA-specific syntax. Additionally, the macro allows us to switch from an asynchronous CUDA kernel launch to a synchronous one with error checking in case of debugging.

Listing 2.28: `CUDA_LAUNCH`

```

1
2 auto part = vd.getDomainIteratorGPU(32);
3
4 // Normal CUDA kernel launch
5 kernel<<<part.wthr,part.thr>>>(vd.toKernel());
6
7 // The same using the macro
8 CUDA_LAUNCH(kernel,part,vd.toKernel())

```

The macro executes the kernel function for every workgroup and thread inside each workgroup. We emulate on CPU two functionalities: one is `__syncthreads` the other are atomic operations. `__syncthreads` function is emulated using a fast context switching without explicit threads. A fast context switch is a way to stop the normal execution of a function or a piece of code, save the status of the current context and resume it later. While saving the status of one context, we also switch to another context resuming its execution. While this can be done with standard threads, the context switches of such types are costly because they involve the operating system. A faster way is to use assembly instruction to code a function that operates a switch directly. A non-inlinable function call

generates at assembly level the instruction `call` that saves on stack the address of the assembly instruction line after the instruction `call`. In order to operate a switch directly, this function must first save the address of the next assembly instruction after the assembly instruction `call`. This indicates where the execution has been stopped and should resume once we want to go back to this context.

In order to have the possibility to jump back to a context we are leaving, it is necessary to save the registers and stack pointer. To switch back or move to another context, we restore the registers and stack pointer previously saved, as final step the CPU executes a direct jump with the assembly `jmp` to the saved address in which the execution has been previously stopped. This functionality must be implemented for every architecture and ABI in assembly. Fortunately, the boost library provides such functionality with `boost::context` for many architectures. We use such a library to implement fast context switching and construct on top a fully functional CUDA `__syncthreads()` function.

Also, atomic operation in the kernel are emulated. When there is only one thread, atomic operations like summation minimum and maximum are equivalent to standard non-atomic operations. Primitive operations explained in section 2.2.14 are implemented on a single-core CPU or wrapped around an already existing implementation from some library.

The actual implementation has two backend: one sequential where one thread sequentially execute every block, and one OpenMP based where one thread execute one block at a time and blocks are assigned to CPU threads.

Chapter 3

Space decomposition and distribution

Until here, the data structures presented were single-node or shared memory. This section extends them to distributed data structures figure 3.1. Because this section focus on N-dimensional simulation distributed data structures, it is necessary to introduce the concept of the N-dimensional simulation domain needed to develop domain-specific data structures. As an example consider a two dimensional grid 100×100 , when defined in a simulation domain from $(0.0, 0.0)$ to $(1.0, 1.0)$ it leads to a spacing of $(0.01, 0.01)$. The second important point to address is data distribution. In order to have extensible code, it is necessary to create a generic framework code able to convert a shared memory data structure into a distributed one. In order to do so, two components are introduced, one is the space decomposition module, and the second is a serialization and communication module.

3.1 Space Decomposition

The space decomposition module divides the space into sub-domains and assigns them to processors. In particular, given a simulation box, this module has to create a set of boxes that define the computational domain for a particular process, GPU, or computational device. In figure 3.2 there is an example of a two-dimensional box decomposed in 3 processors. A processor domain indicated with one color is composed by a union of boxes with edges parallel to the principal axis. Boxes are not forced to create a connected domain, but they are forced to be parallel to the principal axes at the present writing. The motivation behind this constrain lies in specific structures like distributed grid containers not working very well with non-axis parallel decomposition. Non-parallel axis decomposition makes the shape of every patch very complex. Each row requires a different starting and end point and the ghost shape also becomes non-trivial and potentially variable on each row. OpenFPM internally

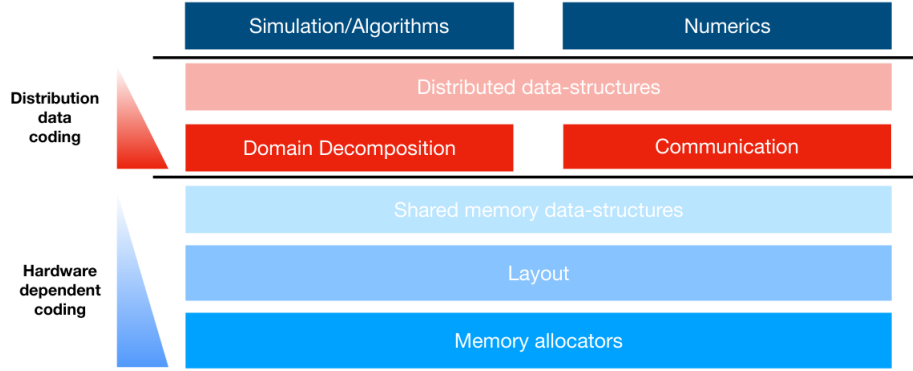


Figure 3.1: Distribution abstraction and toward distributed data-structures

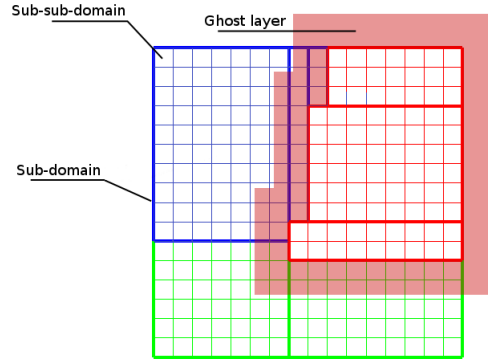


Figure 3.2: Domain decomposition in OpenFPM. The computational domain is decomposed into Cartesian sub-sub-domains (small squares) assigned to processors (colors). After the assignment, cuboidal blocks of sub-sub-domains are merged to form the larger sub-domains (bold lines). One processor may have more than one subdomain. Sub-domain borders at processor boundaries are extended by a ghost layer (shaded area, shown exemplarily for the red processor) to provide all data required for computations locally.

reserves the possibility of having non-axis parallel decomposition specialized for particular distributed containers like particles, but all domain decompositions implemented at the time of writing use parallel axis.

The domain decomposition process in OpenFPM is divided into three phases: *decomposition*, *distribution*, and *sub-domain creation* as summarized in algorithm 5. The decomposition stage divides the physical domain into sub-sub-domains (small squares in figure 3.2). In figure 3.2, the sub-sub-domains are arranged in a cartesian way, but generally, this is not necessarily the case. The number of sub-sub-domains generated is as large as the number of processors, but this number can be larger for certain decomposition algorithms like graph-based. For others like orthogonal recursive bisection, the number is generally kept equal to the number of processors. The number of sub-sub-domains is either specified by the user program or chosen automatically by OpenFPM. In the latter case, OpenFPM queries the distribution algorithm to get a default number. The number of sub-sub-domains determines the granularity of the distribution algorithm in which to search for an optimal configuration.

Algorithm 5 Domain decomposition

- 1: **procedure** DOMAINDEC
 - 2: *Decomposition:*
 - 3: Divide the bounding box of the computational domain into sub-sub-domains, in figure 3.2 show a regular cartesian decomposition in sub-sub-domains. Create a graph where each sub-sub-domain is a vertex and edges indicate potential communication if the two sub-sub-domains are assigned to different processors
 - 4: *Distribution:*
 - 5: Distribute the sub-sub-domains across processors. By using a graph the natural way is to use a graph partitioning algorithm like Metis or ParMetis [62]. The assignment naturally defines processors boundaries
 - 6: *Sub-domain creation:*
 - 7: this stage operates decomposition optimizations like merging the sub-sub-domains in the same processor to form larger sub-domains with less intra-processor mesh ghost layers
-

The main structure that handles phase 1 is called `CartDecomposition` defined in listing 3.1

Listing 3.1: Cartesian Decomposition

```

1  template<unsigned int dim,
2      typename T,
3      typename Memory,
4      template <typename> class layout_base,
5      typename Distribution>
6  class CartDecomposition

```

- `dim` indicates the dimensionality of the space to decompose

- **T** Precision of the space to decompose float or double.
- **Memory** This parameter is the memory type used to store information, this parameter is passed to the multiple internal `openfpm::vector` that requires to run on multi-hardware.
- **layout_base** The layout used by this structure to store information. This parameter is passed to the multiple internal `openfpm::vector` used to store information and for which it is required to run computation on different hardware.
- **Distribution** The distribution algorithm used to distribute sub-sub-domains

The distribution object handles the second phase of the space decomposition and must implement a well-defined interface to interact with `CartDecomposition`. Fixing the interaction between `CartDecomposition` and the distribution object requires encapsulating the algorithm in our distribution class. Doing this gives the possibility to extend `CartDecomposition` with an arbitrary number of distribution algorithms. OpenFPM contains a wrappers for Parmetis graph-based algorithms, Metis graph-based algorithms, space-filling curve, and trivial assignment if the number of sub-sub-domains is equal to the number of processors. A random distribution algorithm is also implemented to check the robustness of `CartDecomposition` to work on any decomposition.

3.2 Distribution algorithm selection and balancing

The class that encapsulates the distribution algorithm must be assignable and implements a pre-established API defined in listing 3.2

Listing 3.2: Distribution

```

1
2 void createCartGraph(grid_sm<dim, void> & grid, Box<dim, T> & dom)
3 void decompose()
4 void refine()
5 void redecompose()
6 float getUnbalance()
7 void getSubSubDomainPosition(size_t id, T (&pos)[dim])
8 void getSubSubDomainBox(size_t id, T (&pos)[dim])
9 inline void setComputationCost(size_t id, size_t weight)
10 size_t getSubSubDomainComputationCost(size_t id)
11 void setMigrationCost(size_t id, size_t migration)
12 void setCommunicationCost(size_t v_id, size_t e, size_t communication)
13 bool weightsAreUsed()
14 size_t getProcessorLoad()
15 size_t getNSubSubDomains() const
16 size_t getNOwnerSubSubDomains() const
17 size_t getOwnerSubSubDomain(size_t id) const
18 size_t getNSubSubDomainNeighbors(size_t id)
19 void write(const std::string & file)
20 void setDistTol(double tol)

```

Without going into the details of every single methods, the API is composed by: **createCartGraph** creates a set of unassigned sub-sub-domains with a suggested granularity given by grid. **decompose** creates the sub-sub-domains. **refine** does a local redistribution, of the sub-sub-domains. **redecompose** creates another assignment where non local movements are allowed, but at the same time tries to keep the new distribution overlapping as much as possible with the old one. The methods, **setComputationCost**, **getSubSubDomainComputationCost** are functions to get/set the computational load of each sub-sub-domain. The method **setMigrationCost** sets the cost in migration for each sub-sub-domain in case a sub-sub-domain is reassigned, and **setCommunicationCost** set the cost in communication in case two adjacent sub-sub-domain are assigned to two different processors. **getProcessorLoad()** gives the total load for the calling processor once decomposition/distribution/optimization is completed. The method **getNSubSubDomains()** gives the number of sub-sub-domain in the full simulation domain. **getNOwnerSubSubDomains()** gives the number of sub-sub domains owned by the calling processor. **getOwnerSubSubDomain** gives the list of the sub-sub domains owned by the calling processor, **getNSubSubDomainNeighbors** gives the neighborhood sub-sub domains for a given sub-sub-domain. **setDistTol** sets the maximum tolerated unbalance for which the algorithm is considered converged. This API is general enough to handle different geometrical decompositions like orthogonal recursive bisection or space-filling curve and graph-based decompositions.

There is no trivial assignment in graph-based decompositions because the number of sub-sub-domain (small-squares in 3.2) is larger than the number of processors (colors in figure 3.2). Instead, the additional degree of freedom is used to improve load balance and to reduce communication overhead. An optimal mapping must ensure that each processor receives the same amount of computational work (in terms of wall-clock time), and at the same time, the total amount of inter-processor communication is minimized.

Graph-partitioning models this problem very well. Each sub-sub-domain is a vertex of the graph and an undirected edge between two vertices represents a communication payload between the respective sub-sub-domains through their overlapping ghost areas. In order to account for varying particle density, each vertex has a computational cost c_i , proportional to the total amount of computing time contained in the sub-sub-domain i . The edges $e_{i,j}$ between sub-sub-domains i and j have a weight proportional to the time used in exchanging information between sub-sub-domains.

In an optimal assignment, two conditions must be satisfied at the same time. The sum of the vertex weights in a partition assigned to processor p must be equal to the sum of the vertex weights in a partition assigned to processor q . The sum of the weights of all edges connecting two different processors sub-sub-domains is minimal. Finding the global optimum solution to this problem is NP-hard and cannot be efficiently computed. Several publicly available libraries compute approximate sub-optimal solutions to this problem in a reasonable time.

3.2.1 Sub-domain creation and decomposition optimization

The last step of the domain decomposition tries to perform optimizations on the decomposition whenever possible. In the case of cartesian sub-sub-domain decomposition, once the distribution assigns them to processors, it tries to merge sub-sub-domains assigned to the same processor into larger sub-domains (bold lines in figure 3.2). Merge of sub-sub-domains reduces the total ghost-layer volume since sub-sub-domains on the same processor do not require a ghost layer between them. While this step is not so crucial for particles, for meshes is an important step. Ghost layers are required on all sub-domain boundaries for fast, structured neighborhood access. Therefore, the goal of sub-domain creation is to merge sub-sub-domains such that the minimum number of sub-domains with the smallest surface-to-volume ratio is created on each processor.

For example, if we have a grid of sub-sub-domain assigned to processors, the sub-domain optimization and creation algorithm runs in parallel on each processor. It starts from the first (by indexing order) sub-sub-domain on that processor and uses it as a seed. It then extends its boundaries uniformly in all directions. For example, the box is enlarged in two dimensions by shifting the border by one sub-sub-domain in direction $+X$, $+Y$, $-X$, $-Y$. This procedure is iterated until the sub-domain border reaches an inter-processor boundary or the end of the simulation domain. When it is not possible to merge any more sub-sub-domains in any direction, a sub-domain is created from all merged sub-sub-domains. The process continues choosing the next (by indexing order) unassigned sub-sub-domain at the border of the just-created sub-domain and repeats until all sub-sub-domains have been assigned to some sub-domain. Despite not finding the optimal solution, i.e., the one with the smallest number of minimum-surface sub-domains, this greedy heuristic is perfectly parallel and finds a sub-solution in linear time in the number of sub-sub-domains.

3.2.2 Internal/External ghost boxes, local internal/external ghost boxes

Once the sub-domains are created, like in figure 3.2, each processor has a list of boxes that compose the processor domain. In general, most types of computation require reading information from the neighborhood processors. Figure 3.2 show a processor domain in red with its ghost area in red. The ghost area is again composed of a list of boxes called external ghost boxes. In the following, we will formalize the terminology about internal/external ghost boxes and local internal/external ghost boxes.

Internal/External ghost box We consider two sub-domains A and B identified by two boxes axis parallel in two different processors called respectively i and j . A is the sub-domain for processor i , and B is the sub-domain for processor j . If we extend A by a ghost area, intersect A with B , and the intersection is different from zero, the intersection is again a box with axis-parallel. We call

that box **external ghost box** from the perspective of i and **internal ghost box** from the perspective of j . This means that there is always a correspondent internal ghost box in a different processor for each external ghost box.

We can also do the same procedure for the sub-domain B extending it by a ghost area. In that case if we have a non-zero intersection, we call the resulting intersection box **internal ghost box** from the perspective of i and **external ghost box** from the perspective of j . The extension procedure is non-symmetric. Extending A to B generates different boxes than extending B to A figure 3.3.

local internal/external ghost box From figure 3.2, we see how the processor in red has three subdomains. This means that intersection between subdomains can happen within the same processor, in this case these boxes are called local internal/external ghost boxes.

Boundary conditions and particular cases Boundary conditions affect the construction of the internal and external ghost boxes. A particular case is periodic boundary conditions. In this case, a sub-domain can intersect with itself. The terminology still applies, with the difference that sub-domain A and B are now the same. We also notice that intersection by periodicity generates an internal ghost box with margins different from the external ghost box like shown in figure 3.3.

Limits on the ghost boxes The only limitation imposed by the ghost boxes is that the ghost boxes must be all axis parallel. There is no limitation on numbers, or in extension like shown in figure 3.4.

performance

The decomposition distribution and optimization algorithm must be reasonably fast in finding a solution that balances the processor's computational load. We have to consider that the time spent finding such a solution is not used to run the simulation. For this reason, the decomposition distribution and optimization must be seen as a secondary problem. If the simulation is going to balance frequently, the time spent in balancing must be justified by a speedup of the simulation. An exception can be made for all problems where we have a simulation with static geometry. In this case, the decomposition is balanced only at the beginning and remains static over time. In this case, because the balancing algorithm runs only once at the beginning of the simulation, a better model providing balance and reduced communications at the same time could be essential for the total runtime of the simulation.

In our experiments, using graph-based decomposition, out of the total time taken for domain decomposition, 94% to 99% is consumed by ParMetis in the *distribution* step. This fraction was stable across all of our benchmarks. The *decomposition* step always took less than 1% of the total time, and the *sub-domain*

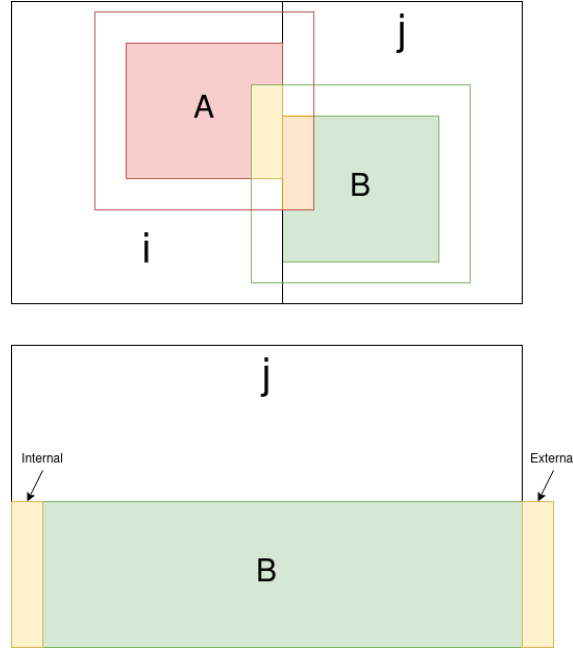


Figure 3.3: **TOP** In red, we see the domain A in green, the domain B. The red line indicates domain A with the ghost, while the green line indicates domain B with the ghost area. The orange box is the intersection of A + ghost with B, while the yellow box is the intersection of B + ghost with A. The orange box is the internal ghost box from the perspective of B and external from the perspective of A, while the yellow is external from the perspective of B and internal from the perspective of A. **BOTTOM** An example where periodicity produce a sub-domain to intersect with itself and creating self internal/external ghost boxes

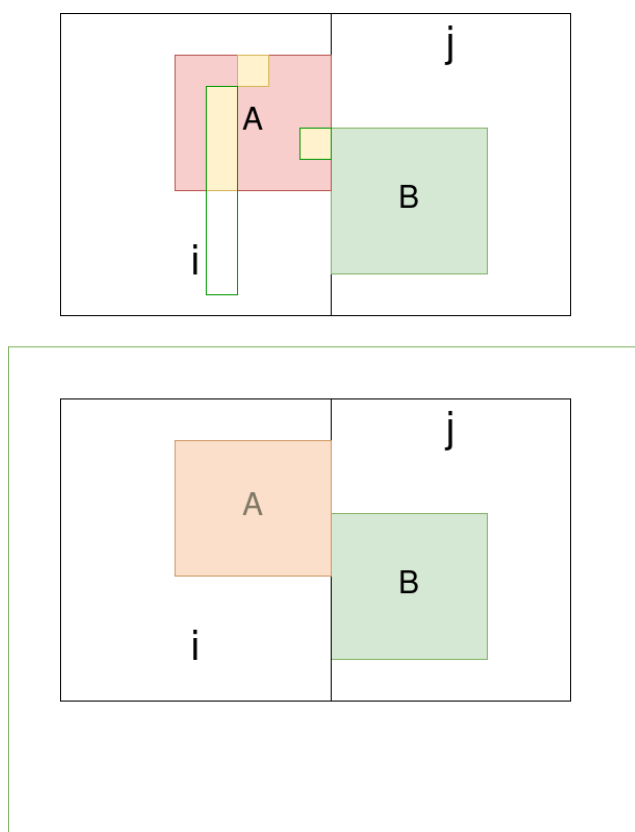


Figure 3.4: Ghost limits

creation step less than 5% of the total time. The time required by ParMetis, therefore, dominates the overall runtime performance of the OpenFPM domain decomposition.

3.2.3 Serialization

Before introducing the communication layer in OpenFPM, the thesis must introduce the concept of serialization. Serialization is the capability to convert a data structure into a stream of bytes, while deserialization is the inverse operation of taking a stream of bytes and convert it back into a data structure. OpenFPM can serialize and deserialize general classes as soon as, given a class A, it exposes three member functions shown in listing 3.3.

Listing 3.3: Serialization interface

```

1
2 template<int ... prp> inline void packRequest(size_t & req) const
3
4 template<int ... prp> inline void pack(ExtPreAlloc<S> & mem,
5                                     Pack_stat & sts) const
6
7 template<int ... prp> inline void unpack(ExtPreAlloc<S> & mem,
8                                     Unpack_stat & ps)
```

The first function is used to query the amount of memory required to serialize the data structure. It takes a long unsigned integer as an argument where the function sum the number of bytes required to serialize the data structure. The second function `pack` is used to load the data structure into a contiguous chunk of memory. The first argument of the method is a memory object, as seen in section 2.1. In this memory object, we save the raw data coming from the serialization process. The second argument `sts` contains a counter of the filled byte in the memory. The `mem` object can be re-used to serialize multiple objects contiguously. The function `unpack` does the inverse operation of the function `pack`; it takes the memory object where previously was saved the data and constructs the original data structure. Like `sts`, `ps` contain a counter of the bytes used to deserialize the original data structure. We mention that the template parameter `S` can be `CudaMemory` or `HeapMemory` depending if the `pack/unpack` method contain CPU or GPU serialization/deserialization code.

The variadic template `prp` is a list of properties to serialize. Thanks to the use of tuples, we can restrict the serialization to a set of properties. In partial serialization, we can decide to serialize only part of the information contained by the structure, selecting a list of properties.

Serialization of containers

Containers like vectors, grids, and sparse grids accept template parameters where the template aggregate parameter can contain in theory everything, including any nested containers like the one expressed in listing 3.4. Such structure is visualized in a tree-based way like in figure 3.5.

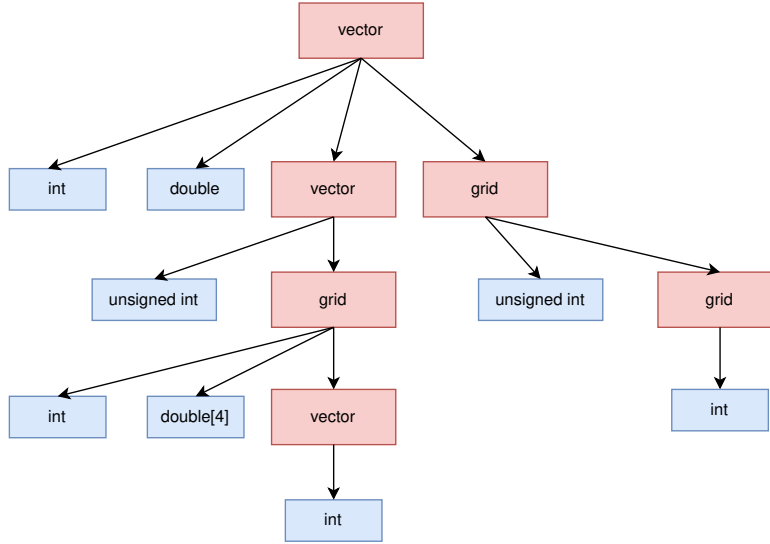


Figure 3.5: Tree structure of nested containers in listing 3.4

Listing 3.4: vector type with nested structure

```

1  vector<aggregate<int,
2    double,
3    vector<aggregate<
4      unsigned int,
5      grid<aggregate<
6        int,
7        double[4],
8        openfpm::vector<aggregate<int>>
9      >
10    >
11  >,
12  grid<aggregate<unsigned int,
13    grid<aggregate<int>>
14  >
15  >
16  >
17  >

```

In the case each node of the tree either implement the interface in listing 3.3 either can be serialized with a swallow memory copy (like a POD object that does not contain any pointers), we do not need to construct any serialization code. The serialization code is automatically generated without user intervention. The generation is possible because the aggregate can be analyzed at compile-time with meta-code and traversed to discover if one or more types in the tuple list implement the interface 3.3 or can be safely copied with a swallow memory copy. During the analysis the code is generated at compile-time accordingly to the feature of the tree node.

In figure 3.5 every red node implements the serialization interface, while blue nodes do not. Every red node in the tree, inside the functions **packRequest**, contains a meta-code to analyze if one of their children is a red node. If none of

the children is a red node, the memory required to serialize is calculated from the number of elements the structure contains, multiplied by each type's size. This size in this special case is equivalent to the size in bytes of the aggregate. If a red node is instead present, a for loop over its elements is generated. Inside the loop, for every blue node, the counter is incremented by the size of the type (unsigned int = 4 bytes, double = 8 bytes), and for every red node, a call to **packRequest** is generated passing the counter. The method **packRequest** triggers the analysis of its children for the nested container again. The process ends when a container does not have red children or, more practically, once a container does not have other nested containers. The pack follows the same concept as packRequest, differing only in the operation performed inside. While **packRequest** count the size of the buffer to serialize, **pack** fills the buffer with raw data. The generation of the code to serialize the structure in figure 3.5 leads to the automatically generated algorithm in listing 6. Equivalently, the generation of the packing code lead to listing 7.

Algorithm 6 Serialization code generation for **packRequest**

```

1: for all elements in vector(1) do
2:   increment the counter by 4 (blue node int)
3:   increment the counter by 8 (blue node double)
4:   for all elements in vector(2) do
5:     increment by 4
6:     for all elements in grid(2) do
7:       increment the counter by 4 (blue node int)
8:       increment the counter by 4*8 (blue node double[4])
9:       increment the counter by N*4 (red node vector(3)) where N is the
       number of elements in the vector
10:  for all elements in grid(1) do
11:    increment the counter by 4 (blue node unsigned int)
12:    increment the counter by N*4 where N is the number of elements in
    the grid

```

The variadic template **prp** selects which properties we want to serialize. An empty list indicates to serialize all properties. At present, in the case of nested containers, we can select only the properties in the immediate level below. We are not able to select properties in the deeper levels. In the case of the multiple levels, all properties from the second level and below are selected.

The meta-code to analyze the type and select between generating the code to pack a primitive or static array versus calling the pack and packRequest function is modularized in an external object called packer exposing static methods. An example of how it can be used is given in listing 3.5

Listing 3.5: Packer usage

```

1 Packer<float,HeapMemory>::pack(mem,9.0f,sts);
2 Packer<Point_test<float>,HeapMemory>::pack(mem,p,sts);
3 Packer<openfpm::vector<Point_test<float>>,HeapMemory>

```

Algorithm 7 Serialization code generation for **pack**

```

1: for all elements in vector(1): i do
2:   pack the integer i (blue node int)
3:   pack the double i (blue node double)
4:   for all elements in vector(2): j do
5:     pack the unsigned int
6:     for all elements in grid(2): k do
7:       pack integer k (blue node int)
8:       pack the 4 doubles (blue node double[4])
9:       memory copy all the N integers of the vector (red node vector(3))
10:  for all elements in grid(1) do
11:    increment the counter by 4 (blue node unsigned int)
12:    increment the counter by N*4 where N is the number of elements in
    the grid

```

```

4      ::pack<pt::x,pt::v>(mem,v,sts);
5 Packer<grid_cpu<3,Point_test<float>>,HeapMemory>
6      ::pack<pt::x,pt::v>(mem,g,sts);

```

For completeness, we write the entire logic of source code generation with a flow chart diagram in figure 3.6

Communication

Communications are fundamental in the context of distributed data structures. Most of the time, we have to exchange data with neighborhood processors and in certain situations with non-neighborhood processors. At the base of OpenFPM, there is the Message Passing Interface API and a library implementation conformant to such API. Although OpenFPM makes use of MPI, the API is not exposed directly. An object called **VCluster** provides its functionality. **VCluster** represents a group of processes. Either all processes like MPI_COMM_WORLD or a subset. Because the distributed data structures operate communications storing or using a reference to this object, tuning the **VCluster**, it is possible to restrict the distributed data structure to work on a sub-set of processes. Another important reason not to expose MPI and hide it behind **VCluster** is the possibility to replace an MPI library with other communication libraries like Unified Communication X (UCX) or NVIDIA communication collective library (NCCL).

VCluster provides both type-unaware API and type aware API for communication. Type unaware communication API is a set of methods to send and receive plain data as a stream of bytes. Type-aware communication API is a set of methods to communicate high-level structures/containers like a sparse grid, a vector, or a nested container like a vector of grids. We have to consider how to deal with the transmission of complex structures with nested pointers like in a vector of grid or a sparse grid. Type-aware communication API contains templated functions that accept any type. They rely on the concept

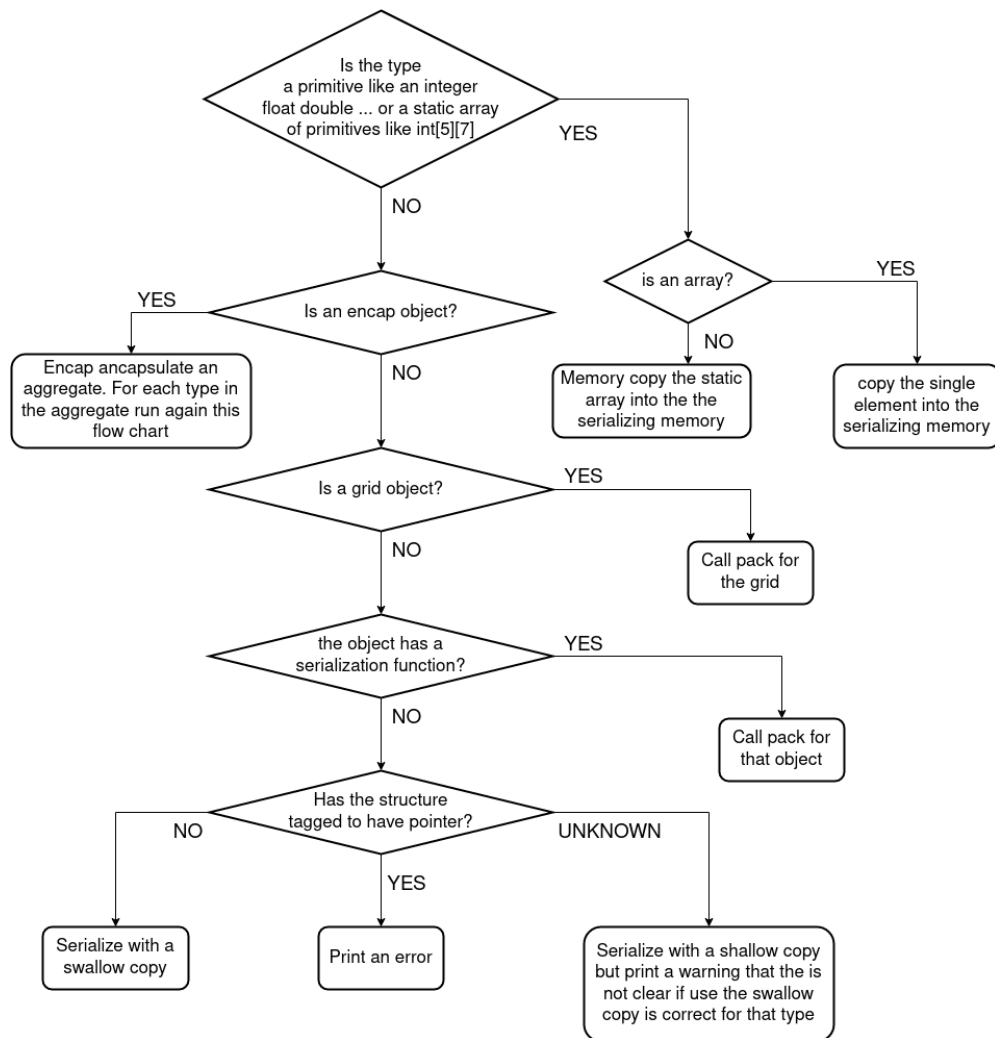


Figure 3.6: Logic to serialize nested containers

of serialization and deserialization of complex data structure explained in section 3.2.3. In a nutshell, type aware communications serialize the structure into a byte stream, and the byte stream is then transported to the destination and deserialized there. The actual communication primitives implemented in type aware and type unaware are: the point-to-point send and receive, allGather, and scatter. Broadcast and reductions like summation, maximum, and minimum are supported in a type aware way only for primitives types like integers or floating-point numbers of any size. All of them are asynchronous by default and can be queued and synchronized before executing code that depends on it. An example of type aware reductions is given in listing 3.7.

VCluster also implements communication patterns like dynamic sparse data exchange, where the senders do not have to know the communication pattern a priori or the size of the message a priori. Standard point-to-point communication primitives require a priori knowledge of the communication pattern for all processors. In standard MPI programming, if process A sends a message to process B, process B must be aware of such communication and explicitly call a receive. In situations where the information of the data structures must be redistributed accordingly to a new decomposition with an unknown global pattern, such knowledge is not available a priori. Dynamic sparse data exchange methods work on type aware and type unaware objects. All the sends are queued in this type of pattern, and the method continuously probes for receiving messages. To detect all communications have been completed across processors, the methods use an asynchronous MPI barrier internally.

There are several variants of the default implementation for dynamic sparse data exchange, synchronous and asynchronous implementation. Additionally, depending on whether there is partial information or complete information about the size and pattern of the receiving messages, the implementation is changed internally dynamically. In case of type unaware communications, a user-supplied callback is given to the method. This function is called in order to allocate the memory necessary to receive every message. In the case of type-aware communications, this is not required. The only requirement is that the receiving object is resizable or can merge or fuse two structures.

The next section presents each implementation of synchronous Dynamic Sparse Data Exchange (DSDE) in OpenFPM.

Dynamic sparse data exchange (DSDE)

As mentioned in the previous section, **VCluster** implements DSDE for type aware and type unaware communication. This section goes into the implementation details regarding synchronous, type aware/unaware, and no partial/complete/information of the receiving messages. All such variations have the same API. Later, we will see how these specializations have different optimized communication patterns implemented on top of a standard MPI API. We start to look at the standard implementation of DSDE synchronous and type unaware in algorithm 8

DSDE in algorithm 8 use `MPI_Issend` to send data. `MPI_Issend` differs from

Algorithm 8 Dynamic sparse data exchange

Input: send arrays

- 1: **for all** sends arrays **do**
 - 2: Send the data using MPI_Issend
 - 3: **repeat**
 - 4: perform an MPI_Probe to check for incoming messages
 - 5: **if** We have an incoming message **then**
 - 6: Call the user-defined callback to allocate memory and get a pointer to the allocated memory
 - 7: receive the message with MPI_recv
 - 8: use MPI_Testall to check that all the sent messages with MPI_Issend have been received
 - 9: **if** All messages has been received **then**
 - 10: call MPI_Ibarrier to notify all the messages have been sent
 - 11: **until** MPI_Ibarrier is not reached/called by all processors
-

MPI_Issend. In the case of MPI_Issend, the operation is considered completed when the sending buffer is ready to be reused. The function returns independently that the message has been already received or not by the destination process. MPI_Issend instead is considered completed only when the other side has received the message. This behavior, results in the asynchronous barrier being hit if all messages have been received. Despite its dynamism, MPI_Ibarrier is a global synchronization point. If we know the communication is between neighborhood processors in the form of a single message, we can identify the processors from which to receive a priori. However, when the size of the message remains unknown, we can proceed as follow: We first send a message specifying the size of the incoming message, allocate the buffer, and finally queue the receive function, like shown in algorithm 9. In case we also know the message size, the algorithm reduces even more into algorithm 10

All the algorithm variations rely on the same API call shown in listing 3.6. The selection of the patterns presented before is based on the information given for the argument `n_recv, prc_recv, sz_recv`. If `n_recv` is set to zero, the receiving processors remain unknown even in number. In such case, no assumptions is made, and the first variation is chosen. In case `n_recv` is non-zero, `prc_recv` must be also filled. If `sz_recv` is filled with zeros the second variation is chosen. If `sz_recv` is filled with non-zero, then the third variation is used. Having only one call for all variations simplifies the logic of the code that uses these functions. The best communication algorithm is selected transparently based on the information we have at the moment. The methods in listing 3.6 is the lowest level call to dynamic data-sparse exchange. To complete the explanation of the parameters needed for DSDE: `n_send` is the numbers of messages to send, `sz` contain the size of each message to send and `prc` contains the destination process for each message. `ptr` contains a pointer to the message data `n_recv prc_recv` and `sz_recv`, as explained before, contain or does not contain information about

Algorithm 9 Dynamic sparse data exchange

Input: send arrays

- 1: **for all** Messages to send **do**
 - 2: Queue an MPI_Isend with the size of the message to the destination process
 - 3: **for all** processors we have to receive **do**
 - 4: queue an MPI_Irecv
 - 5: Wait to receive all messages
 - 6: **for all** Received messages **do**
 - 7: Call user defined function to allocate a buffer big enough to receive the message
 - 8: **for all** Messages to send **do**
 - 9: Send the message using MPI_Isend
 - 10: **for all** Receiving processors **do**
 - 11: Receive the messages using MPI_Irecv and the previous received message size
-

Algorithm 10 Dynamic sparse data exchange

Input: send arrays

- 1: **for all** Processes from which we have to receive **do**
 - 2: Call the user-defined function to allocate a buffer big enough to receive the message
 - 3: **for all** Messages to send **do**
 - 4: Queue an MPI_Isend
 - 5: **for all** Messages to receive **do**
 - 6: Queue an MPI_Irecv
 - 7: Wait for all messages to be received
-

the receiving messages and select the communication algorithm. `msg_alloc` is the callback used to allocate the receiving buffers, while the last argument `ptr_arg` is a general pointer passed to `msg_alloc` in case we want to pass arguments or data to the callback.

Listing 3.6: NBX call

```

1
2 void sendrecvMultipleMessagesNBX(size_t n_send , size_t sz[],
3                                 size_t prc[] , void * ptr[],
4                                 size_t n_recv,
5                                 size_t prc_recv[],
6                                 size_t sz_recv[] ,
7                                 void * (* msg_alloc)(...),
8                                 void * ptr_arg,
9                                 long int opt=NONE)

```

The equivalent type aware function for the DSDE `sendrecvMultipleMessagesNBX` is given by `SSendRecv`. In listing 3.8, it is possible to note that there are several simplifications compared to `sendrecvMultipleMessagesNBX`. `send` is a list (`openfpm::vector`) of objects `T` to be sent for each processor. The vector `prc_send`, store the destination processor number, `prc_recv` and `sz_recv`, if left empty, will be internally filled with the ids of the processors from which we have received, and the number of objects `T` received from each processor. `recv` is the receiving object, it can be again an `openfpm::vector<T>` or any objects of type `S` implementing the functionality `S.add(T)`.

Thanks to this generalization, we can call the function in very different ways. Consider the case where the class `A` respects the interface of serialization, we can call `SSendRecv` with `[T = A S = vector<A>]` or `[T = vector<vector<A>> and S = vector<A>]` or `[T = vector<vector<A>> and S = vector<vector<A>>]`. The first case means that we are sending one object `A` for each processor listed in `prc_send`. We collect all received objects in `recv` merging them into one list. In the second case, instead of sending one object `A` for each process, we are sending a set of objects `A` to each process. Each processor receives a set of objects `A` and collects them in `recv` merging them in one list. In the third case, we are sending a set of objects `A` for each processor and receiving a set of objects `A` from each processor. Every set in this case will not be merged in one list but it remains separated and stored into a vector of vectors of `A`. The last case can be considered equivalent to the first if we transform $A \rightarrow \text{vector} < A >$.

Type aware functions work on the capability of object `A` to be serialized and deserialized. An object `A` automatically generates serialization and deserialization functions if it respects the conditions in section 3.2.3. The approach works in general independently from the layout and the number of properties or nested structures. On the other hand, serializing structures is expensive and generates an extra copy from the data-structure into the sending buffer. In cases where the layout fits one or few messages to send, **VCluster** skip the serialization and send the data with zero-copy. Because the tree is known at compile-time, these optimizations are applied at compile-time.

The graph in figure 3.7 shows the compile-time optimizations in terms of code generation for each case in **SSendRecv**.

3.2.4 reductions

For the case of type-aware communication, instead, VCluster implements reductions like **max()** **min()** **sum()** that calculate the maximum, minimum, and sum across processors asynchronously. An example is given in listing 3.7. The code creates an object `vCluster`, two variables `f` and `i`, for which we want to calculate the maximum and the sum across processors. Calling **max()** and **sum()**, we pipe the operations asynchronously. The type is automatically inferred from the type of the variables. Once all the operations are queued a call to the method **execute()** calculate the **max()** and **sum()** consistently across all processors. After **execute()** `f` and `i` contain respectively the maximum and the summation across processors.

```

1
2 auto & v_cl = create_vcluster()
3
4 double f;
5 int i;
6
7 // f and i filled by each processor
8
9 v_cl.max(f);
10 v_cl.sum(i);
11 v_cl.execute();

```

Listing 3.7: Maximum and summations across processors

Listing 3.8: SSendRecv

```

1
2 template<typename T,
3         typename S,
4         template <typename> class layout_base = memory_traits_lin>
5 bool SSendRecv(openfpm::vector<T> & send,
6                S & recv,
7                openfpm::vector<size_t> & prc_send,
8                openfpm::vector<size_t> & prc_recv,
9                openfpm::vector<size_t> & sz_recv,
10               size_t opt = NONE)

```

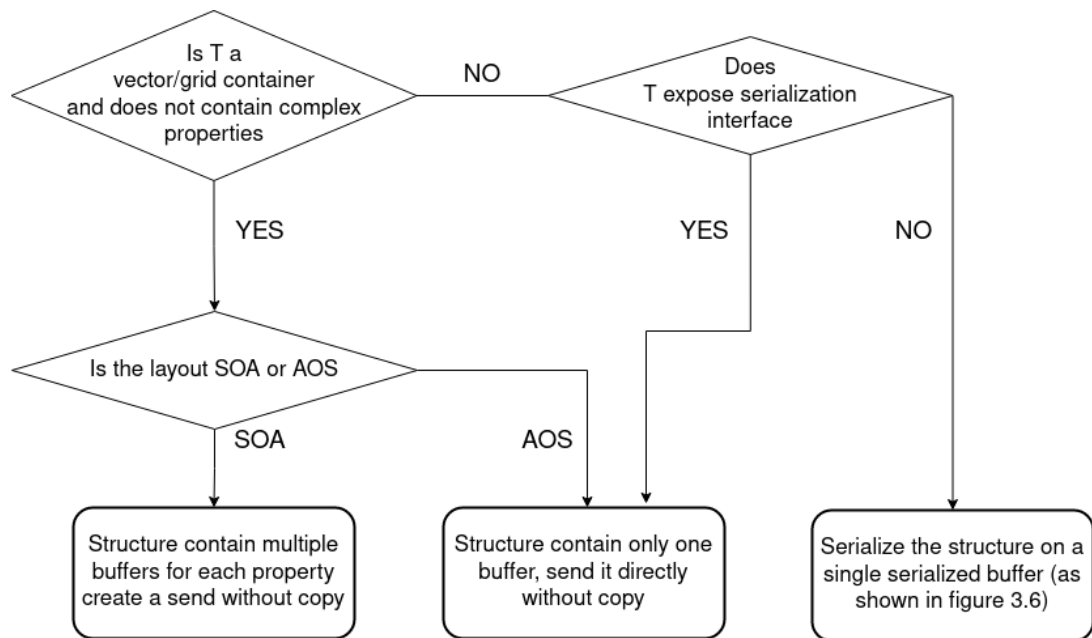


Figure 3.7: Optimizations done internally by SSendRecv

Chapter 4

Distributed data structures

4.1 Distributed grids and `grid_dist_id` as framework for distributed grid

The previous sections presented the shared memory data-structures module, space decomposition module, and communication module. This section presents how to construct space-based distributed data structures on top of these modules. The first structure the thesis presents is the distributed dense/sparse grids defined in listing 4.1

Listing 4.1: `grid_dist_id` definition

```
1 template<unsigned int dim,  
2         typename St,  
3         typename T,  
4         typename Decomposition = CartDecomposition<dim,St>,  
5         typename Memory=HeapMemory,  
6         typename device_grid=grid_cpu<dim,T> >  
7 class grid_dist_id  
8
```

- **dimensionality** the dimensionality of the distributed grid.
- **St** precision used to store or perform calculations in case of position and position informations: like grid spacing and domain in which the grid is defined.
- **T** the aggregate or tuple of properties the data-structure contain
- **Decomposition** which space decomposition module to use, modules are explained in section 3.1
- **Memory** Type of memory used, either HeapMemory or CudaMemory explained in section 2.1

- **device_grid** Type of grid used at single node level. This parameter selects a dense grid cpu explained in section 2.2.8 or the sparse grid cpu explained in section 2.2.14 or sparse grid on gpu explained in section 2.2.14. In general, a new completely different implementation is possible as soon as it respects a grid-like interface as explained in section 2.2.8, and exposes a serialization interface. This opens the possibility of wrapping an external data-structure coming from an external library and making it distributed, thereby importing its functionality.

The first key point of the `grid_dist_id` structure is handling both sparse and dense grids. The second key point is to keep this code transparent from hardware. In order to achieve this goal, we start noticing that the sparse grid interface is a superset of the dense grid interface. While in the dense, there is a **get** function to obtain a reference to a particular data point, in the sparse grids, there are three methods: **insert**, **get**, and **remove**. This means that if `grid_dist_id` must drive both it has to expose a super-set API with **insert**, **get**, and **remove** functions. As a consequence, this requires the dense grid to be extended to expose **insert**, **get**, and **remove** functions. The easiest way to extend the dense grid is map **get** and **insert** function to the standard **get** and the **remove** to a no-operation. The sparse grid has the additional restriction that the **get** function works only on reading, while the dense case works for read and write. While is possible to use the most restrictive convention, this would force the dense grids to be used with two functions **insert** and **get** rather than a single **get**. In order to keep only one function for the dense case, we declare the **get** function of the distributed grid with listing 4.2. Observing that the `loc_grid` in listing 4.1 contains a list of single node data-structures explained in the section 2.2. Declaring the return type as `decltype(loc_grid.get(v1.getSub()))` `template get<p>(v1.getKey())` means to infer the type (`decltype()`) from the underlying single core grid (`loc_grid.get(...)`). Such approach delegates the duty to determine the type, and consequently its constness or non constness, to the underlying single node data-structure. Because the return type of the **get** method in `grid_dist` adaptively inherits the constness from the underlying single node data structure, the method can be adapted to be used as read only or read /write method.

The definition of the function **get** in listing 4.2, is also general in regard to the key used to access a point (generic template parameter `bg_key`). We have to consider that the type used to access an element in a dense grid is different from one in the sparse grid or sparse grid on GPU. Distributed types use the type `grid_dist_key_dx` to access an element. This type contains two elements. The first is an integer indicating the patch created in a sub-domain (figure 3.2), returned by `v1.getSub()`, the second is the information to access an element within that patch number, returned by `v1.getKey()`. The type of the key is different between a dense grid and a sparse grid. The dense grid in 2/3 dimensions contains 2/3 integer numbers to access an element within the patch, while the sparse grid contains a chunk id and an integer as a linearized offset within the chunk, independent of dimension (figure 2.2). For this reason, the

type of the key to access an element within the patch must be templated with a generic place_holder (bg_key)

Listing 4.2: get function in grid_dist_id

```

1
2 template <unsigned int p, typename bg_key>
3 inline auto get(const grid_dist_key_dx<dim,bg_key> & v1)
4 -> decltype(loc_grid.get(v1.getSub()).template get<p>(v1.getKey()))

```

Similar generalizations done for the method **get** apply also to the iterator methods. For the single-core data-structures, the dense expose only one method to get an iterator across the points **getIterator(start,stop)** explained in section 2.2.14, while the sparse provides two **getIterator(start,stop)** and **getGridIterator(start,stop)** explained in section 2.2.14. To create a distributed version able to drive both APIs we must choose a superset, so the iterators for the distributed data structure must include two versions the **getDomainIterator(start,stop)** and **getDomainGridIterator(start,stop)**. Both methods return an iterator that moves transparently across all the patches, transforms the start and stop point into relative coordinates for the local patches excluding the ghost parts and calls the **getIterator** or **getGridIterator** on the patch to get a sub-patch iterator. As soon as the sub-patch iterator is complete, it moves to the next patch. Again, the sub-patch iterator return type is templated to have an adaptive code that works on both dense and sparse grids. The standard shared memory dense grid is extended to have a **getGridIterator** that maps to **getIterator**. We previously introduced also the concept of ghost, consequently there are two additional iterators that include ghost areas: **getDomainAndGhostIterator(start,stop)** and **getDomainAndGhostGridIterator(start,stop)**.

It remains to generate a generic **ghost_get** that works on dense and sparse versions. The next section is dedicated to writing such functions in a generic way, given the previous abstractions introduced.

4.1.1 ghost_get as generic code

In section 3.2.2 the thesis introduced the concept of ghost areas and the concept of internal and external ghost boxes. This section explains how to construct a general **ghost_get** on top of these concepts.

Suppose we have an array containing all the internal ghost boxes for each sub-domain like explained in the decomposition section 3.2.2. Suppose also that all these boxes have been converted into grid-based units by dividing them with spacing. From these elements, algorithm 11 contain the pseudo-code for a generic **ghost_get**.

Algorithm 11 only requires an implementation for **pack** and **unpack** function for each of the single node data-structures, sparse or dense, CPU or GPU. These functions are the serialization functions explained in section 3.2.3 that abstract hardware implementation. Because of these abstractions, the **ghost_get** function remains independent from dense, sparse, or a particular hardware.

Algorithm 11 ghost_get

- 1: **for all** internal ghost box **do**
 - 2: Count how much information to serialize
 - 3: Allocate the necessary buffer to serialize the information
 - 4: **for all** internal ghost box **do**
 - 5: Serialize the information as indicated by the internal ghost with function pack 3.2.3
 - 6: Transport the serialized information to the destination process
 - 7: **for all** Received packages **do**
 - 8: Identify the destination patch and the corresponding external ghost-box (for each internal ghost box there is an external ghost box 3.2.2)
 - 9: Remove the points in the section indicated by the external ghost box. (Necessary for sparse, for dense is an empty function)
 - 10: Merge back the information using the unpack function in the destination grid.
-

4.1.2 Multi-hardware optimizations

The function in ghost_get written in algorithm 11 is general enough to be multi-hardware, sparse, and dense. However, it is necessary to write some steps asynchronously to achieve good performance to reduce latencies coming from synchronization between device and host. We have to consider that every time a kernel depends on some checking performed on the host, latencies are introduced because the next kernel has to wait for a signal from the host to run. For example, in an algorithm that has to converge with some tolerance, the calculated residual error must be transferred from GPU to CPU at the end of the iteration loop. The CPU still has to check the convergence of the algorithm and restarts the loop. Although this check requires a small amount of transfer, it results in additional latency for the next iteration. In order to make this operations more friendly for massively parallel architecture we will introduce several optimization only tailored for GPUs and not CPUs. At the end of the GPU optimization in section 4.1.6 the thesis will show how the CPU API easily adapt to the GPU.

4.1.3 Queue operations for massively parallel architecture

Line 1-2 and 4-5 of algorithm 11 requires to respectively count and serialize for each internal ghost-box the information the grid contains. Launching a kernel for every internal ghost box is inefficient because it would result in small kernels. It is better to launch one kernel that handles all the internal ghost boxes for each patch. This operation requires queueing all internal ghost boxes for each patch and running one kernel for counting and serializing the ghost on each patch. This transformation requires changing the algorithm 11, in particular lines 1-2 are transformed into algorithm 12 and lines 4-5 are transformed into algorithm 13.

Algorithm 12 Internal ghost box queued asynchronously for calculation

- 1: **for all** internal ghost box **do**
 - 2: Queue the internal ghost box to the corresponding patch/sub-domain
 - 3: **for all** sub-domains **do**
 - 4: call `packCalculate` to launch a kernel to calculate the amount of information we have to serialize given all the queued internal ghost boxes
-

Algorithm 13 Internal ghost box queued asynchronously for serialization

- 1: **for all** internal ghost box **do**
 - 2: Queue the internal ghost box to the corresponding patch/sub-domain
 - 3: **for all** sub-domains **do**
 - 4: call `packFinalize` to launch a kernel to serialize the information contained by the internal ghost boxes in the sending buffer
-

The same observation goes for deserialization. All receiving packages are queued on the respective patches as multiple requests to merge the receiving data. Once all the requests are queued, a kernel is launched on each patch to merge the data. These two phases are split in **unpack** and **removeAddUnpackFinalize** leading to the transformation of line 8-9-10-11 of algorithm 11 into algorithm 14.

Algorithm 14 Internal ghost box queued asynchronously for deserialization

- 1: **for all** Received packages **do**
 - 2: Identify the destination patch and the corresponding external ghost-box corresponding to the internal ghost-boxes (for each internal ghost box there is an external ghost box as explained in section 3.2.2)
 - 3: Queue the external ghost boxes with the request to remove the points inside these areas.(Necessary for sparse, for dense is an empty function)
 - 4: Queue the external ghost box with the request to merge the received information using the `unpack` function in the destination grid.
 - 5: **for all** sub-domains **do**
 - 6: call the function `removeAddUnpackFinalize` to execute all the requests queued in the previous loop
-

The transformations explained for serialization and deserialization make the algorithm 11 fit better for massively parallel hardware. Additionally, we will see in the next section how the transformed algorithm continues to work also with synchronously implemented **pack** and **unpack**, for the CPU case. Because the implementation of the methods of calculating the size, serializing, and deserializing the data-structures, or removing points, are implemented at the level of shared memory data-structures, the best case scenario is always selected based on case (CPU/GPU or dense/sparse). Such functions are going to be explained in the next section for the most challenging case, the sparse grid on GPU that

fully use the asynchronous pack/unpack for the serialization stage.

4.1.4 Asynchronous pack/unpack for sparse grid on GPU

Asynchronous pack and unpack has been implemented for the sparse grid on GPU mainly for its specialization on massively parallel architecture. This section is going to explain more in details how algorithms [12,13,14] work and their optimizations.

As we have seen before, serialization is composed of two phases. In the first phase, we calculate the size of the buffer needed to pack the information. In the second phase, the information is packed into the sending buffer.

In the first phase, to queue the serialization requests, there are three main functions in listing 4.3.

Listing 4.3: Function to calculate the sending buffer

```

1
2 void packReset()
3
4 template<int ... prp> inline
5 void packRequest(SparseGridGpu_iterator_sub<dim,self> & sub_it,
6                 size_t & req) const
7
8 template<int ... prp> inline
9 void packCalculate(size_t & req, mgpu::ofp_context_t &context)

```

The first member function **packReset** resets the pack counters, the second member function **packRequest** queues a request to pack, the final function **packCalculate** calculates the size of the buffer required to store the information. The variadic template argument indicates which properties to pack in order to serialize only one or more subtrees of the nested structure. The function **packRequest** accepts an iterator as a delimiter of the section of the grid we want to serialize. The entire calculation of the size is done in **packCalculate** on GPU. The **packCalculate** member function is implemented in algorithm 15.

Algorithm 15 calculates how many points each of the requests has to pack. We then group the counters to determine the size of each sending buffer. Additionally, the prefix sum calculated from the number of points each block contains, indicates the offset at which each chunk must be packed in the sending buffer. In particular, the package is structured like in figure 4.1 and will contain the following elements.

- **1** The number of chunks to pack
- **2** The offset of where in the grid the internal ghost box has been packed, and the size of the internal ghost box
- **3** An array containing for each chunk to be packed its chunk ID
- **4** An array containing for each chunk the number of points to be packed
- **5** An array containing for each point to pack the data of the selected properties. The layout of this array is a SoA as explained in section 2.2.5

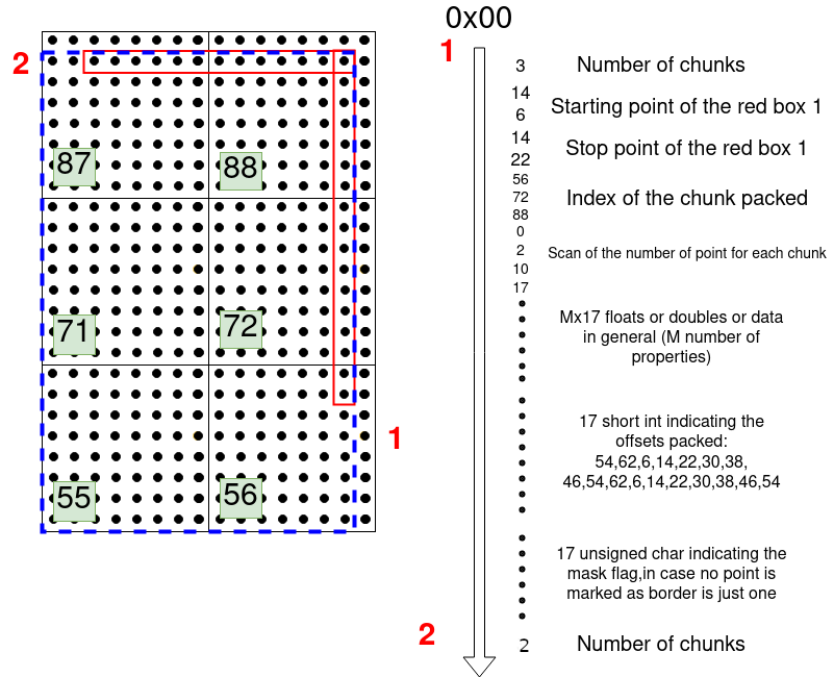


Figure 4.1: On the left, there is with a red rectangle the two internal ghost boxes we want to pack, the dotted blue rectangle indicates the limit of the patch. In particular, the size of the patch does not have to be a multiple of the chunk size. On the right column, we see how the information is packed. The arrow indicates the increasing memory offset in the stream created, the number on the right indicates the number stored in that offset, more on the right there is a small description on the number packed

Algorithm 15 packCalculate

-
- 1: Move the K queued internal ghost boxes to GPU, create K buffers of the size of N number of chunks
 - 2: Create an array of the size of number of boxes multiplied the number of blocks
 - 3: **kernel**:N=(chunks) workblocks, M=(points in chunk) threads
 - 4: **b**=block, **t**=thread
 - 5: K integers **bcount** initialized to zero.
 - 6: flag set to false
 - 7: synctreads
 - 8: Calculate the point **p** in coordinate
 - 9: **for all** j in the K request/internal ghost box **do**
 - 10: **if** p is inside the box j **then**
 - 11: increment atomically the counter **bcount[j]**
 - 12: synctreads
 - 13: Store the K counters in the K buffers at position **b**
 - 14: Perform a exclusive prefix sum operation on all K buffers
-

- **6** An array of short int containing for each packed point its position relative to the chunk
- **7** An array of char containing the mask value of each packed point

The mask is, in general, used to indicate additional flag properties. Each point has 8 bits of mask. In the actual implementation, the bit 0 indicates if a point in the chunk is filled or not. The bit 1 is used to indicate if a point is a border point or not.

After calculating the package size and allocating a buffer for each processor, the ghost_get function starts phase two, where the information must be packed in the allocated buffer. This is done with the pack functions **pack** and **packFinalize**, specified in listing 4.4. The pack call fills the information in point **1** and **2** in the item list. The filling of the data happens in **packFinalize**, the offsets for each chunk have been calculated by algorithm 15. The implementation of phase one and phase two is specific to each single node data structure. In figure 4.1 there is the implementation for the sparse grid GPU case. In the example, there are two internal ghost boxes packing information from a sparse-grid patch on GPU.

Listing 4.4: Function to pack information in the sending buffer

```

1
2 template<int ... prp>
3 void pack(ExtPreAlloc<S> & mem,
4           SparseGridGpu_iterator_sub<dim,self> & sub_it,
5           Pack_stat & sts)
6
7 template<int ... prp>
8 void packFinalize(ExtPreAlloc<S> & mem,
```

```

9      Pack_stat & sts,
10      int opt,
11      bool is_pack_remote)

```

The asynchronous unpack works in a similar way; it adds multiple requests to a queue before performing them. We will see in the next section how the unpacking procedure has several new challenges.

4.1.5 Unpack

Unpacking follows algorithm 14. Steps 3, 4, and 6 require to implement operations like initializing the queue with unpack operations and executing them in one kernel. Unpacking the information received from another processor imposes new challenges for specific data structures like the sparse grid on GPU. Figure 4.4 shows a typical example in two-dimensions where the chunks are miss-aligned. The miss-alignment introduces a map one-to-many between the chunks packed and the destination chunks. One packed chunk can map into four chunks in the destination grid in 2D and up to eight in 3D. The one-to-many map requires applying a map between the packed chunks and the destination grid. The map is constructed on the CPU and is uniquely determined for every patch pair exchanging information. It is the same size as one single chunk and contains for each point the index of the destination chunk (zero bottom-left, one bottom-right, two top-left, three top-right figure 4.3). This map quickly converts from the packed information's data layout to the destination grid's data layout.

Consider in figure 4.4 the yellow box, as explained in section 3.2.2, this is an internal ghost box from the perspective of the blue patch, while it is an external ghost box from the perspective of the yellow patch. In figure 4.4, in order to unpack the information in the yellow patch, we have first to convert the chunk-ids 56, 72 and 88 into the new chunk ids. If we look at chunk 56, this chunk intersects in general 4 chunks in the destination in particular using the notation (bottom-left,bottom-right,top-left,top-right) it intersects (X,X,X,0) where X means invalid chunk. The 72 intersects (X,0,X,30) and 88 (X,30,X,60).

This conversion is done in parallel for each chunk ID received, each thread in the kernel converts one chunk ID into the four destination chunk ids. Since the chunks with X will not be touched, they can remain uninitialized. After the conversion, the unpack operation performs a flush operation explained in section 2.4 with the converted indices, where the final data merging step is skipped. This constructs the array "merged index buffer" in figure 2.4. The last step is explicitly skipped because the data is compressed in the received buffer, and has a different format from the one required by the flush operation.

Once the second from last step in the flush operation produces a merge index buffer, it is possible to construct an additional map indicated in violet in figure 4.2. This buffer contain the index in which the chunks [X,X,X,0], [X,0,X,30], [X,30,X,60] are positioned in the merge index. This will result in [12,13,14,0], [15,1,16,8], [17,9,18,11]. Given also the unique merged sorted list in figure 2.4, it is possible finally to run a kernel that for each point in the received

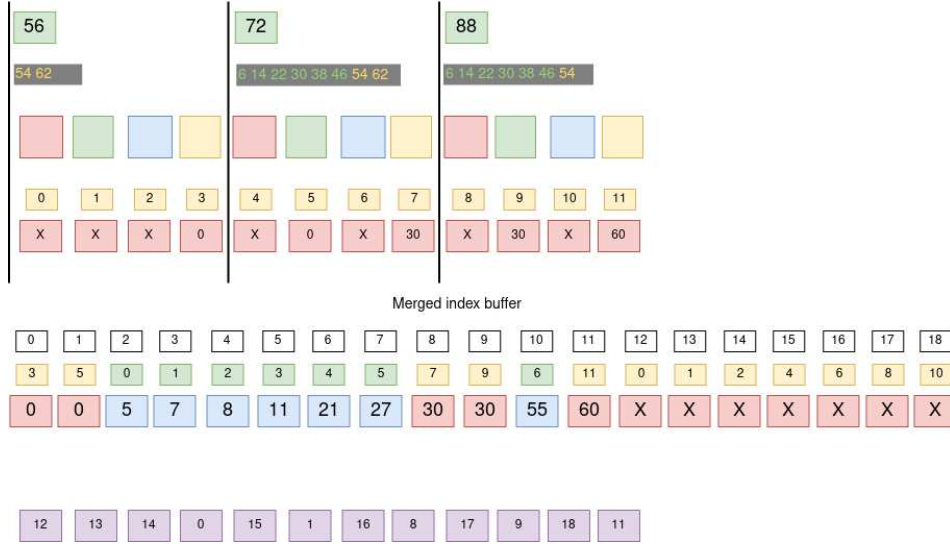


Figure 4.2: In the green box, there are the packed chunks, while under with small grey boxes, the offset packed for each chunk. The color indicates in which section of the map 4.4 the offset fall-off. Each chunk has four sections and four destination boxes. The merge index buffer is constructed using the flush function in figure 2.4, the violet array contain for each split chunk id $[X, X, X, 0][X, 0, X, 30][X, 30, X, 60]$, the position in the merge index array $[12, 13, 14, 0][15, 1, 16, 8][17, 9, 18, 11]$, and is constructed running a kernel on the merge index buffer

buffer places the data in the destination chunk and offset. The operations the kernel does are shown in figure 4.3

If there is an update of the ghost area where no data is added or could trigger the creation of new chunks, it is possible to recycle the maps calculated to unpack a packet more quickly. For this reason, each sparse internal grid on GPU keeps track of these maps and stores them until an external flush is called. Invalidating these maps with a flush will force at the next asynchronous pack/unpack operation the recalculation of the internal maps. This means in general that the first **ghost_get** is expected to be slower than the next subsequent **ghost_get**.

4.1.6 CPU homogenization to the GPU

As the thesis mentioned before, it is not needed in the case of CPU to create all the queue infrastructure and use asynchronous pack/unpack. However the **ghost_get** has been adapted to use asynchronous pack/unpack. Observing that mapping the method **packReset**, **packCalculate**, and **packFinalize** into a no operation, it is possible to re-map an asynchronous packing into a synchronous one. Since **packRequest**, and **pack** are implemented at the level of shared

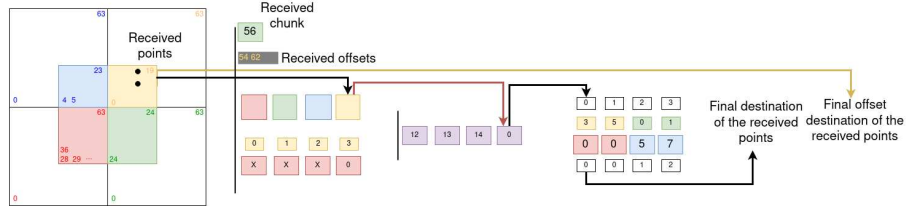


Figure 4.3: This figure shows how it is possible to use the constructed maps to recover the destination of the received points. The received chunk 56 is split into four parts as also explained in figure 4.2 [X,X,X,0]. The map also quickly converts the Received offsets (62 to 18 and 54 to 10). The converted offsets can be used directly as offsets in the destination chunk (yellow arrow). These four parts have a position in the merge index numbers 12,13,14,0 in violet as shown also in figure 4.2. The merge index reported from figure 4.2 has at the bottom a map (bottom white boxes) that indicate where each merged chunk id lives in the unique merge index of figure 2.4. The bottom white box index is finally the chunk position for the received point

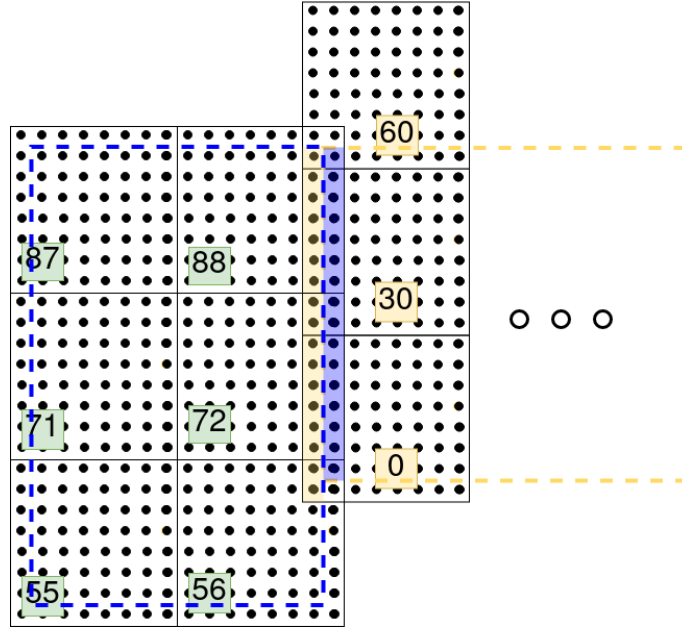


Figure 4.4: In this figure, there are two patches, blue and yellow, dotted with a ghost area of one. The patches do not include the ghost, as we can see the blue and yellow rectangles do not overlap. The starting point of the blue patch is (1,1), and the yellow area is the ghost part of the yellow patch intersecting the blue patch, the blue area is the ghost area of the blue patch intersecting the yellow patch.

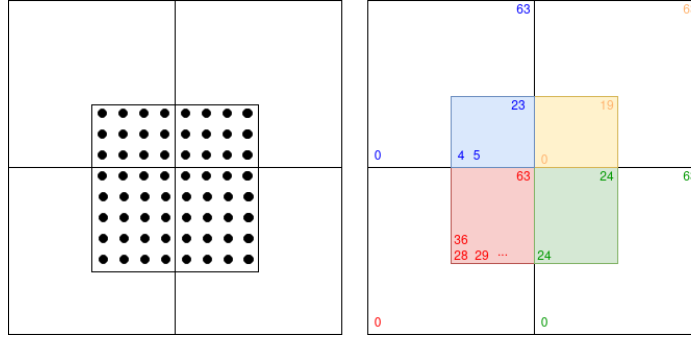


Figure 4.5: Conversion maps between two miss-aligned grids, on the left, There are the chunks of the original grid from where chunks are packed. The empty boxes indicate the chunk alignment of a destination grid, where we see a miss-alignment of the chunks. Each point of the map 8x8 contain two pieces of information a color index 0=red, 1=green, 3=blue, 4=yellow and the offset in the destination chunk

memory data-structures, it is possible to switch to the most suited implementation on a case-by-case basis, leaving the code in the distributed data-structure transparent from hardware (CPU/GPU) or sparsity (sparse/dense).

4.1.7 Performance

Benchmarks are performed in a distributed memory setting to check parallel scalability of the current OpenFPM sparse grid implementation on a cluster with multiple GPUs. All benchmarks are performed on the *furiosa* computer cluster of the MPI-CBG, which has 20 nodes, each containing two Nvidia GTX 1080 GPUs and two CPU sockets with Intel Xeon(R) E5-2698 v4 at 2.20 GHz. Nodes are connected by a Mellanox Infiniband 40 Gb/s interconnect. Unfortunately, the GTX series of Nvidia GPUs do not support GPU-direct RDMA for MPI. OpenMPI 4.0.4 has been used with CUDA support and measured the bandwidth between a GPU and a CPU in the same node and across nodes to both be around 5.5 GB/s (measured with `osu_bibw` from MVAPICH OSU Micro-Benchmarks).

The first benchmark measures the performance of the sparse grid with a Gray-Scott simulation with different numbers of GPUs on a dense Cartesian grid, with periodic boundary conditions in X, Y, Z. Each simulation performs 5000 time steps with ghost layer communication at each time-step as explained in the previous section. The simulation result is visualized in figure 4.6A. The results on 1 to 16 GPUs (strong scaling) are given in table 4.1. In all benchmarks, both GPUs in one node are used. The benchmarks run using both single-precision `float` and double-precision `double` floating-point numbers with different overall grid sizes. When using single-precision arithmetics on 16 GPUs for a 512^3 grid, the network communication result in 72% of the total runtime. This is confirmed by profiling the time spent in the different communication

number of GPUs:	1	2	4	8	16
float , 512^3 dense	99.14	60.2	41.4	27.16	33.44
double , 384^3 dense	78.92	50.02	35.7	25.5	31.85
float , 1024^3 dense	-	-	-	147.6	90.42

Table 4.1: Runtime in seconds for 5000 time steps of the sparse block grid Gray-Scott simulation on dense Cartesian grids of different sizes and floating-point precisions as indicated, run on different numbers of GPUs (strong scaling). The largest case is only possible on 8 and 16 GPUs because of memory limitation. For the same reason, double-precision is only possible on 384^3 or smaller grids.

number of GPUs:	1	2	4	8	16
Packing	0.01	1.93	1.38	0.90	0.73
Send-Receive	0.03	8.05	13.3	13.1	24.8
Unpacking	0.007	2.31	1.68	1.26	1.16

Table 4.2: Time in seconds spent in the different communication operations for performing 5000 time steps of the Gray-Scott simulation on a dense 512^3 Cartesian grid of **float** values.

operations of our sparse block grid implementation. The results in Table 4.2 confirm the increasing communication overhead for the present strong scaling. We also measure the bandwidth on the interconnect achieved by the simulation in the communication phase. Using 2 GPUs in the same node, we measure 7.2 GB/s. This reduces to 4.4 GB/s when using 4 GPUs in two nodes, 3.38 GB/s on 8 GPUs, and 1.2 GB/s on 16 GPUs. This significant bandwidth degradation is due to the number of MPI messages increasing while the size of each message decreases.

In the second benchmark, OpenFPM runs the Gray-Scott system in a complex-shaped domain represented by a sparse grid with an average chunk density of 0.854. A visualization is shown in Fig 4.6B. On the boundary of the complex-shaped domain, we impose no-flux Neumann boundary conditions using the method of images. The overall size of the sparse grid is 968^3 . Communication is significantly reduced because there are no periodic boundaries, and the domain decomposition cuts perpendicular to the narrow cylindric channels connecting the spheres. On 8 GPUs, the communication overhead is now 30%, whereas it was 48% in the dense case. This results in improved scalability, as shown in Table 4.3.

Weak scaling has been also tested on more modern architecture like Nvidia Ampere A100 with 8 GPU interconnected with NVLink in the same node, and 400 Gbps across nodes. The problem is again a 3D Gray-Scott simulation in complex geometry like in figure 4.6. Calculations are done in single precision, and the grid goes from 1024^3 on one GPU to 4096^3 on 64 GPUs. The result are

number of GPUs:	1	2	4	8
double , 968^3 , sparse	200.2	105.303	59.2075	36.8114

Table 4.3: Runtime in seconds for 5000 time steps of the Gray-Scott simulation in a complex-shaped domain represented on a 968^3 sparse block grid with double-precision arithmetics on different numbers of GPUs (strong scaling).

number of GPUs:	1	2	4	8
float , 512^3 , sparse	2.8	4.0	3.9	3.8

Table 4.4: Runtime in seconds for all 100 simulation steps of the expanding spherical shell simulation on a 512^3 sparse block grid on different numbers of GPUs (strong scaling).

given in table 4.5. Until we remain on the same node efficiency remains at 95 percent, with NVLink (measured bandwidth of 986 GB/s). As soon as we move to multiple node the communication starts to bound the simulation, dropping the efficiency to around 30 percent.

The third benchmark uses a dynamic, time-varying geometry. The simulation considers a spherical shell in a cubic domain of edge length 2.5. As time progresses, the shell expands from initially an internal radius of 0.2 and an external radius of 0.4 to a final internal radius of 0.82 and an external radius of 1.02. In each of the 100 simulation time steps, i.e., after each expansion of the shell, the sparse grid is re-adapted followed by a **flush** operation. The measured runtimes for the complete simulation are given in table 4.4 for different numbers of GPUs. The benchmark (unnecessarily) performs two ghost layer communications in each simulation step in order to show the performance difference between the first one that computes all maps as explained in section 4.1.1 and the second one that reuses them (see Table 4.6). Since there is no computation on the grid, performance is limited entirely by communication. While the **flush** operation scales well, the first **ghost_get** is the bottleneck, as expected. The second **ghost_get** has a much smaller runtime than the first one

number of GPUs:	Time(efficiency)
1	75.20(1.00)
8	76.60(0.95)
16	196.0(0.37)
32	238.6(0.30)
64	240.0(0.30)

Table 4.5: Runtime in seconds for 5000 time steps of the Gray-Scott simulation in a complex-shaped domain represented on sparse block grids of different sizes on different numbers and types of GPUs(Weak scaling). Parallel efficiencies are in parentheses

because it reuses the maps (see Section 4.1.1). As expected, the communication overhead increases when distributing the constant grid size over an increasing number of GPUs.

number of GPUs:	1	2	4	8
<code>ghost_get 1</code>	0.0	11.7	13.8	13.4
<code>ghost_get 2</code>	0.0	1.2	3.0	3.4
<code>flush</code>	14	7.5	4.2	2.7

Table 4.6: Time in milliseconds to complete each sparse block grid function for a dynamic, time-varying grid on different numbers of GPUs. Times are given for the last step, when the spherical shell is largest, as this has the maximum communication overhead.

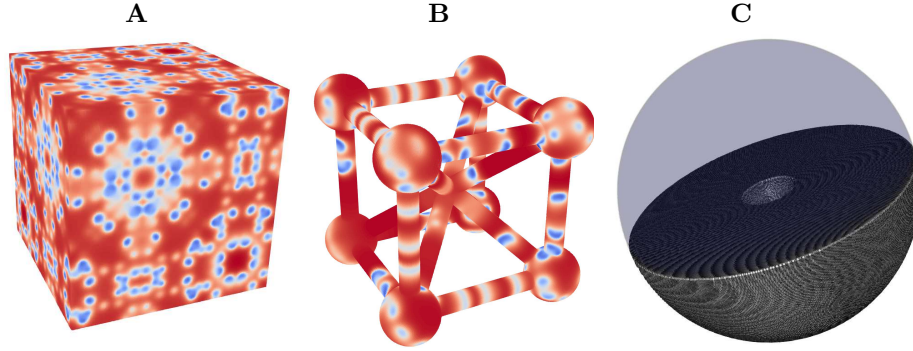


Figure 4.6: Visualizations of the simulations used in the three benchmarks. **A:** Gray-Scott reaction-diffusion simulation at time $t = 3000$ on a dense Cartesian grid computed with second-order central finite differences in space and explicit Euler time-stepping. **B:** The same simulation in complex-shaped domain at time $t = 10000$. The simulation domain consists of eight spheres connected by thinner tubes. **C:** Growing spherical shell with final inner and outer radius shown. For better visualization, the sphere is culled by its mid-plane.

4.1.8 Sparse-grid and level-set problems

Distributed sparse grid has an interesting application in the case of level-set problems. Level-set is a convenient way to represent surfaces without explicitly tracking the interface. For example, Bergdorf et al. [63] use level-set to simulate reaction-diffusion equations on moving surfaces. In the case of 2D surfaces embedded in a 3D space, the level set is a function $\phi(x, y, z)$ defined in the

3D space such that if we indicate with Γ the embedded closed surface, $\phi(x, y, z)$ satisfies 4.1. Because we are interested in solving the equation on the surface, we do not need to define $\phi(x, y, z)$ on the entire domain but only in the narrow band around $\phi(x, y, z) = 0$. It is convenient, for this reason, to construct a sparse grid instead of a grid and explicitly insert the point in the narrow band. From the discretization of the Laplacian on surfaces, we obtain the Gray-Scott reaction-diffusion equations 4.2. The only change is the replacement of the Laplace symbol ∇^2 with the equivalent operator on surface $\nabla \cdot [(I - \vec{n} \otimes \vec{n}) \vec{\nabla} u]$. The operator $(I - \vec{n} \otimes \vec{n})$ project the gradient into the plane normal to \vec{n}

$$\begin{cases} \phi(x, y, z) > 0 \text{ outside the volume enclosed by } \Gamma \\ \phi(x, y, z) = 0 \text{ on } \Gamma \\ \phi(x, y, z) < 0 \text{ in the volume enclosed by } \Gamma \end{cases} \quad (4.1)$$

$$\begin{aligned} \frac{\partial u}{\partial t} &= D_u \nabla \cdot [(I - \vec{n} \otimes \vec{n}) \vec{\nabla} u] - uv^2 + F(1 - u) \\ \frac{\partial v}{\partial t} &= D_v \nabla \cdot [(I - \vec{n} \otimes \vec{n}) \vec{\nabla} v] + uv^2 - (F + k)v, \end{aligned} \quad (4.2)$$

In this simplified problem, the shape is known analytically, and it is possible to calculate both the normal and the distance from the surface of each point. This characteristic allows us to construct the narrow band with the normal to the surface and the sign distance function. The surface is at radius 0.3 the band is wide $5\delta x$ where δx is the grid spacing. The initialization is done by creating a grid iterator on the grid 512^3 and inserting only the points falling in the narrow band. A small perturbation in U and V concentration is created at $0.6 < \theta < 0.8$ and $0.0 < \phi < 0.2$ where θ is the angle between the z axis and the vector connecting the center of the sphere and the point in the narrow band. Figure 4.7 show the problem initialization.

Once initialized, the time stepping follows the algorithm 16. The Level-set extension is based on [64] and is based on solving the equation 4.3

$$\begin{aligned} \frac{\partial u}{\partial t} &= \text{sign}(\phi) \frac{\vec{\nabla} \phi}{\|\vec{\nabla} \phi\|} \cdot \vec{\nabla} u \\ \text{sign}(\phi) &= \frac{\phi}{\sqrt{\phi^2 + \epsilon^2}} \end{aligned} \quad (4.3)$$

The extension is not performed every time-step but every five time-steps, because despite extending the solution in the normal direction ($\frac{\partial c}{\partial \vec{n}} = 0$), it also slightly affects the tangential direction. It is possible to see that one of the solutions of equation 4.3 is the configuration $u = \text{const.}$ Numerically, the extension iteration has the effect of slowly homogenizing the configuration. The simulation ran for 100,000 time steps with $dt = 0.1$ using euler integration. Figure 4.7 shows the simulation results.

Algorithm 16 Time-stepping level-set

```

1: for all Time-steps do
2:   for all Internal points in the band do
3:     Calculate the right hand side of the equation and update with the
       new value
4:   if Time-step is divisible by 5 then
5:     for all All points do
6:       Extend the solution normally
7:   Flip the old values to the new values and vice-versa

```

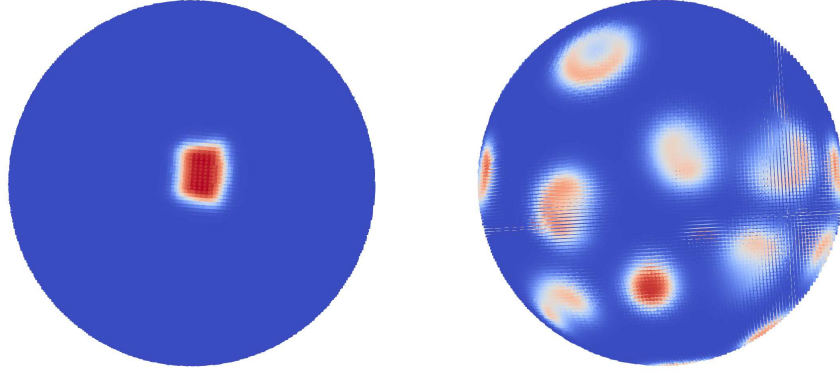


Figure 4.7: On the left initialization evolved after 100 time-steps, on the right gray scott simulation with level-set after 66000 time steps with $dt = 0.1$

4.2 Distributed particles

The thesis has shown until here the distributed grid `grid_dist_id` data structure as a generic code to distribute shared memory data structures. It also showed how to parallelize different variations like sparse-grids on CPUs and GPUs. In general `grid_dist_id` works as a distributed container of shared memory containers, if the single node container expose a particular interface of `get` functions, iterators and serializations as explained in section 4.1. This section will explain how to apply the same concepts to a particle-based distributed container. Such container has a very similar definition to a distributed grid-based container shown in section 4.5. However, these containers work differently from grid-based. The main external difference between particles and grids is that each point now has a position that can be read and written with `getPos`, while on grids, the points are fixed, and the position cannot be modified.

Listing 4.5: Distributed particles

```

1
2 template<unsigned int dim,

```

```

3      typename St,
4      typename prop,
5      typename Decomposition = CartDecomposition<dim,St>,
6      typename Memory = HeapMemory,
7      template<typename> class layout_base = memory_traits_lin>
8      typename vector_dist_pos =
9          openfpm::vector<Point<dim, St>,Memory,layout_base>,
10     typename vector_dist_prop =
11         openfpm::vector<prop,Memory,layout_base>
12     >
13 class vector_dist
14 {
15     vector_dist_pos v_pos;
16     vector_dist_prop v_prop;
17
18     ...
19 };

```

In listing 4.5, there is the definition of a distributed particle set. This structure expose the following parameters:

- **dim**: is the space dimensionality in which the particles lives.
- **St**: precision used to store position information
- **prop** is the aggregate or tuple of properties the particle contain
- **Decomposition**: is the space decomposition module used, as explained in section 3.1
- **Memory**: is the type of memory used, which can either be HeapMemory or CudaMemory, see section 2.1
- **device_vector_pos**: is the structure used at single node level to store the particle position information explained in section 2.2.9.
- **device_vector_prop**: is the type of vector used to store the properties of the particles, analogous to device_vector_pos

Distributed particle-based containers construct one shared memory data structure for each process, unlike grid-based containers that create one for each decomposition patch. The communication function **ghost_get** read particles in ghost areas from the neighborhood processors while **ghost_put** remotely send information written in the ghost area to the original particles. In the case of a particle set, the map function assumes a more important role. On a grid, **map** is generally used when the decomposition change to re-synchronize the information across processors. On particles, the function **map** is also used to redistribute the information when the particles move, their position is initialized, or the decomposition change. In the following, is going to be explained the main differences in implementation between the **ghost_get**, **ghost_put** on particles compared to distributed grids.

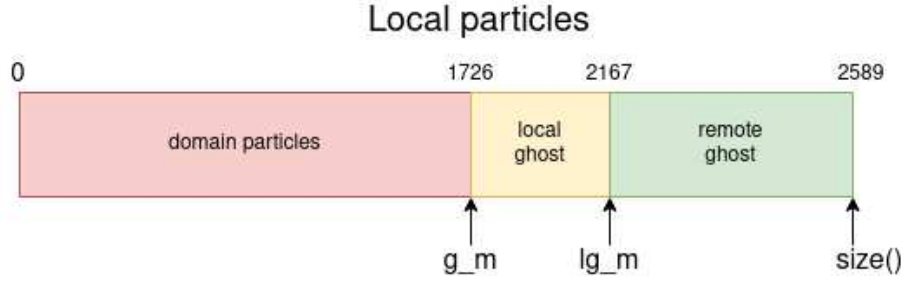


Figure 4.8: Layout of a local particles vector `v_pos` or `v_prop`. First come the local domain particles, marked by `g_m= 1726` than come local ghost particles from `g_m` to `lg_m= 2167`. Finally come the non-local ghost particles from `lg_m` to `size()`

4.2.1 `ghost_get`, `ghost_put`, `map`

Like was done for `grid_dist_id`, this section discuss the skeleton of the function `ghost_get` shown in algorithm 17. We first observe ghost particles are always added at the end of the internal vector, as shown in figure 4.8. The marker `g_m` indicates the transition between domain particles and ghost particles. The ghost particles are subsequently split into local and remote ghost particles, indicated by the marker `lg_m`. An additional buffer called `o_part` contains the indices of particles falling in the internal ghost boxes, equivalent to store the ghost particles sent by this processor to the neighborhood processors. This buffer becomes important when a `ghost_put` is performed. From the skeleton in algorithm 17, the shared memory particle container requires mainly three functionalities. From (steps 1-5-6), the vector container must have the possibility to resize. This function is already implemented for the `openfpm::vector` as explained in section 2.2.9. The second functionality is serializing information (step 3) and deserializing information from the received buffers (step 6). Like for grids, hardware-dependent operations like resizing, serialization and merging, are operated internally in the shared memory data structures like explained in section 3.2.3. The operation of labeling is instead a geometrical operation meaningful only in a particle set. This operation is implemented in the particle set and has two specializations, one for CPU and one for GPU. After labeling, the sending buffers are constructed, and sent using type aware communication as explained in section 3.2.3. Type aware communications are hardware-independent and enable the possibility to send across the network complex properties like dynamic nested structures in a particle property.

`ghost_put` works like the inverse operation of `ghost_get`. The information on the ghost is merged back to the original particles. Because a particle can be replicated in the ghost area of multiple processors, merging information back result in conflicts. In order to solve conflicts, an operation is selected to merge data. Typical operations are summation minimum and maximum for simple properties like we have seen for the flush operation in the sparse grid on

Algorithm 17 `ghost_get`

- 1: Delete all particles outside the processor domain. This is done by resizing the position vector **v_pos** and the property vector **v_prop** down to **g_m**
- 2: Run across all the particles and label them if they fall in one internal ghost box. A cell list is used to speed up this process. Each particle can have multiple destination processes. During the labeling process, each process saves the indices of the particles sent. This information is saved in a vector of vectors (**o_part**). In each vector, the indices are unique.
- 3: Create the communication buffers and serialize position and selected properties information.
- 4: Send the buffers to their corresponding destination process.
- 5: Replicate local particles in the ghost area like in figure 4.8 if some direction has periodic boundary conditions.
- 6: Append/merge the data received at the end of the position and the property vector. Every appended particle is tracked by each processor using markers like in figure 4.7. While the size of each buffer received is saved in **recv_sz_get_prp**

GPU in section 2.2.14. The skeleton algorithm for the **ghost_put** is given in algorithm 18

Algorithm 18 `ghost_put`

- 1: Read **recv_sz_get_prp** from the previous **ghost_get** and construct a sending buffer for each process, reading from the ghost areas
- 2: Send the buffers to their corresponding destination
- 3: Merge the data received using **o_part** with the existing data using the selected operation (sum, max, min).

The hardware specialization on CPU and GPU for the algorithm 18 is hidden internally in the `openfpm::vector`, while the merging operation is defined with a functor structure shown in listing 4.6.

Listing 4.6: ghost put functor for sum reduction”

```

1  template<typename Tdst, typename Tsrc>
2  struct add_
3  {
4      __device__ __host__ static inline void operation(Tdst & dst,
5                                                       const Tsrc & src)
6      {
7          dst += src;
8      }
9  };

```

The functor structure for the **ghost_put** function is a meta-function with two arguments. The two arguments define the type of the operands. The method called **operation** defines how to merge two elements. For example, in the listing 4.6 the functor sum all the data together, with the line `dst += src`.

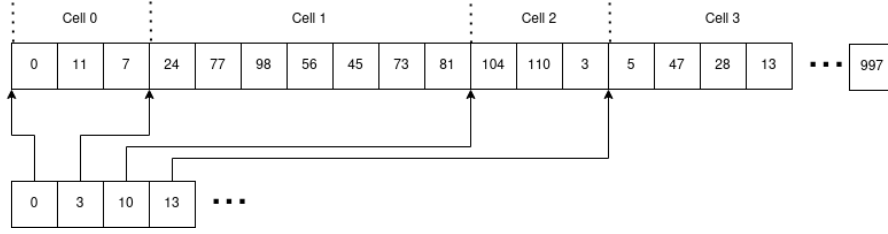


Figure 4.9: Particle in cell buffer contains the indices of the particles in each cell of the cell-list. It also has a second buffer containing the exclusive prefix sum of the number of particles in each cell.

In the case of many-core architectures or GPUs, we try to avoid **ghost_put** operations because of race conditions. It is possible to use **ghost_put** if hardware provides some type atomic operation, like is done for the GPU functor 4.7. Because **ghost_put** use ghost areas and need the buffer o_part to works, it can be performed only after the first **ghost_get**

Listing 4.7: GPU ghost put functor

```

1 template<typename Tdst, typename Tsrc>
2 struct add_atomic_
3 {
4     __device__ __host__ static inline void operation(Tdst & dst,
5                                                     const Tsrc & src)
6     {
7         atomicAdd(&dst,src);
8     }
9 };

```

The **map** procedure, or the procedure to redistribute particles once they move or the decomposition change, works in a very similar way to the **ghost_get**. The main difference is that the labeling process is performed using the sub-domains rather than internal ghost boxes.

4.2.2 Cell-List CPUs and GPUs

Cell lists constitute the basis for finding neighborhood particles. Their construction in OpenFPM follows the standard general algorithms used in computing them on CPUs and GPU. OpenFPM provides several variations for the GPUs and the case of symmetric interactions. In this section, we are going to see these variations and their internal design. This section shows how it is possible to implement such variations in a modular and agnostic way for the user. The cell-list construction on GPU or many-core architecture is very different from the single-core case. The skeleton algorithm on CPU is in algorithm 21 and for GPUs is in algorithm 19.

In the case of GPUs, the first step is to count the particles in each cell. An essential feature of **atomicAdd** is to increment the counter of a cell and at the same time return the old value of the counter before incrementing it. The

index returned by `atomicAdd` can be considered a unique in-cell index (`a_id`) for the particle in the cell. If we now want to create a buffer that contains the particle indices for each cell (particle-in-cell buffer), the most common way is to calculate the exclusive prefix sum of the array containing the number of particles in each cell. The calculated prefix sum indicates the starting point of each cell in the particle-in-cell buffer as shown in figure 4.9. The kernel to fill the particle-in-cell buffer with particles indices runs on each particle and reads the id of the cell in which it resides and its in-cell index (`a_id`). From the prefix sum, we retrieve the starting point of the cell (`s_id`), to finally calculate the position of the particle index in the particle-in-cell given by `a_id + s_id`. The details are given in algorithm 19

The cell list construction just described is not deterministic. If we look at the in-cell indices, we realize that while they are unique for each particle in the cell, the **`atomicAdd`** does not have deterministic behavior. Several runs can produce a different ordering of the particles within the cell. While this is not a problem, when we want to debug our program, this could make the task more complicated. For this reason, the option `MAKE_CELLLIST_DETERMINISTIC` slightly changes the cell-list construction to be deterministic at the cost of performance. Here, the construction of the particle-in-cell buffer using the in-cell index from **`atomicAdd`** is substituted by doing a stable merge-sort on GPU of the cell-index of the particles. The buffer constructed can be considered a particle-in-cell buffer. While, in general, the merge-sort version result to be slower than the variation with **`atomicAdd`**, it makes all the calculations deterministic across different runs. All the constructions described until here are referred as dense cell lists.

In addition to dense cell lists, the GPU also has sparse cell list variants to reduce memory overstress. In the standard cell-list, every cell has a non-zero cost in memory. For each cell of a standard cell-list, there is a counter counting the number of particles in the cell. While this does not produce problems in most cases, consider now the case where there is a significant portion of space without any particles. In this case, one of the GPUs takes most of the space with no computational cost. Despite having no computational cost, they have a non-zero memory cost. Most typical load balancers balance based on computational cost, and this raises a potential situation where the vast majority of space is given to only one GPU. If this is the case, all the empty cells, despite not having a cost in computation, can potentially overflow the memory of the GPU. Sparse cell-list is a variant that avoids constructing the buffer containing the scan of the number of particles for each cell. In particular, the offset buffer is constructed only for the filled cells, leaving out the empty ones. In order to do this, a hash map or sorted map is required. In section 2.2.14 a GPU version of a sorted array map was shown, where it was possible to search and add elements in parallel, keeping the memory footprint of order $O(N)$, where N is the number of elements/cells added. Doing a search unfortunately has complexity $O(\log(N))$. In order to avoid this search, the cell-list constructs an array (`nn_cell.array`) containing for each non-empty cell the index-position of the neighborhood non-empty cells. While the construction still requires $O(N\log(N))$ in complexity

after construction, the complexity to search the neighborhood of a cell is reduced to $O(1)$. The construction of the `nn_cell_array` buffer is advantageous if we have to explore the neighborhood of the particles multiple times before reconstructing the cell list on GPU.

GPU cell list implementation presented until here fit also many-core implementation, on a single-core CPU, the implementation follows the algorithm in 21. The base structure for the particle-in-cell buffer is now an array of arrays or, more practically, a vector of vectors. Each cell contains a vector with the indices of the particles inside that cell. As for the GPU case, it is possible to create multiple implementation variances, it has been done also for the CPU. In the CPU case the variances are based on multiple implementations of the array of arrays concept.

The array of arrays interface is common to each implementation and hides internally the implementation details. Every implementation can have different complexity in memory and computation for the API. The common interface requires a **get** function with two indices (*i,j*). One to retrieve the cell and the other to retrieve the element inside the cell. We remark that it is not a simple 2D array because every cell can have a variable number of elements. To complete the API, we need in addition a function **size()** to return the number of cells and a function **getNelements(i)** to return the number of elements in one cell, and a function to add an element to a particular cell. The structure also provides convenient operators like swap, assignment, duplication, and move semantics. These operations in the array of arrays API are required in order to support them at the cell-lists structure level.

At the moment of writing, three implementations are given of this API. One uses a standard "array of arrays" as an internal structure. In this case, mapping **get(i,j)** is trivial. The first index *i* is used to access the first vector the second index is used to access the nested vector. The second implementation is a single array implementation. The "array of arrays" is replaced by a single array where each cell can store a maximum number of particles N_{max} . In case one of the cells overshoots this threshold, the array is resized to store $N_{max} \times c$ particles for each cell where *c* is some growing factor bigger than one. The third implementation instead substitutes the first array with an unordered map. The effect is very similar to the sparse cell-list to reduce the memory complexity when many empty cells can flood the memory of one process. Table 4.7 specifies the memory and computational complexity for the function `get(i,j)` where *M* is the number of cells, *N* is the number of particles, and *k* is the number of buckets in the unordered map.

Despite the "single array" and "array of array" having the best worst and average complexity equal to $O(1)$ for the function `get(i,j)`, the constant is different. In the case of "array of arrays" we have to follow a pointer, so we need to access the memory two times instead of one.

Impl.	Memory	: Computation	Best	Worst	Average
Array of Array	$O(M)+O(N)$	$O(1)$		$O(1)$	$O(1)$
single array	$O(M*N_{max})$	$O(1)$		$O(1)$	$O(1)$
map of array	$O(N)$	$O(1)$		$O(N)$	$O(N/k)$

Table 4.7: Complexity on memory for the particle-in-cell-buffer, and computation complexity for the function $get(i,j)$ for each implementation. M is the number of cells, N is the number of particles and k is the number of buckets in the unordered map

Algorithm 19 Cell-List GPUs

Input: Processor domain box, divisions on each directions

- 1: **kernel:**threads = M , workblocks = $\lceil N/M \rceil$, N = number of particles
 - 2: Read the position of the particles
 - 3: calculate in which cell-index the particle reside (cell-index is linearized)
 - 4: increment the cell counter using atomicAdd
 - 5: save the number returned (a_id) into a buffer pic_index for each particle.
 - 6: Scan (exclusive prefix sum) the array with the number of particles on each cell (scan_cell)
 - 7: create a buffer as big as the number of particles and call it particle-in-cell-buffer
 - 8: **kernel:**threads = M , workblocks = $\lceil N/M \rceil$, N = number of particles
 - 9: Read the pic_index of the particle i (pi)
 - 10: calculate in which cell-index the particle reside cid (linearized)
 - 11: fill the particle_in_cell_buffer[scan_cell[cid]+pi] = i
-

Algorithm 20 Sparse cell-list GPUs

Input: Processor domain box, divisions on each directions

- 1: **kernel:**threads = M, workblocks = $\text{ceil}(N/M)$, N = number of particles
 - 2: Read the position of the particles
 - 3: calculate in which cell-index the particle reside (cell-index is linearized)
 - 4: Save the index into a buffer cell_index_buffer for each particle
 - 5: Create a sparse vector on GPU 2.2.14 and use the cell_index_buffer as set of keys to add, the values are set to one.
 - 6: flush the map with add operation. This create a unique list of sorted cells indices with the number of particles in each cell.
 - 7: **kernel:**threads = M, workblocks = $\text{ceil}(N/M)$, N = number of cells filled
 - 8: **while** neighborhood cells: cid **do**
 - 9: use binary search to find the position of the neighborhood cell p.nn.
 - 10: **if** neighborhood is found **then**
 - 11: increment nn_buffer[cid] with atomicAdd
 - 12: Scan the buffer that count the neighborhood cells for each cell (nn_buffer) and save this buffer into nn_buffer_scan. The last element of the scan contain the size of the buffer needed to create the buffer of neighborhood cells index.
 - 13: reset nn_buffer to zero
 - 14: Create a buffer big enough to store the neighborhood cells index
 - 15: **kernel:**threads = M, workblocks = $\text{ceil}(N/M)$, N = number of cells filled
 - 16: **while** neighborhood cells: cid **do**
 - 17: use binary search to find the position of the neighborhood cell p.nn.
 - 18: **if** neighborhood is found **then**
 - 19: increment nn_buffer[cid] with atomicAdd, ca_id is the integer returned.
 - 20: Fill nn_cells_buffer[nn_buffer_scan[cid] + ca_id] = p.nn
-

Algorithm 21 Cell-List CPUs

Input: Processor domain box, divisions on each directions

- 1: create an array of arrays (abstract concept as explained for CPU) as big as the number of cells covering the domain of the processor
 - 2: **for all** Particles **do**
 - 3: Calculate the cell index of the particle
 - 4: Add to the the particle index in the particle.in.cell.buffer add(cid,p.id)
-

Chapter 5

Numerics

5.0.1 Expression Parsing to solve equations

Until here, we have shown every layer that leads to the distributed data structures. The following layers of OpenFPM introduce concepts used in numerical simulations. Such concepts aim to ease the coding of numerical simulation in continuous particle methods.

Listing 5.1: Expression Based System

```

1  auto P=getV<Pressure>(particles);
2  auto Pd=getV<Pressure>(particles);
3
4  Derivative_x D_x;
5  Derivarive_y D_y;
6
7  auto equation = D_x(P) + D_y(P) + 5.0;
8  Pd = equation;

```

We start by introducing a system to solve continuous models described by partial derivative equations. In particular, we separate the mathematical representation of the equations from its discretization.

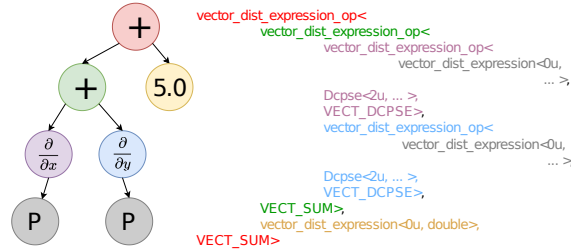


Figure 5.1: The equation $\frac{\partial P}{\partial x} + \frac{\partial P}{\partial y} + 5.0$

expressing equations in C++

OpenFPM is written in C++, and it uses template expression parsing techniques to implement an interface to write generic differential equations. In order to understand how template expression parsing can be used to express equations, we have first to understand how a partial differential equation can be represented with a tree. In this tree, the leaves are fields or constant numbers, in the following will indicate them as terminal nodes. Any non-leaf node is a unary or binary operator and, in the following, will be indicated as a non-terminal node. An example is given for the equation $\frac{\partial P}{\partial x} + \frac{\partial P}{\partial x} + 5.0$ in figure 5.1. To construct a tree from an equation in C++, we use template expression parsing re-adapted for our case.

From figure 5.1 we see how every tree expression has an equivalent representation as a type. In particular, we use the possibility to nest template parameter types to represent an entire tree as a type. Every node is a templated class representing the basic binary operators like $+$ $-$ $*$ $/$ or unary operators like an n-order derivative. An example of how they are defined is given in the listing 5.2.

Listing 5.2: Expression Based System

```

1 // Example of definition for operator+ non terminal node
2 template <typename exp1, typename exp2>
3 class vector_dist_expression_op<exp1,exp2,VECT_SUM>
4 {
5     ...
6 }
7
8 // Example of definition of field terminal node
9 template<unsigned int prp, typename vector>
10 class vector_dist_expression
11 {
12     ...
13 }
```

The non-terminal nodes have three templated parameters. In the case of binary operators like $+$ $-$ $*$ $/$, the first template parameter indicates the expression on the operator's left-hand side. The second parameter indicates the right-hand side of the operator. The third argument indicates the type of operation. Any operation is a non-terminal node that can operate on a terminal node like fields and number nodes or another non-terminal node. We restrict non-terminal operators in the tree to binary operators like $+$ $-$ $*$ $/$ and unary operators like partial derivatives. In the case of derivative, the second argument is the class encapsulating the method used to discretize the derivatives like DCPSE. Terminal nodes instead represent a field linked to a particular property in the particle set or a number. Given such nodes an expression like $\frac{\partial f(x,y)}{\partial x} + \frac{\partial f(x,y)}{\partial x} + 5.0$ has a templated type representation given in figure 5.1.

In order to construct such C++ expression type, we have to create productions rules using operator overload. For example to express the production of the node $+$ in the expression $\frac{\partial f(x,y)}{\partial x} + \frac{\partial f(x,y)}{\partial y}$ we overload the operator $+$. The binary function in listing 5.3 has two parameters accepting two non terminal nodes. The first parameter is a non-terminal node indicating the left-hand side

of the "+" expression, the second parameter indicate the right hand side of "+" the expression. In figure 5.1 the left parameter is equivalent to the violet type/node (and nested), and the second parameter indicates the right-hand side of the expression and is equivalent to the blue type/node and nested. Inside the function, we create the node plus in figure 5.1 equivalent to the green node and return this new node representing the summation of the two derivatives.

In the case of derivatives, it possible to introduce a class **Derivative**. The production rule, in this case, is expressed as overload of the *operator()* of the class, in which it is possible to produce production rules from the expression $D(f)$ or $D(f + g)$. Like for the operator +, the derivative the code in the *operator()* produces the nodes in violet and blue in figure 5.1. In the case of unary operators, the second parameter contains the algorithm used to discretize the derivative. In the example, is shown the class **DCPSE** that contains the DCPSE discretization algorithm.

Listing 5.3: Production rule for the operator+

```

1
2 template<typename exp1 ,
3         typename exp2,
4         unsigned int op1,
5         typename exp3 ,
6         typename exp4,
7         unsigned int op2>
8 inline vector_dist_expression_op<
9     vector_dist_expression_op<exp1,exp2,op1>,
10    vector_dist_expression_op<exp3,exp4,op2>,
11    VECT_SUM>
12 operator+(const vector_dist_expression_op<exp1,exp2,op1> & va,
13           const vector_dist_expression_op<exp3,exp4,op2> & vb)
14 {
15     vector_dist_expression_op<
16         vector_dist_expression_op<exp1,exp2,op1>,
17         vector_dist_expression_op<exp3,exp4,op2>,
18         VECT_SUM> exp_sum(va,vb);
19
20     return exp_sum;
21 }
```

Once defined the nodes for the expression tree and the production rules, an expression like in listing 5.1 produces a templated type that contains the equation as a tree. Computation can now be constructed creating a function **value** in each tree node, and having each node calling the **value** function of its children. Because the type is known at compile-time, the call sequence defining the tree traversal is also known at compile. In this case, function inlining and context optimization will generate the code of the expression. Thanks to expression parsing, it is possible to compose computation constructing expressions.

While the **value** function accomplish the task to compute expressions, it is possible to use expression to accomplish other tasks like generating a matrix representing the Laplacian operator or a matrix representing a system of equations. The second task is performed by the function **value_nz**. In this case, the nested call of **value_nz** generates instead the code needed to construct the non-zero columns of a matrix representing the equation for one particle.

For the case of **value**, The instruction triggering the computation is the line `Pd=equation` in listing 5.1. This line relies on the fact that terminal nodes have defined an operator `equal`. Such operator contain the code in listing 5.4. As we can see, it contain a loop for every particle **key_orig** in which the expression **v_exp**, containing the entire expression graph, is evaluated and stored in the particle set **v** at position **key**. When **v** is a standard particle set the transformation **v.getOriginKey(key)** is the identity, and **key_orig** is equivalent to **key**. It starts to have a role when we use subsets. Consider the case where we have in **v** ten particles with indices from 0 to 9. We now construct a subset of particles with ids 1,3,7. A particle subset is a restriction of the particle set to a subset. The type contains all the methods that the particles have with the difference that the iterators function like **getDomainIterator()** return iterators running only on the subset particles. If now we pass instead of **v** the subset, then the loop will run for three iterations with **key_orig** assuming value 1,3,7 and **key** 0,1,2. In particular the particle id 0 in the subset map to 1, id 1 to 3, 2 to 7. The key, **key_orig** split design, allows us to introduce a map between a set of contiguous indices (0,1,2) with a set of scattered indices (1,3,7). In our case, the map is used to implements operation acting only on a subset of particles.

Listing 5.4: Expression computation

```

1
2 template<typename vector, typename expr>
3 static void compute_expr(vector & v,expr & v_exp)
4 {
5     v_exp.init();
6
7     auto it = v.getDomainIterator();
8
9     while (it.isNext())
10    {
11        auto key = it.get();
12        auto key_orig = v.getOriginKey(key);
13
14        pos_or_propL<vector,prp>::value(v,key) = v_exp.value(key_orig);
15
16        ++it;
17    }
18 }
```

if the type of **v_exp** is the type presented in figure 5.1 **v_exp.value()** will trigger the tree-traversal at compile time and the generation of the instruction to evaluate the equation in the point **key_orig**.

Tensorial expressions: scalars and vectors and use on GPUs

Expressions are transparent from the discretization used for derivatives, as seen in the previous section. They are also transparent from the type on which the operator acts like a scalar or a vector. This feature relies on the method **value** to partially redefine what does internally. Let consider the code in the **value** method of the node `+`. The code is reported in listing 5.5. The result type of the expression `o1.value(key)+o2.value(key)` indicating the summation of the

left expression with the right expression is left on purpose generic with `decltype(o1.value()+o2.value())`. This lead to different options depending of what the underlying left (`o1.value(0)`) and right (`o2.value(0)`) sub-tree generate as type. For example, in a scalar like double or float, the summation is the sum of two floating-point numbers, and the `decltype` of double + double is a double. In the case of a complex number, the meaning lives in the overload of the operator+ for a complex number. In the case of an `openfpm` point with three components `Point<3, double>` the plus drive a secondary nested template expression parsing. The summation of two points does not generate the code to sum the three components of the two points but generates a type representing the code to sum two points. The generation of the code calculating the expression is not done inside the function value of the operator. The generation is delayed to when the expression is assigned to the destination particle property. We can use this method to make generic the operator +, and we can use it to drive a secondary nested template expression parsing defined for the `Point<3, double>` data structure.

The specification `__device__` make possible to use these type of expressions to generate code on GPU. In the case of GPU, the loop in listing 5.4 is replaced by an equivalent kernel code for GPU that handles one particle for each thread. Each thread execute one iteration of the code inside the loop in listing 5.4. The compiler (nvcc for CUDA) generates the GPU to evaluate the expression.

Listing 5.5: Prodiction rule for the operator+

```

1
2 __device__ __host__
3 inline auto value(const unsigned int & key)
4 -> std::remove_reference::type > const
6 {
7     return o1.value(key) + o2.value(key);
8 }

```

Attributes on the expression graph

In order to discover the type of particle set this expression is acting on, we have to search at least one leaf or terminal node working on a field of a particle set, take the type and return it to the root node.

In order to do this for each node terminal and non-terminal, an attribute called `vtype` is introduced. The attribute is calculated or synthesized from its children. In the listing 5.6 we see the definition of this attribute for the operator plus. `has_vtype` is a meta function that check if the sub-tree contain a field terminal node. More explanations are given in figure 5.2. If the answer is true the meta-function `first_or_second<...>::vtype` map to `exp1::vtype`. If the answer is false, it maps to `exp2::vtype`. In figure 5.2 we see how the attribute computes in two different cases.

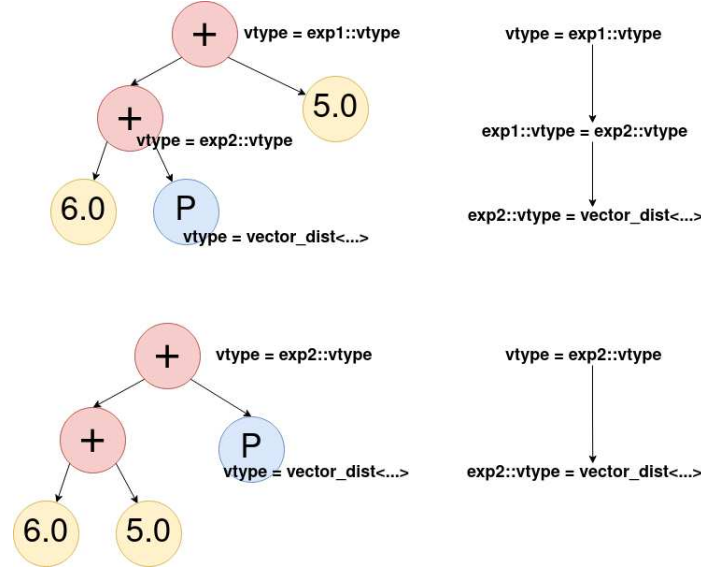


Figure 5.2: Attribute computation. In the first case **UP** the field (blue node P) is on the left part of the root node, in the second case **DOWN** the field (blue node P) is on the right part of the root. **UP** The root "+" node query the left children with `has_vtype<exp1>::value` to see if `vtype` is defined on `exp1`. `exp1` is again a non-root "+" node, this trigger again the same meta-code that check with `has_vtype<exp1>::value` if the left children (6.0) in yellow has `vtype` defined. The query will result in failure because the node 6.0 does not have `vtype` defined. This will make `has_vtype<exp1>::value` in the non-root "+" return false. Returning false will make the non-root + node to define `vtype` from the right children (P). P is a field and has a `vtype` defined. The `vtype` of the non-root "+" node will be assigned to type defined by P. This will also make to succeed `has_vtype<exp1>::value` of the root node that will assign the `vtype` of the root + node to `exp1::vtype` equivalent to `vtype` in P. **DOWN** The root + node again query the left children with `has_vtype<exp1>::value` to see if `vtype` is defined. `exp1` is a non-root + node, this trigger again the same meta-code that check with `has_vtype<exp1>::value` the left children (6.0) in yellow. The query will result in failure because the node 6.0 does not have `vtype` defined. This will make `has_vtype<exp1>::value` in the non-root "+" return false. Returning false will make the nested + node to define `vtype` from the right children 5.0. This will also fail and make `has_vtype<exp1>::value` in the root + to return false. This will make the root node + to define its `vtype` from the right node P. This will succeed and will define `vtype` in the root node to `exp2::vtype`

Listing 5.6: Attribute calculation for operator+

```

1
2 typedef typename first_or_second<
3           has_vtype<exp1>::value,
4           exp1,exp2>::vtype vtype;

```

5.0.2 Iterators

Distribution across processors or loops optimizations can make the code less readable or, even worse, make it error-prone. To see how iterators can simplify the code to write a loop, we consider a loop across the points of a grid. Looking at section 4.1 we know that the distributed grid creates one or more patches for each processor. Multiple patches require creating a double loop, one over the patches and one nested over the points of one patch. The loop over the patch is split in each dimension. Overall, if we want to loop over a distributed grid in three dimensions, we end with four nested loops. This complexity can be hidden with an iterator hiding patch and dimension loops. In the listing 5.7 we show the typical loop with iterators for the grid.

Listing 5.7: Iterator

```

1
2
3 auto dom = grid.getDomainIterator();
4
5 while (dom.isNext())
6 {
7     auto key = it.get();
8
9     // ... computation
10
11     ++dom;
12 }

```

The loop in listing 5.7 has the same form of a loop over particles and abstract dimensionality and patch iteration. The code in 5.7 is the most generic loop existing in OpenFPM and homogenizes particles and grid iterations.

Iterators can also be used to create cache friendliness iterations for certain structures. For example, using iterators like in listing 5.8 we can create an iteration over particles according to a space-filling curve. Using **getCellList_hilb** we obtain a Cell-list and get an iterator from the cell-list to generate an iterator that moves across the particles cell-by-cell following a Hilbert curve as a space-filling curve. If instead we use **getCellList** the space-filling curve is normal striding.

Listing 5.8: Iterator for hilbert

```

1
2 auto NN = part.getCellList_hilb(r_cut); // Hilbert
3 // auto NN = part.getCellList(r_cut); // Non-hilbert
4
5 auto dom = NN.getIterator();
6
7
8 while (dom.isNext())
9 {
10     auto key = dom.get();
11
12     // ... computation
13
14     ++dom;
15 }

```

Iterators also help to run across very complex iterations, like symmetric interactions with diagonal interactions for particle sets. These types of interactions are explained in PPM’s original paper [44]. Without details, these types of interaction require iterate across ghost particles depending on their geometrical position and the domain decomposition. Iterators hide such complex iteration enabling diagonal-based interaction implicitly, as we can see from the code in listing 5.9.

Listing 5.9: Iterator for symmetric diagonal interactions

```

1
2 auto dom = vd.getParticleIteratorCRS(NN);
3
4 while (dom.isNext())
5 {
6     auto key = dom.get();
7
8     // ... computation
9
10    ++dom;
11 }

```

Safetiness and differences from standard C++ iterators

OpenFPM uses index-based iterators, a conceptually different type of iterator compared to the standard. Index based iterators return keys or indices that can be used with the data structure. Standard C++ iterators instead act like pointers by design. This means the C++ standard iterators inherit all the problems relative to pointers. In particular, once we have an iterator/pointer, we do not know its validity. The primary data structure could have died, changed in range, or reallocated its internal memory, invalidating the iterators. Many operations can invalidate iterators in the same way pointers can be invalidated, or worst, arithmetic with iterators lead to the same dangers as pointers, leading to memory corruptions hard to find.

Range iterators force to access the main data structure for data read/write.

This gives the possibility to control every access for validity based on the memory's range or existence. In addition, iterators cannot be invalidated. They return only numbers and do not access memory, so they are valid by definition. If an index overflow the structure, the range will be checked once we use it on a data structure. Because all data-structure derives from `grid_base_impl` explained in section 2.14, adding these validity/range check to `grid_base_impl` will guarantee that all the top-level data structure explained until here are safe from memory corruption and overflow from accesses.

Range/validity checks at every access are costly. For this reason, such checks can be activated and deactivated at compile-time using the macro `SE_CLASS1`. Using a raw pointer, new and delete, and standard functions of C++ language is unsafe. Because of this, the development of OpenFPM above the layer of `grid_base_impl` explained in section 2.2.8, has been conducted avoiding pointers and with index based iterators. High-level languages have already shown how pointers are avoidable for coding, the result in section 6 will show how not only are avoidable, but it is possible to retain performance with high levels constructs.

5.0.3 File I/O, check-point restart

All the distributed data structures presented can be saved to file. All of them, have the method `save()`. This function can be used to save to file a data structure. Every data structure also has a method `load()` able, irrespective of how data were distributed across processors during save, to reload the data into the distributed data structure. OpenFPM files can be used to restart a simulation from a previously saved state (check-point restart).

Each processor internally serializes the local memory of the data structure using the serialization system explained in 3.2.3. This operation reduces any data structure containing nested containers like a grid of vectors or vector of vectors into a contiguous stream of bytes. The N streams with N equal to the number of processors must be written on a single file. This last operation is done using HDF5 [65] or ADIOS2 [66]. Meta-data are added automatically to allow de-serialization on another numbers of processors and other domain decompositions. During load, the saved streams are read in parallel by individual processors, and after distributed de-serialization, mapped into the new domain decomposition. This map-after-read strategy is preferred, as it causes data to be read/written in large contiguous blocks rather than in multiple smaller random reads, producing less load in the I/O subsystem.

In addition to HDF5 output, distributed data can also be saved to files in VTK format [67] using the OpenFPM method `write()`. This enables direct visualization of the particle and mesh data, e.g., in Paraview [68], an open-source scientific data visualization software that natively reads VTK files. While OpenFPM HDF5 files cannot directly be visualized in Paraview, they can easily be converted to VTK files using `load()` to reload the HDF5 files into distributed data-structures followed by `write()` to export to VTK. VTK output and Paraview were used to generate all visualizations in Section 6 of this paper.

OpenFPM also supports OpenPMD files creation on HDF5 and ADIOS2 format. With this format, we can save large data-set from parallel simulations into a format directly visualizable from data analysis and visualization applications without conversion.

Chapter 6

Results

This section demonstrate the usability, performance, and scalability of OpenFPM in various test scenarios and application domains. Each application is examined with the domain’s state-of-the-art code, demonstrating that a generic framework like OpenFPM can match, and in some cases even outperform, application-specific codes that have grown over many years. All tests are run on the TU Dresden Center for Information Services and High-Performance Computing’s computer cluster. The cluster’s nodes each include two 2.5GHz 12-core Intel Xeon E5-2680v3 CPUs (for a total of 24 cores per node) sharing 64GB of RAM. The memory bus’s max bandwidth is 60GB/s per socket. The memory bus has a peak bandwidth of 60GB/s per socket. The cluster connection is a 40-gigabit-per-second Infiniband network. The operating system on the machine is RedHat Enterprise Linux (RHEL) Server release 6.9 (Santiago). OpenFPM was compiled with GCC g++ 7.1.0 and linked against the OpenMPI 3.0.0 [12] implementation of the MPI standard version 3 [11] for all tests.

On the benchmark machine, each processor (socket) has its own independent memory bus. The cores within each processor, however, share the memory bandwidth. The results of a concurrent memory-read benchmark are shown in Table 6.1. The memory bandwidth reduces from 14.7 GB/s when using only 1 core to 3.65 GB/s when using all 12 cores of a processor in parallel. Running on 16 cores with rank-affinity binding will, e.g., result in the first 12 cores running at 3.65 GB/s and the remaining 4 cores at 9.7 GB/s. Unless otherwise noted, all benchmarks reported here use rank-affinity binding. This synthetic benchmark reveals that OpenFPM can saturate the machine’s memory bus when using 6 or more cores per socket.

#cores/socket:	1	2	4	8	12
bandwidth GB/s:	14.7	13.7	9.7	5.4	3.65

Table 6.1: Memory bandwidth per core on the benchmark machine when using different numbers of cores all on the same processor socket.

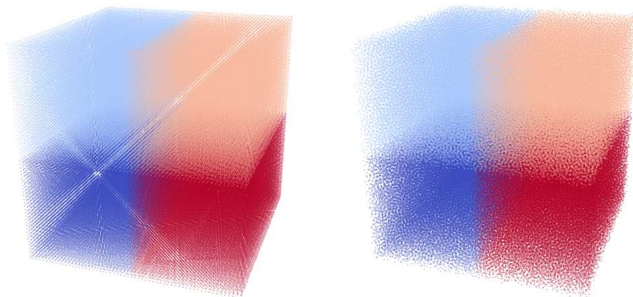


Figure 6.1: Particle configurations at the start of the simulation (left) and after 5000 time steps (right) for the Lennard-Jones molecular dynamics test case. The system was thermally equilibrated after 1000 time steps. In this example, a decomposition into four sub-domains is used, indicated by different colors. Each particle is plotted as a dot with the color of the respective sub-domain.

6.0.1 Molecular dynamics

First, let us look at a classic Molecular Dynamics (MD) simulation of atoms interacting with the Lennard-Jones potential. Particles represent atoms in this simulation, and they interact according to the pairwise potential:

$$V_{\text{LJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right] \quad (6.1)$$

as a function of the interaction distance r between the two particles. The parameters σ and ϵ define the potential's zero-crossing and well depth, respectively. To efficiently discover the interaction partners of the particles, we use OpenFPM's implementation of Verlet lists [48] to build the simulation. Due to the small time steps commonly employed in molecular dynamics simulations, implementing a Verlet-list becomes necessary. In doing so, we take advantage of the fact that the interactions are symmetric and therefore the forces between any given pair of interactions need be computed only once. This also involve the creation of ghost particles, which are copies of particles owned by nearby processors appended to the list of particles owned by a processor of interest, for the calculation of partial forces. These calculated partial forces are then merged using the OpenFPM's `ghost_put` (see Section 4.2.1).

We compare the OpenFPM-based solution to LAMMPS, a well-known and highly efficient parallel MD code [23]. 216,000 particles are placed on a regular Cartesian grid of size 60^3 and spacing one as shown in in figure 6.1. The domain is decomposed into four sub-domains which are demarcated by different colors in figure 6.1. A symplectic velocity Verlet time-stepping strategy is employed with step size $\delta t = 0.01$ to initially equilibrate the system before simulating its dynamics. We choose $\sigma = 0.1$ and $\epsilon = 1$ for the Lennard-Jones potential, and periodic boundary conditions in all three coordinate directions. The simulation

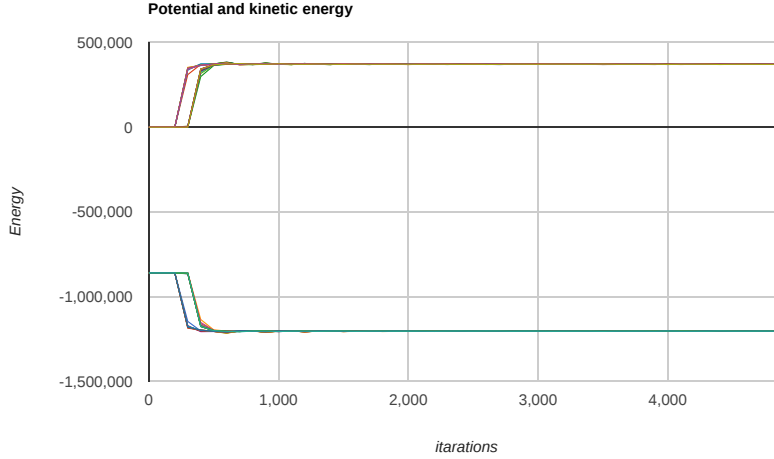


Figure 6.2: Kinetic and potential energy profile for OpenFPM and LAMMPS with different number of processors. Initially the particles are on a grid in an unstable equilibrium, due to numerical error the particles start to move. On the x axis we have number of iteration and on y axis we have dimensionless kinetic energy E_{kin} for the positive curve and dimensionless potential energy E_{pot} for the negative curve. Particles initially does not move, $E_{kin} = 0$, and energy potential is around $-8.6e5$ after 100 iterations for LAMMPS and 150 for OpenFPM steps particles start to move and part of the potential energy is transformed into kinetic until particles stabilize at around $E_{kin} = 3.7e5$ and $E_{pot} = -12.041e5$

runs for 5000 time steps and the final particle configuration is in the right panel of figure 6.1.

The temporal evolution of the kinetic, potential, and total energies calculated by LAMMPS and OpenFPM were the same as shown in the figure 6.2. The total energy is conserved, showing that the simulation was consistent with the principle of conservation of energy. Equilibration is reached after approximately 1000 time steps.

#cores	OpenFPM (seconds)	LAMMPS (seconds)	OpenFPM (Efficiency)	LAMMPS (Efficiency)
1	1010.69 \pm 1.58	976.10 \pm 3.30	100%	100%
4	262.55 \pm 0.80	257.00 \pm 6.48	96.2%	94.9%
8	143.81 \pm 0.26	137.10 \pm 0.31	87.8%	89.0%
16	77.10 \pm 0.40	73.00 \pm 0.46	81.9%	83.6%
24	52.70 \pm 0.27	49.89 \pm 0.17	79.9%	81.5%
48	29.70 \pm 0.12	28.98 \pm 0.22	70.9%	70.2%
96	15.16 \pm 0.11	15.64 \pm 0.60	69.4%	65.0%
192	8.07 \pm 0.16	8.22 \pm 0.32	65.2%	61.8%
384	4.73 \pm 0.16	4.66 \pm 0.17	55.6%	54.5%
768	3.15 \pm 0.09	3.37 \pm 1.10	41.8%	37.7%
1536	2.20 \pm 0.24	1.93 \pm 0.77	29.9%	32.9%

Table 6.2: Molecular dynamics benchmark results. We report wall-clock execution times (mean \pm standard deviation over 10 independent runs) and parallel efficiencies of the OpenFPM client compared with LAMMPS [23] for a strong scaling from 1 to 1536 processor cores simulating 216,000 Lennard-Jones particles in the unit cube over 5000 time steps (see figure 6.1).

#cores per socket \times #sockets	OpenFPM (seconds)
8×1	147.4
4×2	143.6
2×4	133.5
1×8	128.0

Table 6.3: Average runtime of the OpenFPM molecular dynamic code using different numbers of cores for the same total of 8 cores for the problem from Table 6.2

We identified the case of the drop in efficiency on a single node. We took the case of 8 cores in table 6.2 this is the case where we have 4 cores on 2 sockets, equivalent to the case 4×2 in the table 6.3. Increasing memory/L3 bandwidth, equivalent to shift down in the table 6.3 we can see how this brings efficiency from 87.9

As illustrated in Listing 6.0.1, the OpenFPM-based molecular dynamic simulation can be implemented in less than 40 lines of C++ code without comments. Lines 10–15 define the pairwise Lennard-Jones interaction between the particles in the example listing. Lines 19–30 define the size of the simulation domain, in this case, a cube of unit size, the boundary conditions, periodic in all three dimensions, and the size of the ghost layers, determined by the interaction cut-off radius `r_cut`. Line 33 creates the particle interaction object based on the definition of the interaction in lines 10–15. It is important to consider that the macro `DEFINE_ITERATION_3D` is syntactic sugar for defining a functor named `ln_force`. Line 36 creates a particle set on the domain and ghost area defined in the previous lines. The particle properties are defined as a tuple or aggregate described in section 2.2.2. For this simulation, we need two 3D points/vectors of doubles to store the velocity and the force. The particle position comprises three

doubles defined by the dimensionality (first template parameter) and precision used for the space (second template parameter). In line 37, we initialize the particles on a cartesian lattice, with the number of grid points defined in each direction by `sz` defined in line 27. Lines 41–43 provide aliases for particle position, velocity, and force explained in section 5.0.1 that can be used to generate computation on particles with statements like the one on line 58. Line 50 generates symmetric particle cell lists for quick neighbor access, which are then used in line 51 to compute the initial forces using symmetric particle interactions. `applyKernel_in_sym` uses `lennard_jones` to calculate forces in a symmetric manner. In particular we calculate the force of a pair interaction `lennard_jones(A,B)` once avoiding the calculation of `lennard_jones(B,A) = -lennard_jones(A,B)`. Omitting `_sym` the computation would use non-symmetric interaction computation. The simulation time loop uses the two-step velocity Verlet symplectic time integrator on lines 54-73. (lines 58, 59, 72). Communication procedures explained in section 4.2.1 are in lines 63 and 64. Line 63 ghost-get synchronize particle coordinates for only the particle coordinates. Line 64 does mapping to migrating particles that have moved across processor boundaries. At the end of the program, line 76 closes the OpenFPM library.

We compared the performance of the OpenFPM with LAMMPS for strong scaling using Verlet lists, distributing the 216,000 particles across an increasing number of processors. The result are given in Table 6.2. On a single-core, the absolute wall-clock time each time step is less than one second. A simulation time step takes 0.5 milliseconds on 1536 cores. Even though OpenFPM is a general-purpose particle-mesh library not limited to MD, its performance is nearly identical to that of LAMMPS.

Listing 4.1: C++ code for Lennard-Jones molecular dynamics using OpenFPM]

```

1  // define parameters
2  double sigma12, sigma6, epsilon = 1.0, sigma = 0.1; // parameters of the
   potential
3  double dt = 0.0005, r_cut = 3.0*sigma; // parameters of the
   simulation
4  double r_cut2;
5
6  constexpr int velocity_prop = 0; // velocity is the first particle property
7  constexpr int force_prop = 1; // force is the second particle property
8
9  // Define Lennard-Jones interaction to be used in applyKernel_in_sym
10 DEFINE_INTERACTION_3D(ln_force)
11     Point<3,double> r = xp - xq;
12     double rn = norm2(r);
13     if (rn > r_cut2) return 0.0;
14     return 24.0*epsilon*(2.0*sigma12/(rn*rn*rn*rn*rn*rn)-sigma6/(rn*rn*rn*rn))
   *r;
15 END_INTERACTION
16
17 int main(int argc, char* argv[]) {
18     // Initialize OpenFPM
19     openfpm_init(&argc,&argv);
20
21     // Initialize constants
22     sigma6 = pow(sigma,6), sigma12 = pow(sigma,12);
23     r_cut2 = r_cut*r_cut;
24
25     // Define initialization grid, simulation box, periodicity

```

```

26  // and ghost layer
27  size_t sz[3] = {60,60,60};
28  Box<3,float> box({0.0,0.0,0.0},{1.0,1.0,1.0});
29  size_t bc[3]={PERIODIC,PERIODIC,PERIODIC};
30  Ghost<3,float> ghost(r_cut);
31
32  // Lennard-Jones potential object used in applyKernel_in
33  ln_force lennard_jones;
34
35  // Define particles and initialize them on a grid
36  vector_dist<3,double,aggregate<Point<3,double>,Point<3,double>>> particles(0,
    box,bc,ghost);
37  Init_grid(sz,particles);
38
39  // Define aliases for the particle force, velocity, and position
40  // to simplify notation
41  auto force = getV<force_prop>(particles);
42  auto velocity = getV<velocity_prop>(particles);
43  auto position = getV<PROP_POS>(particles);
44
45  // initialize all particle velocities to zero
46  velocity = 0;
47
48  // Generate the cell lists and compute the initial forces using the
    Lennard-Jones
49  // potential evaluated with exploiting symmetry
50  auto NN = particles.getCellListSym(r_cut);
51  force = applyKernel_in_sym(particles,NN,lennard_jones);
52
53  // Time loop
54  for (size_t i = 0; i < 10000 ; i++) {
55      // 1st step of velocity Verlet time integration
56      //  $v(t + 1/2*dt) = v(t) + 1/2*force(t)*dt$ 
57      //  $x(t + dt) = x(t) + v(t + 1/2*dt)$ 
58      velocity = velocity + 0.5*dt*force;
59      position = position + velocity*dt;
60
61      // communicate particles that have crossed processor boundaries and
62      // update the ghost layers for all properties (empty props list)
63      particles.map();
64      particles.ghost_get<>();
65
66      // Calculate the forces at  $t + dt$ 
67      particles.updateCellListSym(NN);
68      force = applyKernel_in_sym(particles,NN,lennard_jones);
69
70      // 2nd step of velocity Verlet time integration
71      //  $v(t+dt) = v(t + 1/2*dt) + 1/2*force(t+dt)*dt$ 
72      velocity = velocity + 0.5*dt*force;
73  }
74
75  // Finalize OpenFPM and deallocate all memory
76  openfpm_finalize();
77 }

```

6.0.2 Smoothed-particle hydrodynamics

SPH (Smoothed-Particle Hydrodynamics) is a popular approach for simulating continuous fluid dynamics models. It is the preferred method to represent multi-phase flows and fluid-structure interaction [69],[70] due to its simplicity and flexibility in simulating complicated fluid properties and free fluid surfaces.

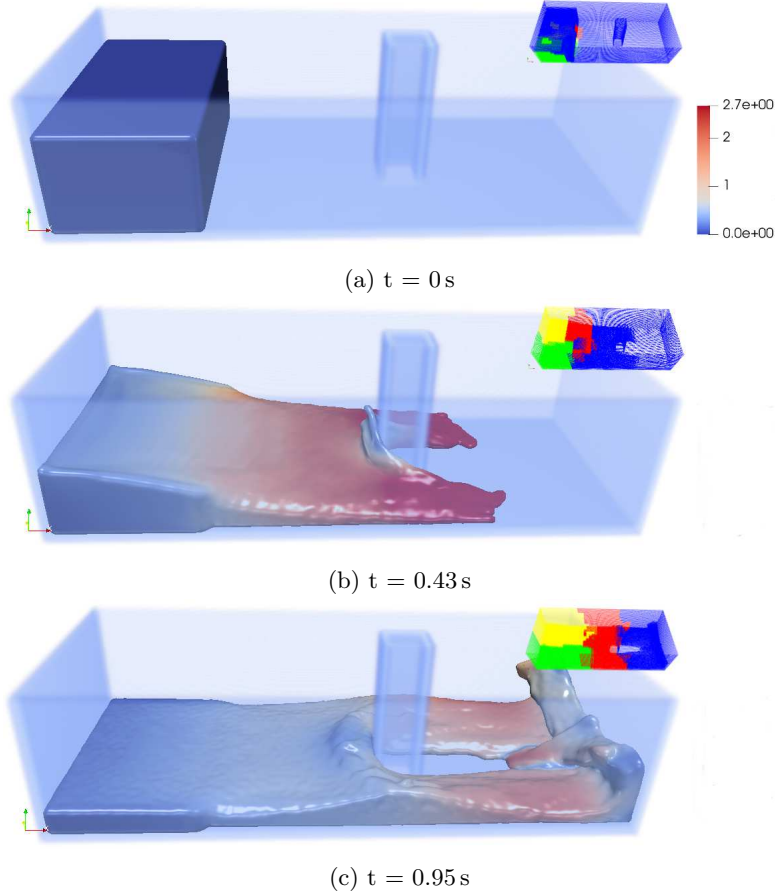


Figure 6.3: Visualization of the SPH dam-break simulation. We show the fluid particles at times 0, 0.43, and 0.95 s of simulated time, starting from a column of fluid in the left corner of the domain as shown. We use the OpenFPM SPH to solve the weakly compressible Navier-Stokes equations with the equation of state for pressure as given in Eqs. 6.2–6.4. The figure shows a density iso-surface indicating the fluid surface with color indicating the fluid velocity magnitude. The small insets show the distribution of the domain onto 4 processors with different processors shown by different colors. The dynamic load balancing of OpenFPM automatically adjusts the domain decomposition to the evolution of the simulation in order to maintain scalability.

We use OpenFPM to develop a weakly compressible SPH Navier-Stokes solver, where each particle p has a velocity \mathbf{v}_p , a pressure P_p , and a density ρ_p . [71]: The evolution of these particle attributes is determined by

$$\frac{d\mathbf{v}_p}{dt} = - \sum_{q \in \mathcal{N}(p)} m_q \left(\frac{P_p + P_q}{\rho_p \rho_q} + \Pi_{pq} \right) \nabla W(\mathbf{x}_q - \mathbf{x}_p) + \mathbf{g} \quad (6.2)$$

$$\frac{d\rho_p}{dt} = \sum_{q \in \mathcal{N}(p)} m_q \mathbf{v}_{pq} \cdot \nabla W(\mathbf{x}_q - \mathbf{x}_p) \quad (6.3)$$

$$P_p = b \left[\left(\frac{\rho_p}{\rho_0} \right)^\gamma - 1 \right] \quad (6.4)$$

$$b = \frac{1}{\gamma} c_{\text{sound}}^2 |\mathbf{g}| h_{\text{swl}} \rho_0, \quad (6.5)$$

where h_{swl} is the fluid's maximum height, $\gamma = 7$, and $c_{\text{sound}} = 20$ [71]. The set of all particles within a cutoff radius of $2\sqrt{3}h$ from p is called $\mathcal{N}(p)$, and h is the distance between nearest neighbors at initialization stage. The standard cubic SPH kernel [71] is $W(\mathbf{x})$, and the gravitational acceleration is \mathbf{g} . The relative velocity between particles p and q is $\mathbf{v}_{pq} = \mathbf{v}_p - \mathbf{v}_q$, where $\nabla W(\mathbf{x}_q - \mathbf{x}_p)$ is the analytical gradient of the kernel W centered at particle p with the neighborhood particle q . The equation of state 6.4 connects the pressure P_p to the density ρ_p , where ρ_0 is the density of the fluid at $P = 0$. The viscosity term Π_{pq} is defined as:

$$\Pi_{pq} = \begin{cases} -\frac{\alpha c_{pq}^- \mu_{pq}}{\rho_{pq}} & \mathbf{v}_{pq} \cdot \mathbf{r}_{pq} > 0 \\ 0 & \mathbf{v}_{pq} \cdot \mathbf{r}_{pq} < 0 \end{cases} \quad (6.6)$$

with constants defined as: $\mu_{pq} = \frac{h \mathbf{v}_{pq} \cdot \mathbf{r}_{pq}}{r_{pq}^2 + \eta^2}$ and $c_{pq}^- = \sqrt{g \cdot h_{\text{swl}}}$.

In the simulation a column of water collides with a fixed obstacle using OpenFPM. The 'dam break' simulation is a standard SPH test case. Figure 6.3 shows a depiction of the OpenFPM result at three different time points. The performance are compared to those achieved with DualSPHysics [28], a famous open-source SPH code. We show comparisons in the non-distributed case since the publicly available version of DualSPHysics only supports shared-memory multi-core systems and GPGPUs.

The current OpenFPM SPH implementation employs the same techniques as DualSPHysics[28], including the same initialization, boundary conditions, viscosity term treatment, and Verlet time-stepping [48] with a dynamic step size. For this reason, the outcomes between OpenFPM and DualSPHysics are directly comparable. In the 'dam break' case, particles are not evenly distributed around the domain, and they move around during the simulation. As a result, we need to use dynamic load balancing explained in section 3.2.

The simulation is validated by calculating and comparing the velocity and pressure profiles at multiple points between OpenFPM and DualSPHysics [28]. Pressure and velocity profiles show similar behaviour between OpenFPM and

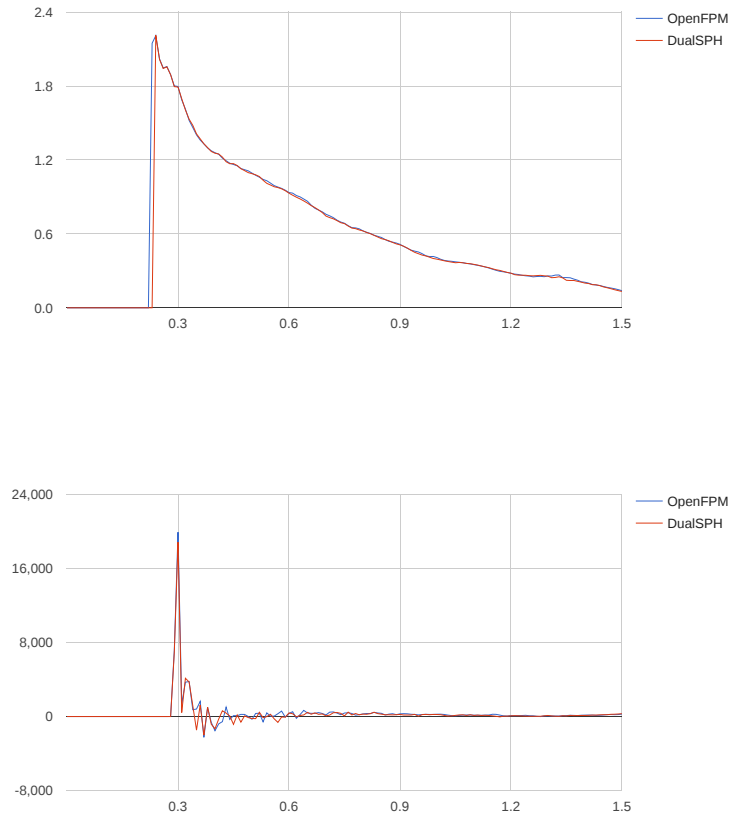


Figure 6.4: **UP** Pressure profile on one point in front of the obstacles OpenFPM and DualSPH. **DOWN** Velocity profile in the same point OpenFPM and DualSPH

DualSPHysic in figure 6.4. The simulation do all the calculations in float precision because the option for DualSPH only increases the precision of the particle position. However, most calculations are still performed in single precision. In OpenFPM, it is also possible to hybrid the computation choosing the precision for the space to be double and the precision for the property to be float. Unfortunately, to ensure every operation is done with the same precision on both platform, we should track every single operation. For the sake of simplicity in the comparison, all the computations are homogeneous in single precision.

We compare the performance of the OpenFPM-based solution to the DualSPHysics code operating on a single cluster node's 24 cores. With 171,496 particles. We simulate the dam-break case till a physical time of 1.5 seconds. The OpenFPM code takes about 500 seconds to complete the simulation, while DualSPHysics takes around 950 seconds. OpenFPM improved performance can be attributed to the use of symmetry when analyzing interactions and the usage of optimized Verlet lists, both of which are not apparent in DualSPHysics[28].

We also profiled this test scenario regarding how much time is spent computing, communicating, and load-balancing. The results for various numbers of particles on 1536 processors are shown in Table 6.4, illustrating the code's scalability to large numbers of particles. The little insets in figure 6.3 all demonstrate how OpenFPM's domain decomposition dynamically adapts to shifting particle distributions via dynamic load balancing, as detailed in 3.2. The load distribution, in this case, varies substantially due to the velocity distribution of the particles. The dynamic load-balancing mechanisms used by OpenFPM consume between 5% and 25% of overall execution time. The ParMETIS [62] performance is restricting the overall performance. As a result, OpenFPM reduces the impact of communication and load imbalance on parallel performance.

The primary overhead is the graph partitioning stage required by the load balancing, which is done in a distributed fashion but unfortunately does not scale with the number of processors. As a result, the relative amount of communication and load-balancing decreases as the number of particles rises, while the average imbalance remains practically constant due to dynamic load balancing. Load balancing and communication are not required when running on a single core. We can then consider the percentage of time spent calculating (second column in Table 6.4) as the parallel efficiency on 1536 cores.

We also compare the OpenFPM-based implementation in a distributed-memory model with DualSPHysics operating on a GPGPU. DualSPHysics is primarily tuned for use on GPGPUs. We are running on an Nvidia GeForce GTX1080 GPU, the benchmark run with 15 million SPH particles. When the OpenFPM code runs on the benchmark machine's 270 CPU cores, it reaches the same speed.

We also ported the code from CPU code to multi-architecture GPU/CPU in OpenFPM. The port is based mainly on the following changes:

- copy-pasting the CPU inner code loop into kernel functions.
- Convert CPU Cell lists into GPU versions, that mainly provides the same API as the CPU.

- We ask to OpenFPM to run functionalities like `ghost_get` directly on GPU with the option `RUN_ON_DEVICE`
- We make use of GPU aware primitives to remove particles in parallel on GPU

The code is around 531 C++ lines of code and uses around 17 million particles. Compared to DualSPHysics, the code is distributed, and on 1 GPU OpenFPM is only 27% slower, efficiency with different numbers of GPUs are given in figure 6.5.

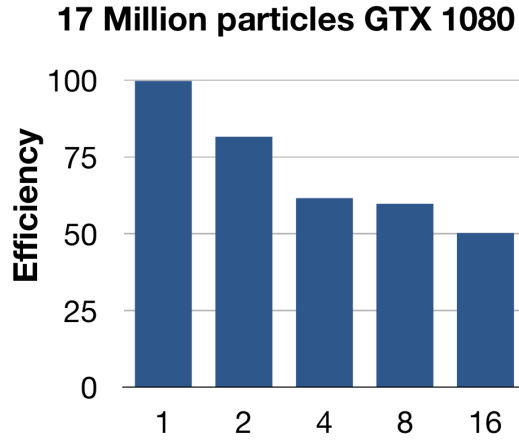


Figure 6.5: Efficiency of SPH gpu on multiple GPUs 16 gpus are 8 nodes with 2 gpus each

N particles	Computation (%)	Imbalance (%)	DLB (%)	Communication (%)	Time (s)
0.46M	19.1963%	15.1259%	25.6729%	40.005%	148.397
1.20M	32.3121%	30.3677%	10.4985%	26.8217%	346.83
4.0M	49.9799%	22.3809%	10.6147%	17.0246%	1424.64
10.78M	63.9155%	21.1935%	6.805%	8.08599%	4666.78
14.63M	65.4522%	21.4576%	5.37555%	7.71468%	7035.01

Table 6.4: OpenFPM’s percentage of total runtime spent on various jobs for the SPH dam-break simulation on 1536 cores with various quantities of particles (1st column). The computation time is the average wall-clock time spent on local computations across processors, whereas the load imbalance is the difference between the maximum and average wall-clock time across processors. DLB (dynamic load balancing) is the time it takes to breakdown the problem and allocate sub-domains to processors, and communication is the time it takes for `ghost_get` and `map` combined. The last column shows the simulation’s overall runtime up to a simulated time of 1.5,s.

6.0.3 Finite-difference reaction-diffusion code

As a third showcase, we explore a mesh-only-based application, namely a finite-difference code, to solve a reaction-diffusion system numerically. Because of their potential to create concentration patterns, such as Turing patterns (Turing:1952), reaction-diffusion systems are intensively investigated. A well-known example is the Gray-Scott system ([72],[73],[74],[75]), which produces a wide range of patterns under different parameter regimes. The partial differential equations that describe it are as follows:

$$\begin{aligned}\frac{\partial u}{\partial t} &= D_u \nabla u - uv^2 + F(1 - u) \\ \frac{\partial v}{\partial t} &= D_v \nabla v + uv^2 - (F + k)v,\end{aligned}\tag{6.7}$$

The diffusion constants of the two species u and v , respectively, are D_u and D_v . The type of pattern created is determined by the parameters F and k . We use second-order centered finite-differences on a regular Cartesian mesh in 3D of size 2563 with periodic boundary conditions in all directions to develop an OpenFPM-based numerical solver for these equations. The performance of the OpenFPM-based implementation is compared to that of [35], an efficient AMReX-based solution.

Even though AMReX is capable of multi-resolution adaptive mesh refinement, which would make a straight comparison unfair, it uses a patch-based technique, which is incredibly fast even in the isotropic scenario when constrained to one level. Indeed, the implementation of one-level patches in AMReX is extremely similar to that of OpenFPM meshes.

AMReX, on the other hand, requires that the user adjusts the maximum grid size for data distribution [35]. AMReX does not have enough granularity to parallelize if the maximum grid size is set too high. If it is set too low, scalability suffers as a result of increased ghost-layer communication overhead. This AMReX value was carefully determined to ensure that sub-grids are always greater than the number of processor cores. Tables 6.5 and 6.6 show the actual values used in the last columns. The domain decomposition in OpenFPM is determined automatically. Therefore the user does not need to set this parameter. In order to compare the results of AMReX with OpenFPM, we use MPI-only parallelism.

The following parameter values are used in the benchmark simulations: $D_u = 2 \cdot 10^{-5}$, $D_v = 10^{-5}$, with k and F as indicated in the legends of figure 6.6. We repeat the nine patterns described by Pearson [76] using visualizations shown in Fig 6.6 to validate the simulation.

Listing 6.0.3 shows an OpenFPM source-code example of applying a simple 5-point finite-difference stencil to a standard Cartesian mesh. In line 2, an OpenFPM grid key array with relative grid coordinates is defined as the stencil. The stencil object, in this case, is called `tstar` stencil 2D and has 5 points. Line 5 creates a mesh iterator for this stencil, which is applied to the mesh object

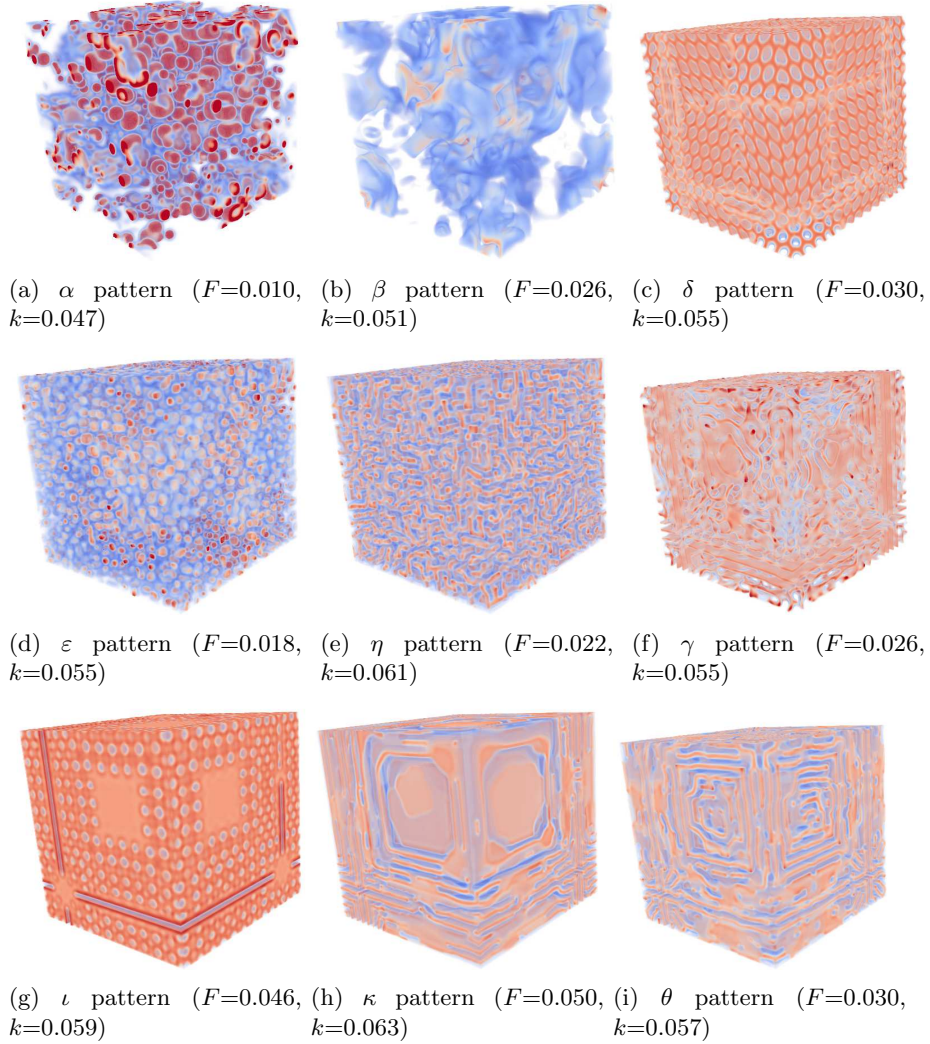


Figure 6.6: Visualizations of the OpenFPM-simulations of nine steady-state patterns produced by the Gray-Scott reaction-system in 3D [76] for different values of the parameters F and k .

01d. Lines 7–24 define the loop over all mesh nodes, and the stencil is applied. The stencil expression (lines 18–20) is made easier by declaring aliases for the shifted nodes in lines 10–14.

Listing: 4.3: OpenFPM code example for stencil operations on a regular Cartesian mesh]

```
1  ///// finite-difference stencil definition
2  static grid_key_dx<2> star_stencil_2D[5] = {{0,0},{-1,0},{+1,0},{0,-1},{0,+1}};
```

```

3
4  ///// create an iterator for the stencil on the mesh "Old"
5  auto it = Old.getDomainIteratorStencil(star_stencil_2D);
6
7  while (it.isNext()) {
8      ///// define aliases for center, minus-x, plus-x, minus-y, plus-y.
9      ///// The template parameter is the stencil element.
10     auto Cp = it.getStencilGrid<0>();
11     auto mx = it.getStencilGrid<1>();
12     auto px = it.getStencilGrid<2>();
13     auto my = it.getStencilGrid<3>();
14     auto py = it.getStencilGrid<4>();
15
16     ///// apply the stencil to field U on mesh "Old" and store
17     ///// the result in the field U on mesh "New"
18     New.get<U>(Cp) = Old.get<U>(Cp) +
19         (Old.get<U>(my)+Old.get<U>(py)+Old.get<U>(mx)+Old.get<U>(px) -
20          4.0*Old.get<U>(Cp));
21
22     ///// Move to the next mesh node
23     ++it;
24 }

```

Table 6.5 6.6 and figure 6.7 show the performance of OpenFPM compared to AMReX for two distinct sizes. With wall-clock times in the same range as AMReX, OpenFPM scales marginally better. We use up to 24 cores within one cluster node for the task of 256^3 mesh nodes (Table 6.5). As a result, this benchmark demonstrates how the codes scale in terms of memory bandwidth. The bigger problem of 784^3 mesh nodes (Table 6.6) spans several computer nodes and demonstrates how the algorithms scale with respect to network communication overhead. For this benchmark, both AMReX and OpenFPM employ a mix of C++ and Fortran code, with all stencil iterations written in Fortran. Fortran produces more efficient assembly code than C++ because it has native support for multi-dimensional arrays. A completely C++ version was around 20% slower in our tests than a mixed C++/Fortran version. In addition, we show how the OpenFPM code's runtime changes when employing alternative distributions of the same total number of cores over different numbers of CPU sockets in Table 6.7. The memory bus saturates when employing more than 4 cores per socket, as seen in Table 6.1, and runtimes considerably increase.

6.0.4 Vortex Methods

This benchmark consider a complete vortex-in-cell algorithm. A hybrid particle-mesh approach [50] to numerically solve the incompressible Navier-Stokes equations in vorticity formulation with periodic boundary conditions, to highlight how OpenFPM handles hybrid particle-mesh problems. These are the equations:

$$\begin{aligned}
 \frac{D\omega}{Dt} &= (\omega \cdot \nabla)\mathbf{u} + \nu\Delta\omega \\
 \Delta\psi &= \nabla \times \mathbf{u} = \omega,
 \end{aligned}
 \tag{6.8}$$

ω is the vorticity, ψ is the vector stream function, ν is the viscosity, and \mathbf{u} is the fluid velocity. [50] The operator $\frac{D}{Dt}$ specifies a Lagrangian (material)

#cores	OpenFPM (seconds)	AMReX (seconds)	AMReX param
1	393.1 ± 1.3	388.5 ± 1.5	256
2	207.5 ± 1.3	265.0 ± 0.8	128
4	105.8 ± 1.3	144.8 ± 0.3	128
8	65.1 ± 2.1	106.6 ± 2.6	128
12	65.6 ± 2.6	90.9 ± 5.0	64
16	57.6 ± 1.9	173.6 ± 3.6	64
20	56.8 ± 2.0	66.0 ± 1.7	64
24	60.5 ± 0.3	60.9 ± 4.0	64

Table 6.5: Performance of the OpenFPM finite-difference code compared with AMReX [35]. Times are given in seconds as mean \pm standard deviation over 10 independent runs for a fixed problem size of 256^3 mesh nodes over 5000 time steps (strong scaling). The grid-size parameters used for AMReX are given in the last column.

#cores	OpenFPM (seconds)	AMReX (seconds)	AMReX param
1	199.7 ± 0.4	205.5 ± 1.4	740
4	52.5 ± 0.3	56.1 ± 0.3	370
8	32.0 ± 0.1	39.7 ± 0.1	370
16	15.2 ± 0.1	28.1 ± 0.1	294
32	8.4 ± 0.1	13.8 ± 0.1	233
64	4.3 ± 0.1	6.8 ± 0.1	185
128	3.1 ± 0.1	4.33 ± 0.2	147
256	2.3 ± 0.1	2.7 ± 0.1	117

Table 6.6: Performance of the OpenFPM finite-difference code compared with AMReX [35]. Times are given in seconds as mean \pm standard deviation over 5 independent runs for a fixed problem size of 784^3 mesh nodes over 100 time steps (strong scaling). The grid-size parameters used for AMReX are given in the last column.

#cores per socket \times #sockets	OpenFPM (seconds)
8×1	61.03
4×2	32.0
2×4	25.53
1×8	24.2

Table 6.7: Average runtime of the OpenFPM finite-difference code using different numbers of cores per socket for the same total of 8 cores for the large problem from Table 6.6.

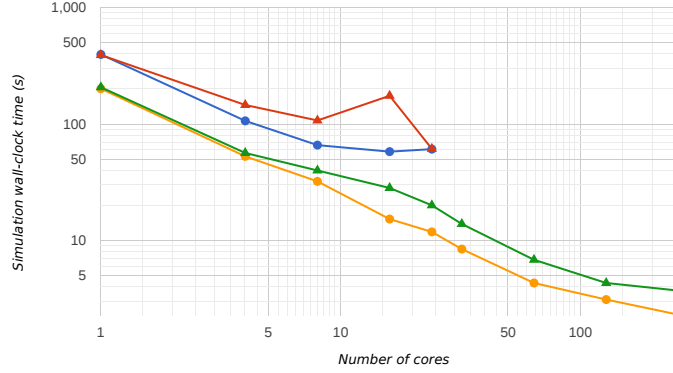


Figure 6.7: OpenFPM Gray-Scott finite-difference code (5-point stencil) code scalability in comparison to AMReX [35] for strong scaling. OpenFPM scalability on a single node 256^3 (Table 6.5) and multiple nodes 784^3 (Table 6.6) problems are represented by blue and yellow lines, respectively, where AMReX single node and multiple nodes are represented by red and green lines. On a 256^3 uniform Cartesian grid with varied numbers of cores, we report the wall-clock time in seconds to complete 5000 time steps, for the 784^3 we report the time to run 100 time steps

time derivative. With two-stage Runge-Kutta time-stepping, we numerically solve these equations using an OpenFPM-based implementation of the classic vortex-in-cell method as provided in Algorithm 22. The moment-conserving M'_4 interpolation kernel is used in particle-mesh and mesh-particle interpolations ([71]).

We run a simulation that replicates earlier self-propelling vortex ring results [77]. The vortex ring is created on a grid $1600 \times 400 \times 400$.

$$\omega_0 = \frac{\Gamma}{\pi\sigma^2} e^{-s/\sigma}, \quad (6.9)$$

where $s^2 = (z - z_c)^2 + [(x - x_c)^2 + (y - y_c)^2 - R^2]$, $R = 1$, $\sigma = R/3.531$, and the domain $(0 \dots 5.57, 0 \dots 5.57, 0 \dots 22.0)$ with periodic boundary conditions. The original vortex ring was defined by $\Gamma = 1$ and $x_c = y_c = z_c = 2.785$ as the center of the torus.

The time-stepping strategy is Runge-Kutta of order 2 with a fixed step size of $\Delta t = 0.0025$. The time step size is determined empirically, ensuring stability qualitatively.

All differential operators are discretized using second-order symmetric finite differences on the mesh. To model the behavior of the vortex ring at Reynolds number $Re = 3750$ till final time $t = 225.5$, we employ 256 million particles

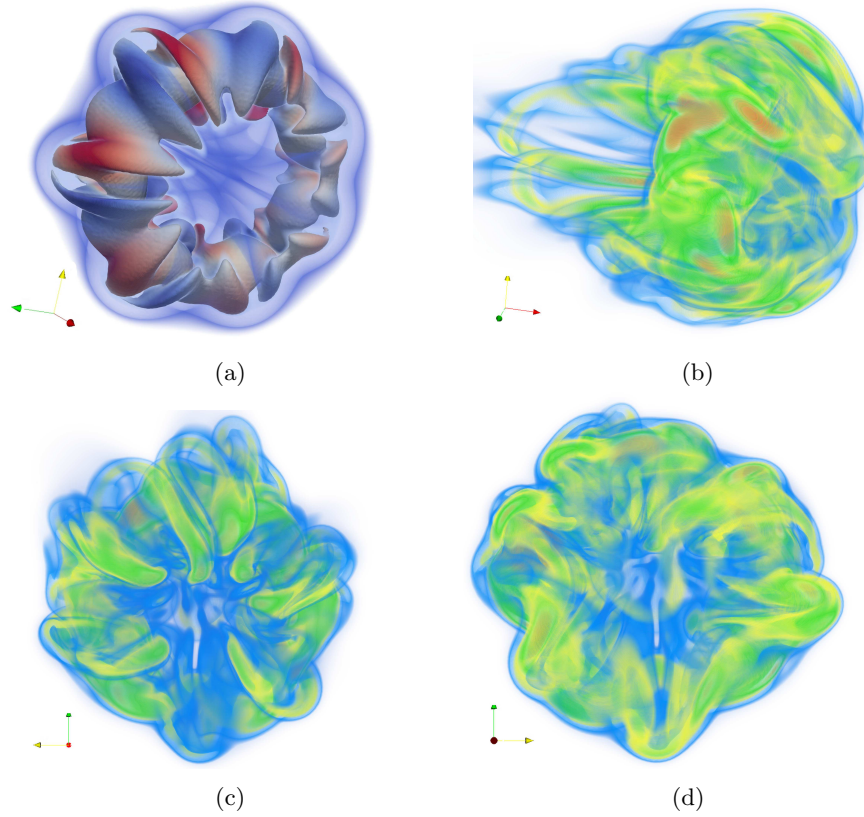


Figure 6.8: Visualization of the OpenFPM simulation of a vortex ring at $Re=3750$ with 256 million particles on 3072 processors using a hybrid particle-mesh Vortex Method (Algorithm 22) to solve the incompressible Navier-Stokes equations. When the ring is going to become turbulent, the results are depicted for $t = 195.5$. (a) The vorticity iso-surfaces emphasize the tubular dipole structures in the vortex ring. The color correlates to the vorticity's x component, with red suggesting positive and blue suggesting negative vorticity. (b)–(d) Three different perspectives of a volume rendering of four vorticity bands: orange represents $\|\omega\|^2 = 3.239 \dots 2.3$, green represents $\|\omega\|^2 = 1.16 \dots 1.372$, yellow represents $\|\omega\|^2 = 0.7 \dots 0.815$, and blue represents $\|\omega\|^2 = 0.3 \dots 0.413$.

dispersed over 3072 processors. OpenFPM creates VTK files, which can be viewed directly in Paraview [68]. In the ring, we see the same patterns and structures as in Ref.[77], as shown in figure 6.8.

To solve the Poisson equation, the linear system solver requires computing the velocity from the vorticity on the mesh. Solving the linear system is where the simulation spends most of the time. For this reason the performance and scalability of the OpenFPM code are mainly given by the linear solver. Inter-

nally, OpenFPM employs a solution supplied by the PetSc library [55]. We test the solver and overall code parallel scalability in a weak scaling, starting with an $109 \times 28 \times 28$ mesh on 1 processor and scaling up to $1207 \times 317 \times 317$ mesh nodes on 1536 processors. The PetSc solver and the OpenFPM components of the code (particle-mesh/mesh-particle interpolation, remeshing, temporal integration, right-hand side evaluation) are benchmarked separately. The results are shown in 6.9 in figure 6.9. The decline in efficiency inside a cluster node (1...24 cores) can be explained by the shared memory bandwidth (see Table 6.1). When switching from one cluster node to two cluster nodes, PetSc displays yet another significant loss in efficiency (48 cores). Following that, the efficiency remains consistent until 768 cores, at which point it begins to decline again gradually.

We compare the particle-mesh interpolation element of the code to the comparable section of a PPM-based hybrid particle-mesh vortex code previously employed [44] to put these results in context. This code section is simply compared to rule out discrepancies between PetSc and PPM internal linear solver. Using the M'_4 interpolation kernel, interpolating two million particles to a $129 \times 65 \times 65$ mesh takes 0.078s in OpenFPM on a single core, and it takes the same amount of time in PPM on the same computer.

The OpenFPM particle-mesh interpolation achieves a parallel efficiency of 75% on 128 cores after performing a mild scaling starting from a 128^3 mesh on 1 CPU (16 nodes using 8 cores of each node). This is equivalent to PPM's scalability on the same test case (see Ref. [44], figure 13).

6.0.5 Discrete element methods

Discrete element methods (DEM) help study granular materials, especially for deriving effective macroscopic dynamics when the governing equations are unknown. They explicitly model each grain of the material, with all collisions resolved. The dynamic of the grains is governed by the forces and torque calculation for each grain. The fundamental distinction from MD is that forces are only produced through direct contact, and contact sites are subject to elastic deformation. As a result, DEM is simply a collision-detection technique. Lists of interaction sites between particles must be managed in order to integrate the dynamics throughout time accurately. Parallelizing DEM is difficult since these lists are of different lengths in both time and space, and collisions involving ghost particles must be correctly accounted for in the lists of the corresponding source particles. DEM has previously been parallelized on distributed-memory machines using the PPM Library [44], allowing DEM simulations of 122 million elastic spheres to be distributed across 192 processors [78].

In order to directly compare performance with the previous PPM version, we implement the identical DEM simulation in OpenFPM. We pre-allocate the contact list to their maximum length for comparison. In OpenFPM, dynamic lists can potentially bypass the limit to bound the maximum number of contacts.

We use the traditional Silbert grain model [79], which includes Hertzian contact forces and elastic grain deformation, as discussed earlier [78]. The radius R , mass m , and moment of inertia I of all particles are the same. The location

Algorithm 22 Vortex-in-Cell Method with two-stage Runge-Kutta (RK) time integration

```

1: procedure VORTEXMETHOD
2:   initialize the vortex ring on the mesh
3:   do a Helmholtz-Hodge projection to make the vorticity divergence-free
4:   initialize particles at the mesh nodes
5:   while  $t < t_{\text{end}}$  do
6:     calculate velocity  $\mathbf{u}$  from the vorticity  $\boldsymbol{\omega}$  on the mesh (Poisson equation solver)
7:     calculate the right-hand side of Eq. 6.8 on the mesh and interpolate to particles
8:     interpolate velocity  $\mathbf{u}$  to particles
9:     1st RK stage: move particles according to the velocity; save old position in  $\mathbf{x}_{\text{old}}$ 
10:    interpolate vorticity  $\boldsymbol{\omega}$  from particles to mesh
11:    calculate velocity  $\mathbf{u}$  from the vorticity  $\boldsymbol{\omega}$  on the mesh (Poisson equation solver)
12:    calculate the right-hand side of Eq. 6.8 on the mesh and interpolate to particles
13:    interpolate velocity  $\mathbf{u}$  to particles
14:    2nd RK stage: move particles according to the velocity starting from  $\mathbf{x}_{\text{old}}$ 
15:    interpolate the vorticity  $\boldsymbol{\omega}$  from particles to mesh
16:    create new particles at mesh nodes (remeshing)

```

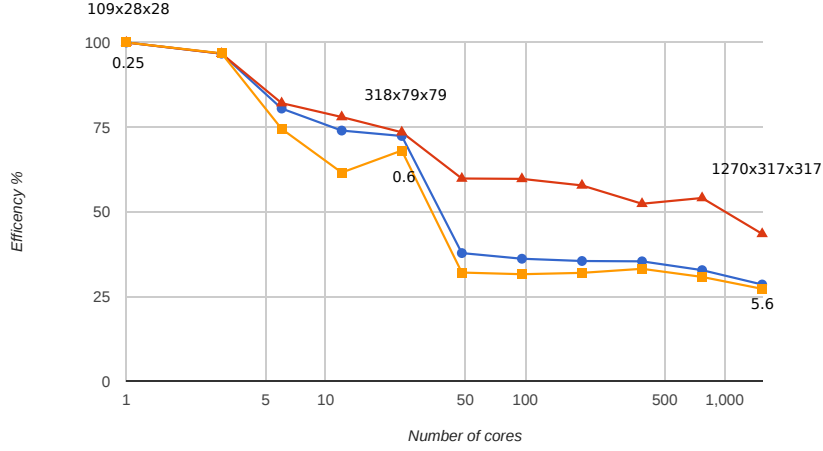


Figure 6.9: Parallel efficiency of the OpenFPM-based hybrid particle-mesh vortex code for a scaled-in-size problem (weak scaling). On a single processor core, the problem scales from $109 \times 28 \times 28$ mesh nodes to $1207 \times 317 \times 317$ mesh nodes on 1536 cores (24 cores per node). We plot individually, the PetSc Poisson solver's parallel efficiency (yellow squares), the OpenFPM components of the code (red triangles), and the overall scalability are all shown (blue circles). Next to the symbols, we give the sizes of the problem for three points and the wall-clock time for one time-step in seconds. In order to compare across runs, we force the linear solver to use a fixed number of iteration

of each particle's center of mass r_p is used to symbolize it. The radial elastic contact deformation occurs when two particles p and q come into touch with each other and is given by:

$$\delta_{pq} = 2R - r_{pq}, \quad (6.10)$$

with $\mathbf{r}_{pq} = \mathbf{r}_p - \mathbf{r}_q$ the vector between the two particle centers and $r_{pq} = \|\mathbf{r}_{pq}\|_2$ its distance.

We use an explicit Euler integration scheme, and the evolution of the tangential elastic deformation $\mathbf{u}_{t_{pq}}$ is integrated as:

$$\mathbf{u}_{t_{pq}} = \mathbf{u}_{t_{pq}} + \mathbf{v}_{t_{pq}} \delta t, \quad (6.11)$$

where δt represents the simulation time step and $\mathbf{v}_{pq} = \mathbf{v}_{t_{pq}} + \mathbf{v}_{n_{pq}}$ is the tangential and radial components of the relative velocity of the colliding particles, respectively. The deformation of the contact points is tracked for each particle, and the elastic tangential displacement is initialized with $\vec{u}_{t_{ij}} = 0$ for each new contact point. The normal and tangential forces for any pair of particles in

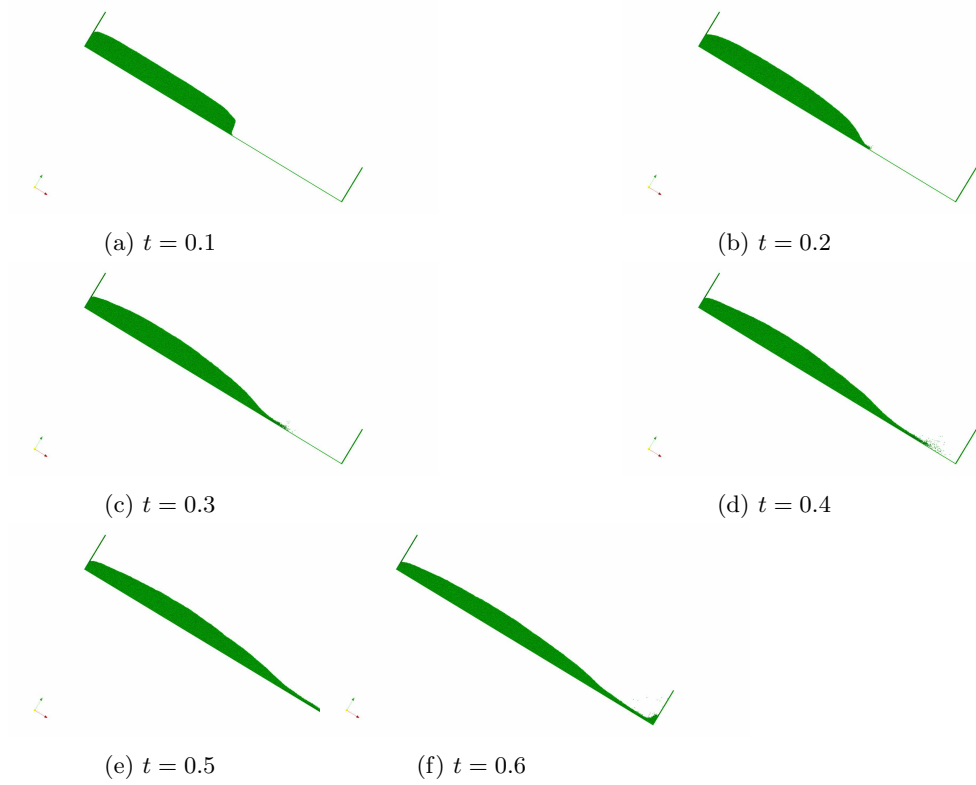


Figure 6.10: Visualization of the Discrete Element Method (DEM) simulation of an avalanche of spheres down an inclined plane (inclination angle: 31.2 degrees) at different times. While the simulation is 3D, we visualize the same 2D cross-section here as in [78] to allow direct visual comparison.

contact with each other are [79]:

$$\mathbf{F}_{n_{pq}} = \sqrt{\frac{\delta_{pq}}{2R}} \left(k_n \delta_{pq} \mathbf{n}_{pq} - \gamma_n m_{\text{eff}} \mathbf{v}_{n_{ij}} \right), \quad (6.12)$$

$$\mathbf{F}_{t_{pq}} = \sqrt{\frac{\delta_{pq}}{2R}} \left(-k_t \mathbf{u}_{t_{pq}} - \gamma_t m_{\text{eff}} \mathbf{v}_{t_{pq}} \right), \quad (6.13)$$

where $k_{n,t}$ are the elastic constants for normal and tangential directions, and $\gamma_{n,t}$ are the friction constants. $m_{\text{eff}} = \frac{m}{2}$ is used to calculate the effective collision mass. In addition, as detailed in [79, 78], the tangential deformation is rescaled to enforce Coulomb's law.

By combining the contributions over all collisions q and the gravitational force vector, we calculate the total resultant force $\mathbf{F}_p^{\text{tot}}$ and torque $\mathbf{T}_p^{\text{tot}}$ on particle p are computed.

We integrate the equations of motion using the second-order leapfrog scheme, as:

$$\mathbf{v}_p^{n+1} = \mathbf{v}_p^n + \frac{\delta t}{m} \mathbf{F}_p^{\text{tot}}, \quad \mathbf{r}_p^{n+1} = \mathbf{r}_p^n + \delta t \mathbf{v}_p^{n+1}, \quad \boldsymbol{\omega}_p^{n+1} = \boldsymbol{\omega}_p^n + \frac{\delta t}{I} \mathbf{T}_p^{\text{tot}}, \quad (6.14)$$

where \mathbf{r}_p^n , \mathbf{v}_p^n , and $\boldsymbol{\omega}_p^n$ are the particle p 's center-of-mass position, velocity, and rotational/angular velocity at time step n , respectively.

We simulate an avalanche down an inclined plane, which has been used as a benchmark case for distributed-memory parallel DEM simulations using the PPM Library [78]. The simulation has 82,300 particles with $k_n = 7.849$, $k_t = 2.243$, $\gamma_n = 3.401$, $R = 0.06$, $m = 1.0$, and $I = 1.44 \cdot 10^{-3}$ shown in figure 6.10. The simulation domain is $8.4 \times 3.0 \times 3.18$ in size. As seen in figure 6.10 a, all particles are initially put on a Cartesian lattice inside a box of size $4.26 \times 3.06 \times 1.26$. The simulation box contains fixed-boundary walls in the x direction, a free-space boundary in the positive z direction, and periodic boundaries in the y direction, and is inclined by 30 degrees by rotating the gravity vector correspondingly.

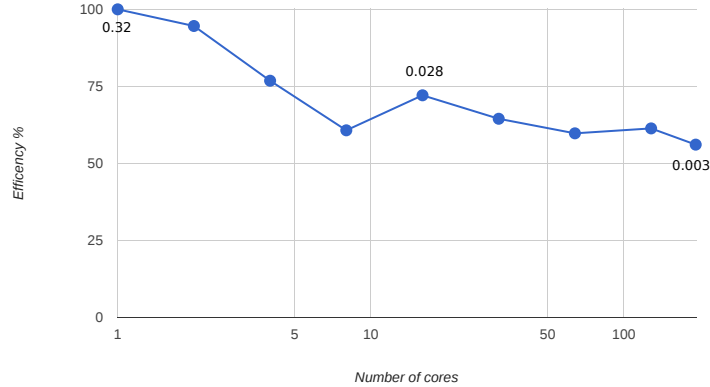


Figure 6.11: Using 677,310 particles divided across 192 cores with 8 cores on each cluster node, the OpenFPM DEM simulation scaled well. The numbers next to the symbols represent the absolute wall-clock time in seconds each time step.

Using the identical test scenario, we evaluate the performance of the OpenFPM DEM with the old PPM code [78]. We display the parallel efficiency of the OpenFPM DEM simulation with strong scaling on up to 192 processors in figure 6.11. OpenFPM takes 0.32 seconds to finish a time step with 677,310 particles on a single core, but the PPM-based code takes 1.0 second per time step with 635,780 particles. OpenFPM completes a time step of the same problem in

3 milliseconds on 192 cores, with a parallel efficiency of 56 percent. On the other hand, the PPM DEM client requires 11 milliseconds per time step on 192 cores and has a parallel efficiency of 47%, according to [78]. The PPM code was tested on a Cray XT-3 system, whose AMD Opteron 2.6,GHz processors are around 3 times slower than the 2.5,GHz Intel Xeon E5-2680v3 of the present benchmark machine, showing that both codes had identical effective performance.

6.0.6 Particle-swarm covariance-matrix-adaptation evolution strategy (PS-CMA-ES)

One of the most significant advantages of OpenFPM over other simulation frameworks is that it can transparently handle spaces of any dimension. This feature allows simulations in higher-dimensional spaces, like in lattice quantum chromodynamics [80, 81] with four and five-dimensional spaces, or non-simulation applications such as image analysis algorithms [82], and Monte-Carlo sampling strategies [83].

The Covariance-Matrix-Adaptation Evolution Strategy (CMA-ES) is a Monte-Carlo sampling technique used for black box optimization [84, 85]. The objective is to find a (local) optimum for a (non-convex) function $f : \mathbb{R}^n \mapsto \mathbb{R}$. The domain's dimensionality n in the tested applications ranges from 10 to 50. CMA-ES has already been parallelized by running numerous instances of the program at the same time that exchange information in the same way that a particle-swarm optimizer does. On multi-funnel functions, the resulting particle-swarm CMA-ES (PS-CMA-ES) has been demonstrated to outperform regular CMA-ES [86], and an efficient Fortran version, pCMAlib [87], is available.

We use OpenFPM to implement PS-CMA-ES in order to show how OpenFPM transparently handles high-dimensional spaces and can also be used for non-simulation tasks like sampling and computational optimization. Each OpenFPM particle corresponds to one CMA-ES instance in our solution, allowing us to implement PS-CMA-ES via particle interactions across processors. We employ the multi-modal test function f_{15} from the IEEE CEC2005 collection of standard optimization test functions to assess the OpenFPM implementation [88]. To compare directly with pCMAlib, we set the maximum number of function evaluations to 5×10^5 and run both implementations 25 times each. In 10, 30, and 50 dimensions, we compare the success rate, which is the fraction of the 25 runs that reached the genuine global optimum, and the success performance, which is the average best function value discovered over all 25 runs [86, 87]. When employing the same pseudo-random number sequence, the findings from the OpenFPM-based implementation are similar to those from pCMAlib (not shown).

We also evaluate the OpenFPM-based implementation's runtime performance and parallel scalability to the highly optimized Fortran pCMAlib. For dimension 50, the findings are shown in figure 6.12. The results for dimensions 10 and 30 are similar and are not shown. This corresponds to robust scaling because the total number of function evaluations is kept constant at 5×10^5 regardless of the number of cores employed. However, because this is a hard

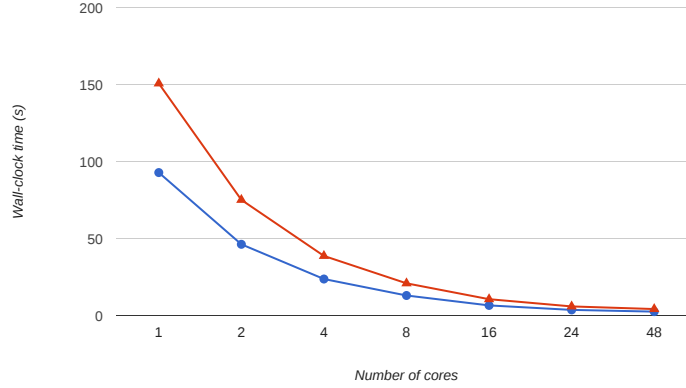


Figure 6.12: Scaling from 1 to 48 cores for IEEE CEC2005 test function f_{15} in dimension 50 using the OpenFPM PS-CMA-ES client (blue circles) in comparison to the Fortran pCMALib (red triangles). The minimum total wall-clock time in seconds for 5×10^5 function executions is shown (over 25 independent repeats).

requirement of pCMALib, the number of swarm particles is always chosen to be equal to the number of cores, but OpenFPM does not. The OpenFPM implementation is about a third quicker than pCMALib in all circumstances.

Because dimensionality is a template parameter in all data structures, implementing arbitrary-dimensional codes with OpenFPM is simple. The PS-CMA-ES data structures in 50 dimensions are defined in OpenFPM using the code example in Listing 6.0.6. All iterators and mappings operate in a transparent manner. This example shows how OpenFPM naturally extends to problems in higher-dimensional spaces, something that the original PPM Library [44] could not do.

Listing 4.6: OpenFPM code example for high-dimensional spaces]

```

1  constexpr int dim = 50;          // define the dimensionality
2
3  ///// Define the optimization domain as (-5:5)^dim
4  Box<dim,double> domain;
5  for (size_t i = 0; i < dim; i++) {
6      domain.setLow(i,-5.0);
7      domain.setHigh(i,5.0);
8  }
9
10  ///// Define periodic boundary conditions
11  size_t bc[dim];
12  for (size_t i = 0; i < dim; i++) {bc[i] = NON_PERIODIC;};
13
14  ///// There are no ghost layers needed for this problem
15  Ghost<dim,double> g(0.0);
16

```

```
17  ///// define the particles data structure
18  vector_dist<dim,double,aggregate<double,double[dim]>> particles(8,domain,bc,g);
19
20  ///// get an iterator over particles and loop over all of them
21  auto it = vd.getDomainIterator();
22  while (it.isNext()) {
23      .....    // do PS-CMA-ES here
24      ++it;
25  }
```


Chapter 7

OpenFPM Development

While in the previous chapters, focused mainly on discussing the design, features, and performance of OpenFPM. This chapter, is going to discuss the development infrastructure. Such infrastructure makes OpenFPM maintainable, keeping high-quality code standards in terms of robustness against multiple compilers, different systems, and hardware. During the development of OpenFPM a lot of lessons has been learned in many aspects regarding its development. In this chapter, will go through some of them.

7.1 Repository structure

As we already saw throughout this thesis, OpenFPM is composed of six repositories. Each of these repositories contains classes developed for a determined purpose. The six repos stack up to create the software stack in figure 1.1. The stack starts from `openfpm_devices` where we have memory allocators, passing through `openfpm_data` where we have single-core data structures. `Openfpm_vcluster`, where we have functionalities to communicate across processes. `Openfpm_pdata` where we have distributed data structures and finally `openfpm_numerics`, where we have numerical algorithms on top of data structures.

Each repo has its tests suites and is tested without the top-level modules. This guarantees that each module can be tested with classes in the same or below repos in the stack. Having vertical development and multi-level testing give robust criteria to develop tests that are useful to detect regressions near the place where they are introduced. It also provides good feedback on the quality and granularity of the tests. Let us imagine we have a failure in the `openfpm_pdata` project tests. Tracking the problem from the failing test, we realize the error requires commits not only in `openfpm_pdata` but also in the project `openfpm_data`. Having commits in two modules gives the feedback that we lack lower level tests in `openfpm_data` that control regressions and correctness of the `openfpm_data` module. Additionally, they enforce an order on the test. Test in data has to pass before checking the test in `openfpm_pdata`

or any other module higher than `openfpm_data`. In the ideal case, a failure in `openfpm_pdata` and not `openfpm_data` should restrict our bug search to the `openfpm_pdata` module. Within the repo, the classes are developed using a test-driven development approach. The test is the starting point that defines the class member functions and their behavior. Bugs are seen as a lack of testing, so fixing a bug start from developing a test that detect the problem failing. The bug is considered fixed when we are able to make pass the test.

An important aspect to consider when creating a test is to avoid redundancies. Avoid redundancies, mean that each test has to control some new code developed on the top of code already tested by other tests. This explain also why giving an order respecting the vertical development is fundamental. Any mechanism to organize tests in suites or even more strict mechanisms like separate repos play a role in the verticality of the development and testing. Every test ideally checks a small section of code on top of already tested code. These are the main criteria used to develop a valuable test suite in OpenFPM.

The rules provided above give criteria to decide when to add a test and when not, providing equilibrium in finding the right amount of testing. Although the rules above are good practice to find a good spot, context must always be considered. A software controlling critical system in which a bug can lead to significant damages like an autopilot or security system requires special consideration and complete solid testing before releasing. The software controlling non-critical systems like a library for simulation can initially release a new feature with less testing and add tests to fix bugs later on.

Templated code makes it difficult to evaluate coverage, but the coverage reported by `gcov` using the full test suite with over 600 test cases is around 90% and 95% or above for each repo. While the overall testing coverage is above 95%

7.2 Build pipeline and full automation

It has been shown the importance of creating a valuable testing suite for the OpenFPM code in the previous section. This chapter will show how to automate the testing and separate tests in stages, giving them priorities. The purpose of running the code under an automated pipeline is to test code robustness against regressions and robustness against multiple compilers, systems, and hardware systematically. The full pipeline is shown in Fig: 7.1

Each rectangle indicates a particular test/operation or a swarm of tests/-operations, while their color indicates the automation service that triggers or manages such test/operation. Red for GitLab blue for Rundeck. Instead, the green part indicates the web services front end for the user, like the OpenFPM website or web-based developer tools like GitLab development repo, Github staging repo, and the Github release repo. The pipeline automatically publishes on the OpenFPM website reports regarding performance tests, static analysis, or coverage.

In the red section, we mainly find the integration phase, which include mainly

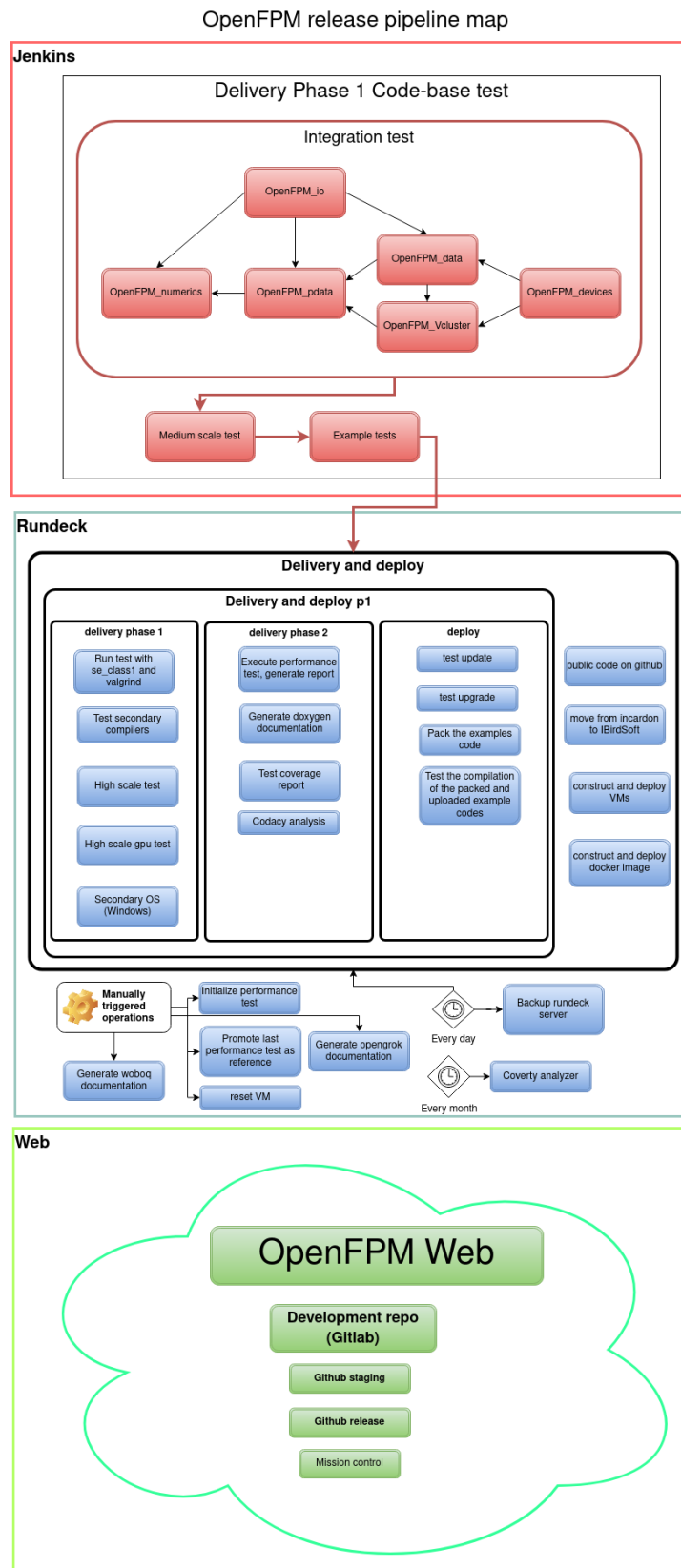


Figure 7.1: Build pipeline used to test and release OpenFPM

code-based testing. As we said, each repo has its tests for a total of over 600 tests cases (growing). All of them are grouped in the rectangle "Integration test" in figure 7.1. These tests are automated and triggered at every commits by GitLab-CI. Gitlab-CI is connected to dedicated machines to provide feedback in running the tests with newly added code as soon as possible. Overnight the rest of the pipeline is triggered, starting from the delivery phase 1 section, then the delivery phase 2, and finally the deploy phase 1 and deploy phase 2. All the jobs in each section (delivery phase 1-2, deploy phase 1-2) are triggered in parallel on the MPI-CBG cluster during the night. Each section has a barrier, and the next session is only triggered if all the tests/operations in the previous section have been completed successfully. If the entire automated pipeline is completed overnight successfully, new online documentation, examples code, docker containers, super bundles, and source code are uploaded and accessible from the OpenFPM website.

Here we will give a small description of what every test/operation does:

OpenFPM_io: Integration tests for the repository OpenFPM_io
OpenFPM_devices: Integration tests for the repository OpenFPM_devices
OpenFPM_data: Integration tests for the repository OpenFPM_data
OpenFPM_vcluster: Integration tests for the repository OpenFPM_vcluster
OpenFPM_pdata: Integration tests for the repository OpenFPM_pdata
OpenFPM_numerics: Integration tests for the repository OpenFPM_numerics

Medium scale test: Integration tests are tests that run in a short amount of time on the range of 1, 2, and 3 processes maximum. This test runs overnight and runs the tests in the range of 4 to 12 processors.

Example tests: In the repository, there is also the folder with the example codes. Example codes are distributed in order to show how to use the library. There are around 60 examples. This test compiles and install the library and check that the test compiles and run against the installed library. This step guarantee that the released examples code never goes outdated with the library.

Run test with SE_CLASS1 and sanitizers: This step runs the tests with libsanitizer and the internal assertion active. Internal assertion enables detection of buffer-overflow in all single-core data structures, and any internal abuse of functions: going out of range for parameters, using functions when not allowed.

Test secondary compilers: In this test, we check if the code is robust against additional compilers different from the one used in CI. Like for example, the Intel compiler.

High scale test: Is the same as medium-scale tests but scale up to 240 cores

High scale GPU test: In this case, we test our code with GPU activated up to 14 GPU on the MPI-CBG cluster.

Secondary OS: The code is tested against secondary operating systems. In this case, the secondary operating system is Windows with MSYS2.

Execute performance test and generate report: Run a test that checks the performance of the basic data structure and compares it with the last ref-

erence point. The suite is based on 50 graphs. In Fig 7.2 we have an example of a graph when we have a performance gain and one when we do not have a performance gain.

Generate Doxygen documentation: This is not a test but generates Doxygen documentation out of the openfpm code base, and it made it available online as documentation for the in-development version.

Test coverage report: This job generates a coverage report of the openfpm test suites. The results are analyzed with coveralls.

codacy analysis: Copy the code in repositories for static analysis. Codacy does static analysis of the code. In particular regarding copy-paste, code styles or potentially uninitialized variables.

Pack the examples: pack the example codes in a zip and upload them on the website

Test the compilation of the packed and uploaded example codes: Download the example codes packed from before and test them against the actual release candidate

public code on github: Upload the code on GitHub staging from the development repository

move from incardon to IBirdSoft Move the repository from GitHub staging, to GitHub release.

construct and deploy dockers Test the installation on several operating systems with different pack managers to check that the installation system proposes the correct packages installation for a correct installation of OpenFPM from scratch. The systems are: Fedora, OpenSuse, Debian, Ubuntu, Mac + Homebrew, Linux Mint

7.3 Tools for development: Parallel debuggers

In order to develop a parallel library, development tools like parallel debuggers are fundamental. For MPI-based parallel debugger, two predominant options exist, Allinea DDT, TotalView. Both work very well, but they are proprietary and costly if the research institute/university does not provide licenses. OpenFPM includes gdbgui, an open-source debugger, extended as part of the OpenFPM work to support an MPI debugger; therefore, its architecture extension is discussed in the next section. To the best of our knowledge, gdbgui extended to MPI is the only free opensource parallel debugger for MPI programs.

7.3.1 gdbgui architecture

Gdbgui is a simple project composed of an HTTP/Websocket server that manages multiple gdb sessions and a GUI that run on a web browser figure 7.3. In the original architecture shown in figure 7.3, there is one browser tab that connects to one gdb session through the server. The web browser tab running the gdbgui web app sends commands to control the underlying gdb session. A new gdb session starts with an HTTP request from the web browser. The server

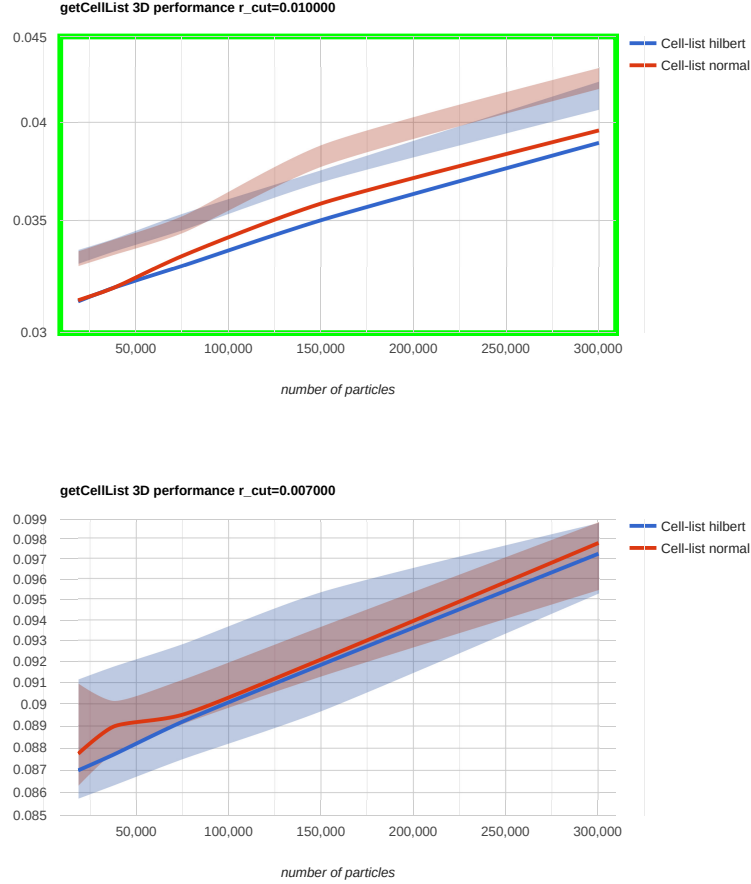


Figure 7.2: Performance graphs with different r_{cut} and for two type of cell-lists denoted by red and blue lines. Graphs are plotted with the number of particles along the x axis and time along the y axis. The lines denote the mean performance of the actual code. The regions have a width of 6 times the standard deviation of the previous performance test, with 3 times up and down from the previous measured mean. In the first case (left), the mean of the new tests is lower than 3 times sigma, signifying an improvement in performance and consequently the graph is marked by a green border. The second graph (on the right) denotes a case in the which the mean stays within the regions indicating the absence of any performance gain. If the lines were to stay above the regions, it would denote a degradation of performance and in this case, the graph is marked by a red border.

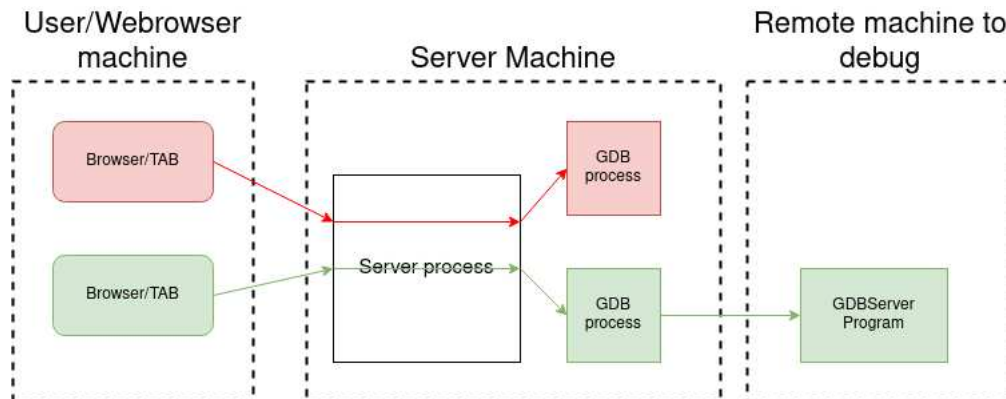


Figure 7.3: Standard architecture of gdbgui. Every browser tab is connected to a server machine. The server machine can debug either a local program spawning a local gdb process (red rectangle) and loading the program, or connect to a remote GDBServer spawning a to a remote gdb session

answers with a web page containing the gdbgui web app with the user interface code. The code on the browser triggers a secondary connection to the server via WebSocket. The user interface use this connection, to send command to the gdb session. The same connection is used to receive the answers or messages from the gdb sessions. The connection triggers a new gdb session on the server-side. Subsequent gdb commands and responses from the gdb session are sent and received through the WebSocket connection.

Gdbgui is a relatively compact code to modify and is easy to maintain. The user interface is composed of 7.7 thousand lines of Javascript with TypeScript and JSX extensions. The server is around 1.2 thousand lines of python code. Typescript is an extension of Javascript to introduce types in the javascript language, while JSX is an extension of the language to handle ReactJS components with a more compact code. ReactJS is a framework for building modular user interfaces and is the main library used to build the web-based GUI for gdbgui.

7.3.2 gdbgui extension to MPI programs architecture

Figure 7.3 shows the typical architecture of gdbgui with multiple debugging sessions where multiple users want to debug multiple programs. From this base, gdbgui has been extended to support a network of the type in figure 7.4. In the figure, there is one browser connected to a server typically running on the login node or another allocated node for the purpose, able to access the computational nodes. The server opens multiple gdb sessions that connect to the respective gdbserver on the computational nodes, and only one web browser commands the multiple sessions.

Many clusters have a batch system that allocates nodes dynamically, and it is impossible to know in advance which node to connect to. Launching the

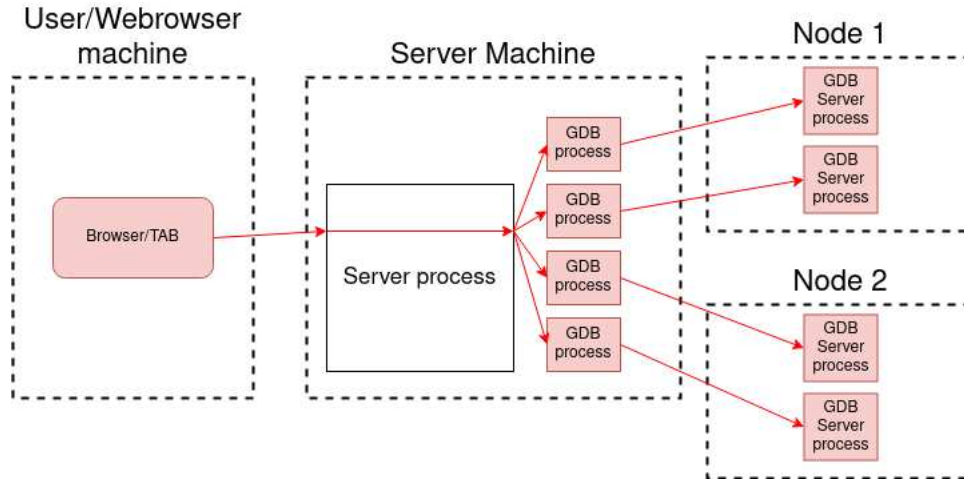


Figure 7.4: Architecture of gdbgui in case of MPI programs

MPI program to debug is done through a script that first runs an MPI program to discover all nodes in which the program is running. Once ranks and node names are collected, the information is saved in a file called `nodes_name`. After the program to debug is launched with the debugger `gdbserver`. Running the program with `gdbserver` makes the program pause at the beginning and wait for connections.

The next step is to launch the server that manages gdb sessions, open a web browser and connect to the server. Once connected, the gdbgui user interface runs on the browser. The gdbgui has been extended to have the option "Connect to mpi-gdbserver" in the menu. The input box is set to `*:60000` and does not need to be changed for most of the cases. To connect to a standard `gdbserver`, we need the address "host:port" where the host is the machine where `gdbserver` is running, and port is the port where gdb is listening. Because `gdbserver` is running on multiple nodes, "*" instruct the user interface to check for a list of nodes' name, and 60000 indicate the port in which rank zero is listening (rank one listen at 60001, rank two listen at 60002 ...). The file with node names is on the server and is requested by the gdbgui UI running on the browser. The number of lines indicates the number of sessions to open, while every line indicates how to connect each session remotely. The gdbgui UI requests the server to open as many sessions as many lines and connect each session to the same gdbgui UI. This ensures that all the messages generated by the opened session are forwarded to the same gdbgui UI.

7.3.3 gdbgui extension to MPI programs user interface

Although the web browser handles multiple sessions, its appearance remains mostly the same as figure 7.5. The only additional element is the process bar on

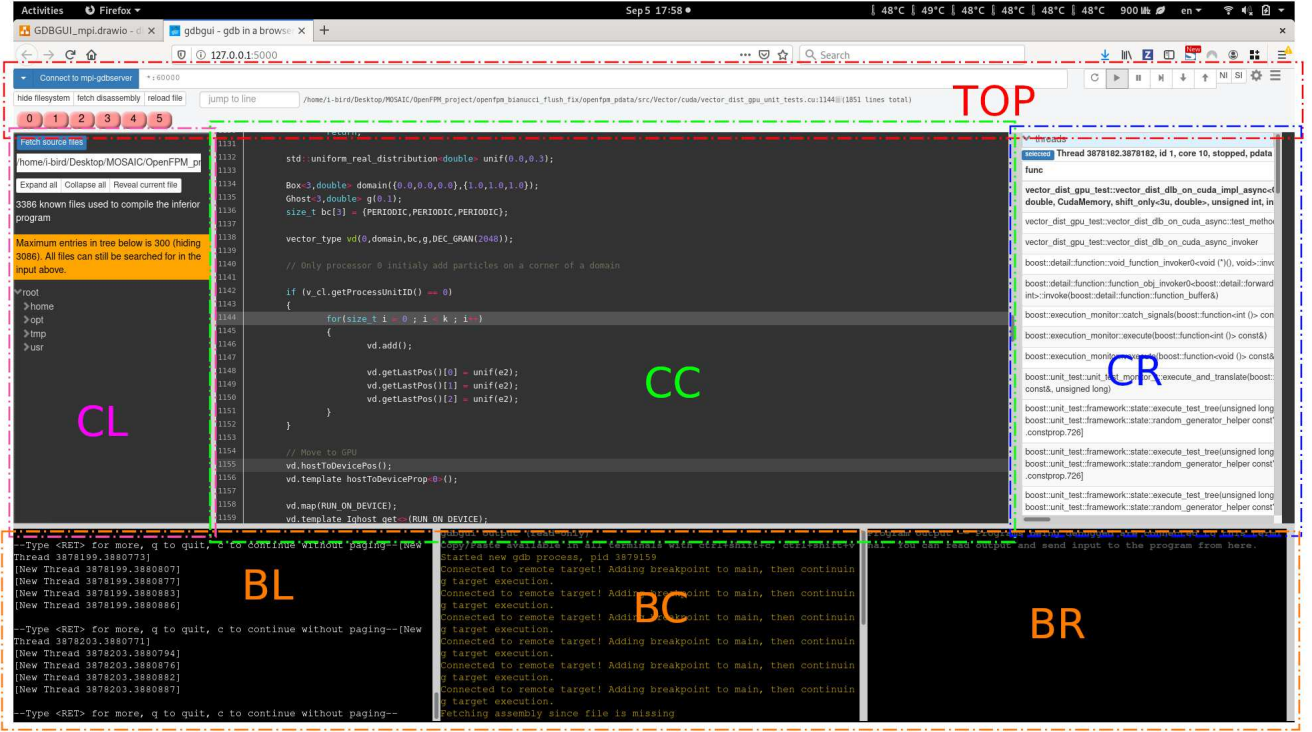


Figure 7.5: gdbgui user interface when debugging MPI programs

top containing the six red buttons from zero to five. The processor bar contains buttons to switch between processors and select the processor in focus. It works very similar to the Allinea DDT processor bar from which it takes inspiration. The color gives a visual indication of the state of the process: green if the process is running, red if the process is on pause like when it hits a breakpoint or it is in a step by step mode, and grey if the process is disconnected or terminated. Commands are issued either to all the sessions or to the process in focus.

The elements of the user interface figure 7.5 remain the same. However, in the case of an MPI program, depending on the GUI element used, commands can be sent either to all the process sessions or only one process. It remains to go across all the GUI elements of gdbgui and explain how they work in non-MPI programs and how they have been changed in the case of MPI programs. The GUI is composed of 7 sections marked in the figure 7.5. The top bar, the central/left (CL) panel, the central/central (CC), the central/right panel (CR), the bottom/left (BL), the bottom/center (BC), and the bottom right (BR). The top panel contains information about the source code opened in the source code view (CC). The typical command panel is used to resume the execution, stop the execution, step over, step into, step over. In MPI programs, the top bar is

enriched with the processor buttons bar explained before.

The CL panel remains aesthetically the same. This panel provides information about the sources used to compile the binary. Because the binary does not change across processes in an MPI program, the command to fetch sources used to create the binary is sent only to one gdb session, more precisely the process in focus.

The CC-panel is used to visualize the source code in which the processes have stopped. The line marked in white indicate where the processes have been stopped in the source code. A bright white mark is used for the process on focus and lighter white for the other processors. In the case of total divergence in the sense that processes stop in entirely different sources, the view always points to where the focus process has stopped. Switching the process will also switch the Source View to the source code in which the process we selected has stopped. Breakpoints can be set by clicking on the line number of the source code. A simple click sends a breakpoint command to all gdb sessions.

The CR-Panel contains information on the process on focus. Like local variables, the number of threads, expressions to watch registers. Switching the process on focus will also switch the process information visualized by the CR-Panel.

The BR, BC, and BL panels contain respectively the gdb session raw output collected from all gdb-sessions, the errors sent by all gdb sessions, like a gdb session that crash or get stuck, and the stdin of the program. In the case of MPI, the stdin remains unused.

Appendices

Appendix A

C++ Functional meta-Language

This section will explain the essential elements of the C++ functional meta-language or, more simply, the C++ template engine. At the end of this section, it should be clear how to write or map an imperative algorithm into a C++ functional meta-language.

The first important point is that there are no variables but only functions in this language. We will return later on, showing how it is possible to store data in them. For now, we analyze the main features of a meta-function.

Functions can take one or more inputs and return an output. In general, the input is a combination of integers or functions and returns either one integer or another function. Listing A.1 shows the definition of a function in the context of the C++ functional meta-language.

Listing A.1: meta-function definition

```
1
2 template<unsigned int integer>
3 struct value_function
4 {
5     enum
6     {
7         value = integer
8     };
9 };
10
11 template<typename arg_f1,typename arg_f2, unsigned int s>
12 struct function
13 {
14     typedef value_function<arg_f1::value + arg_f2::value + s> value;
15 };
```

The first definition is simply a function that returns the input itself as a number. The expression `value_function<5>::value` returns the value 5 as a number. The second function is more complicated than the first; it takes two functions as arguments indicated as **typename** parameters and an additional integer `s`.

Internally the function sum the values returned by the argument functions and the integer `s`. Instead of a number, this time a function is returned, which returns the result of the summation via the structure `value_function`. The expression `function<value_function<5>,value_function<4>,3>::value` is resolved by the compiler to `value_function<12>`.

Operationally, the compiler tries to resolve the type given by the expression `::value` of the type `function`. The expression at line 14 in listing A.1 requires to resolve the expression `value_function<arg_f1::value + arg_f2::value + s>`. This expression triggers the resolution of `arg_f1::value`. `arg_f1::value` in our case is `value_function<5>::value` resolving to 5, and `arg_f2::value` is `value_function<4>::value`, resolving to 4. At the end of the sequence, this leads to `value_function<5 + 4 + 3>` and finally `value_function<12>`.

As we can see, a sequence of instructions is substituted by a nested set of functions. While in an imperative language, a sequence of instructions are used to transform data progressively `b = f1(a); c = f1(b);`, in a functional language, a sequence of instructions is constructed by a nested set of functions, like `c = f1(f1(a))`.

Additionally to how instructions are formulated, imperative and functional languages provide different mechanisms for storing states or information. The primary way to store information in an imperative language is variables. In the C++ functional language, a state or information is stored in the function arguments and returned when queried afterward.

A.1 control statment "if"

Another vital element of imperative and functional languages are a control flow statements like the typical "if" condition. The partial specialization mechanism of "struct" enables constructing "if" conditions in the C++ meta-language. Suppose, for example, to create a function. expressed in standard C++ code as

```

1
2 unsigned int value_function(unsigned int integer)
3 {
4     if (integer < 5)
5     {return integer;}
6     else
7     {return integer+1;}
8 }
```

In meta-code, this function look like

```

1
2 template<unsigned int integer,bool condition = integer < 5>
3 struct value_function_meta
4 {
5     enum
6     {
7         value = integer
8     };
9 };
10
11 // Partial specialization in case cond is false
```

```

12 template<unsigned int integer>
13 struct value_function_meta<integer,false>
14 {
15     enum
16     {
17         value = integer + 1
18     };
19 };

```

The specialization mechanism tells the compiler to use the second variant if the condition parameter is false, while the first variant is chosen for all the other cases. The condition is defined as second argument of the metafunction with `condition = integer < 5`. We can use and test the metafunction with `value_function_meta<3>::value` returning the integer 3 and `value_function_meta<7>::value` returning the integer 8 at compile-time.

In the C++ meta-language, the partial specialization is equivalent to the if-else condition. The if condition without else does not exist. The constrain makes sense in functional language because `value_function_meta` must always return a value for every possible argument, hence requiring an else condition. It is possible to leave it blank in practice, but unfortunately, this will lead to a compile-time error if the compiler selects such a branch.

A.2 Arrays

We have seen how functions as objects are helpful to store states. It is possible now to extend this concept and see them as an encapsulator of a list of variables. For example, consider

```

1
2 template<typename ... Args>
3 struct list_or_tuple
4 {};

```

In this case, the struct `list_or_tuple` can be seen as a tuple or a list of types. The metafunction does not operate on the arguments, and its purpose is only to store arrays of types as arguments. Reminding that C++ templated types are functions in the meta-language, in the following, the name function or types will be used as an interchangeable term.

`list_or_tuple` store types and can be seen as an equivalent of an array in an imperative language. The concept of having a buffer in C++ Meta-language does not exist, out of having meta-function with variadic template arguments as a mechanism to store a variable length of information.

A.3 functions

This section presents functions that operate on tuples of types.

Listing A.2: function working on tuples

```

1
2 template<typename first, typename ... Others>

```

```

3 struct take_first_element
4 {
5     typedef first value;
6 }
7
8 template<typename args>
9 struct function_on_list_or_tuple
10 {};
11
12 template<typename ... Args>
13 struct function_on_list_or_tuple<list_or_tuple<Args ... >>
14 {
15     typedef take_first_element<Args ... > value;
16 };

```

The function `function_on_list_or_tuple` is a function that operates on a tuple. From listing A.2 it is possible to see how to create a function that operate specifically on `list_or_tuple`. First we have to create a non specialized function taking one argument `function_on_list_or_tuple` followed by a specialization for the case when the argument is a `list_or_tuple`. The specialization use the meta-function `take_first_element` to retrieve the first element of the list. The reason why it is necessary to specify two functions is that a meta-function with arguments must be specified for every type it receives as an argument.

A.4 Loops

The last element from an imperative language is the loops. There are two ways to create loops in the C++ meta-language. The first is through recursion, and the second is with the variadic unpack operator. The recursion method gives more flexibility than recursion, but the variadic unpack operator provides more compactness than recursion. This thesis shows the variadic unpack operator because it is enough in many cases as well as to write the algorithm 1

Given a tuple, it is possible to apply a function to each element using a code like in listing A.3.

Listing A.3: loops on tuples

```

1
2 template<unsigned int>
3 struct integer_e
4 {};
5
6 template<typename args>
7 struct function_on_list_or_tuple
8 {};
9
10 template<typename ... Args>
11 struct function_on_list_or_tuple<list_or_tuple<Args ... >>
12 {
13     typedef list_or_tuple< add_one<Args>::value ... > value;
14 };
15
16 template<typename ie>
17 struct add_one
18 {
19     typedef integer_e<ie::value + 1> value;

```

```

20 }
21
22
23 typedef list_or_tuple<integer_e<4>,
24                     integer_e<10>,
25                     integer_e<24>> integers_list;

```

The first `integer_e` is used as a function that encapsulates an integer. The `function_on_list_or_tuple` at line 7 together with its specialization at line 11 is a function that operate on a tuple. Inside this function with the line 13 `add_one<Args>::value ...` the notation “...” means to apply the meta-function `add_one` to each elements in `Args`. The result is again a variadic list encapsulated into a `list_or_tuple`. For example if we have a `list_or_tuple<integer_e<4>,integer_e<10>, integer_e<24>>` and we renamed as `integers_list` line 23 – 25 and we apply the meta-function `function_on_list_or_tuple` on `integers_list` we will obtain from `function_on_list_or_tuple<integers_list>::value` the type `list_or_tuple<integer_e<5>,integer_e<11>, integer_e<25>>`. We can also notice that `function_on_list_or_tuple` has been designed to work only on `list_or_tuple` as argument based on the specialization 10 – 14. Any attempt to make it works on a different type as argument will lead the compiler to choose the first specialization 6–8 and generate an error when we ask for the element `::value`.

At this point, we have all the elements to construct the algorithm 1.

Listing A.4: Align the types to 8 bytes

```

1  template<typename T>
2  struct if_float_or_int // No transform
3  {
4      typedef T value;
5  }
6
7  template<>
8  struct if_float_or_int<float> // transform float to double
9  {
10     typedef double value;
11 }
12
13 template<>
14 struct if_float_or_int<int> // transform int to long int
15 {
16     typedef long int value;
17 }

```

In listing A.4, the meta-code starts creating the logic to transform a type from 4 bytes to the 8 bytes counterpart using a meta-function. The code has to create an “if” condition, so it creates one partial specialization for the float case that returns double, one for int that returns a long int, and the standard case that does not do any transformation. This meta-function encapsulate the lines from 2–7 in the algorithm 1. As an example, the meta-function produces the following result when queried with `::value`

```

1  if_float_or_int<float>::value a      // equivalent to double a
2  if_float_or_int<int>::value b        // equivalent to long int b
3  if_float_or_int<std::string>::value c // equivalent to std::string c;

```


It remains to loop over all the elements of the tuple and apply the meta-function. This operation, as seen before, can be done with the template unpack operator

Listing A.5: Meta loop

```

1
2 template<typename T>
3 struct align_to_8_byte
4 {}
5
6 struct align_to_8_byte<typename ... props>
7 {
8     typedef aggregate<if_float_or_int<props>...> value;
9 }

```

In the listing A.5, `align_to_8_byte` can be seen again as a meta-function that encapsulates the compile-time algorithm and the `if_float_or_int<props>...` applies `if_float_or_int` function for every property. The output of `if_float_or_int<props>::value...` is again a variadic list. A variadic list can be finally given to an aggregate. The aggregate is our final return type of our meta-function `align_8_byte`. All together the algorithm 1 at compile-time will look like

```

1
2 template<typename T>
3 struct if_float_or_int // No transform
4 {
5     typedef T value;
6 }
7
8 template<>
9 struct if_float_or_int<float> // transform float to double
10 {
11     typedef double value;
12 }
13
14 template<>
15 struct if_float_or_int<int> // transform int to long int
16 {
17     typedef long int value;
18 }
19
20 template<typename T>
21 struct align_to_8_byte
22 {}
23
24 struct align_to_8_byte<typename ... props>
25 {
26     typedef aggregate<if_float_or_int<props>...> value;
27 }

```

To use the meta-function `align_to_8_byte` on an aggregate of types, we can use the expression `align_to_8_byte<aggregate<float,double,int,int,long int>>::value`. This will produce a transformed aggregate of type `aggregate<double,double,long int,long int,long int>`. There are other and more extended ways to express loops in metaprogramming, but the technique above is enough for our purpose. A complete explanation of template meta-programming techniques is out of the scope of this thesis.

Another essential technique used throughout the library is introspection. It is possible to create a metafunction to introspect if a particular structure exposes a method. An example in C++11 is shown in listing A.6

Listing A.6: introspecion

```

1
2
3 struct test
4 {
5     typedef double value_type;
6 }
7
8 template<typename> struct Void
9 {
10     //! define void type
11     typedef void type;
12 };
13
14 template<typename T, typename Sfinae = void>
15 struct has_value_type: value_function <false > {};
16
17
18
19 template<typename T>
20 struct has_value_type<T,
21     typename Void< typename T::value_type >::type
22     >
23     : value_function <true >
24 {};

```

The meta-code is very verbose, but its functionality is straightforward. `has_value_type<T>::value` return true only if the type `T` expose the member `value_type`. For example `has_value_type<double>::value` returns false, while `has_value_type<test>::value` returns true. Without going into the details, the meta-code in listing A.6 uses one feature of the compiler called substitution failure is not an error (SFINAE). In particular, if one template substitution fails, the compiler discards the specialization rather than produces an error. Based on this feature, we can test the validity of certain expressions with some generic type `T` at compile time. In the example above `has_value_type` check if `T::value_type` is a valid expression. If not, the specialization is discarded, and the compiler falls back to the first definition.

Finally the first definition inherits false value `value_function<false>` marked in blue in the code, while the second inherit `value_function<true>` marked again in blue in the code. This inheritance gives the possibility with `has_value_type<T>::value` to check which specialization the compiler has chosen.

Bibliography

- [1] I. F. Sbalzarini, Abstractions and middleware for petascale computing and beyond., *Intl. J. Distr. Systems & Technol.* 1(2) (2010) 40–56.
- [2] OpenACC web page [cited 2018].
URL <https://www.openacc.org/>
- [3] OpenMP Architecture Review Board, OpenMP application program interface version 3.0 (May 2008).
URL <http://www.openmp.org/mp-documents/spec30.pdf>
- [4] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with cuda, *Queue* 6 (2) (2008) 40–53. doi:10.1145/1365490.1365500.
URL <http://doi.acm.org/10.1145/1365490.1365500>
- [5] J. Stone, D. Gohara, G. Shi, Opencl: A parallel programming standard for heterogeneous computing systems, *Computing in science & engineering* 12 (3) (2010) 66.
- [6] C. H. Koelbel, *The High Performance FORTRAN Handbook*, MIT Press, 1993.
- [7] N. Carriero, D. Gelernter, Linda in context, *Commun. ACM* 32 (4) (1989) 444–458.
- [8] A. A. Wray, A manual of the vectoral language, Internal report, NASA Ames Research Center, Moffett Field, California (1988).
- [9] J. Bezanson, S. Karpinski, V. B. Shah, A. Edelman, Julia: A fast dynamic language for technical computing, Tech. rep., arXiv:1209.5145 (09 2012).
arXiv:1209.5145.
- [10] N. D. Matsakis, F. S. Klock II, The rust language, in: *ACM SIGAda Ada Letters*, Vol. 34, ACM, 2014, pp. 103–104.
- [11] D. Padua (Ed.), *Message Passing Interface (MPI)*, Springer US, Boston, MA, 2011, pp. 1116–1116. doi:10.1007/978-0-387-09766-4_2085.
URL https://doi.org/10.1007/978-0-387-09766-4_2085

- [12] E. Gabriel, E. F. Graham, G. Bosilca, A. Thara, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proceedings, 11th European PVM/MPI Users' Group Meeting, 2004, pp. 97–104.
- [13] W. Gropp, Mpich2: A new start for mpi implementations, in: Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, London, UK, UK, 2002, p. 7.
- [14] H. Kaiser, M. Brodowicz, T. Sterling, Parallelex an advanced parallel execution model for scaling-impaired applications, in: Proceedings of the 2009 International Conference on Parallel Processing Workshops, ICPPW '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 394–401. doi:10.1109/ICPPW.2009.14.
URL <http://dx.doi.org/10.1109/ICPPW.2009.14>
- [15] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23 (2011) 187–198. doi:10.1002/cpe.1631.
URL <http://hal.inria.fr/inria-00550877>
- [16] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing locality and independence with logical regions, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Washington, DC, USA, 2012.
- [17] R. W. Numrich, J. Reid, Co-array fortran for parallel programming, SIGPLAN Fortran Forum 17 (2) (1998) 1–31. doi:10.1145/289918.289920.
- [18] T. El-Ghazawi, L. Smith, Upc: Unified parallel c, in: Proc. 2006 ACM/IEEE Conference on Supercomputing, SC '06, ACM, New York, NY, USA, 2006, p. 27. doi:10.1145/1188455.1188483.
URL <http://doi.acm.org/10.1145/1188455.1188483>
- [19] D. Padua (Ed.), SHMEM, Springer US, Boston, MA, 2011, pp. 1812–1812. doi:10.1007/978-0-387-09766-4_2050.
URL https://doi.org/10.1007/978-0-387-09766-4_2050
- [20] K. F rlinger, T. Fuchs, R. Kowalewski, DASH: A C++ PGAS library for distributed data structures and parallel algorithms, in: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016, pp. 983–990. arXiv:1610.01482.

- [21] S. Potluri, D. Rossetti, D. Becker, D. Poole, M. Gorentla Venkata, O. Hernandez, P. Shamis, M. G. Lopez, M. Baker, W. Poole, Exploring open-shmem model to program gpu-based extreme-scale systems, in: Revised Selected Papers of the Second Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies - Volume 9397, OpenSHMEM 2015, Springer-Verlag, Berlin, Heidelberg, 2015, p. 18–35. doi:10.1007/978-3-319-26428-8_2. URL https://doi.org/10.1007/978-3-319-26428-8_2
- [22] L. V. Kale, S. Krishnan, CHARM++: A portable concurrent object oriented system based on C++, in: Proc. 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, 1993, pp. 91–108.
- [23] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comput. Phys. 117 (1995) 1–19.
- [24] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation, J. Chem. Theor. Comput. Web-published version; volume and pages still not available; DOI: 10.1021/ct700301q.
- [25] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, K. Schulten, Scalable molecular dynamics with NAMD, J. Comput. Chem. 26 (2005) 1781–1802.
- [26] J. A. Anderson, J. Glaser, S. C. Glotzer, Hoomd-blue: A python package for high-performance molecular dynamics and hard particle monte carlo simulations, Computational Materials Science 173 (2020) 109363. doi:<https://doi.org/10.1016/j.commatsci.2019.109363>. URL <https://www.sciencedirect.com/science/article/pii/S0927025619306627>
- [27] F. Weik, R. Weeber, K. Szuttor, K. Breitsprecher, J. de Graaf, M. Kuron, J. Landsgesell, H. Menke, D. Sean, C. Holm, Espresso 4.0 – an extensible software package for simulating soft matter systems, The European Physical Journal Special Topics 227 (14) (2019) 1789–1816. doi:10.1140/epjst/e2019-800186-9. URL <https://doi.org/10.1140/epjst/e2019-800186-9>
- [28] A. J. C. Crespo, J. M. Domínguez, B. D. Rogers, M. Gómez-Gesteira, S. Longshaw, R. Canelas, R. Vacondio, A. Barreiro, O. García-Feal, Dual-physics: Open-source parallel cfd solver based on smoothed particle hydrodynamics (sph), Computer Physics Communications 187 (2015) 204–216.
- [29] A. Hérault, G. Bilotta, R. Dalrymple, Sph on gpu with cuda, Journal of Hydraulic Research 2010 (2010) 74–79. doi:10.1080/00221686.2010.9641247.

- [30] T. Trilinos Project Team, The Trilinos Project Website.
- [31] A. Dedner, R. Klöfkorn, M. Nolte, M. Ohlberger, A generic interface for parallel and adaptive scientific computing: Abstraction principles and the dune-fem module, *Computing* 90 (2010) 165–196. doi:10.1007/s00607-010-0110-3.
- [32] D. Arndt, W. Bangerth, B. Blais, M. Fehling, R. Gassmöller, T. Heister, L. Heltai, U. Köcher, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, S. Proell, K. Simon, B. Turcksin, D. Wells, J. Zhang, The `deal.II` library, version 9.3, *Journal of Numerical Mathematics*.
URL <https://dealii.org/deal93-preprint.pdf>
- [33] Rocky-dem.
URL <https://rocky.esss.co/software/>
- [34] A. Wissink, R. Hornung, S. Kohn, S. Smith, N. Elliott, Large scale parallel structured AMR calculations using the SAMRAI framework, in: *Proc. SC01 Conf. High Perf. Network. & Comput.* Denver CO, 2001.
- [35] Amrex web page [cited 2018].
URL <https://amrex-codes.github.io/amrex/>
- [36] Openfoam.
URL <https://www.openfoam.com/>
- [37] A. Schäfer, D. Fey, Libgeodecomp: A grid-enabled library for geometric decomposition codes, in: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, Springer, 2008, pp. 285–294.
- [38] E. Gallopoulos, E. Houstis, J. R. Rice, Computer as thinker/doer: problem-solving environments for computational science, *IEEE Computational Science and Engineering*.
- [39] Modelica web pge [cited 2018].
URL <https://www.modelica.org/>
- [40] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, D. Padua, Falcon: A matlab interactive restructuring compiler, in: C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 269–288.
- [41] Ansys.
URL <https://www.ansys.com>
- [42] Comsol.
URL <https://www.comsol.com/>

- [43] J. Reynders, J. Cummings, M. Tholburn, P. Hinker, S. Atlas, S. Banerjee, M. Srikant, W. Humphrey, S. Karmesin, K. Keahey, Pooma: a framework for scientific simulation on parallel architectures, in: A. Bode, M. Gerndt, R. Hackenberg, H. Hellwagner (Eds.), *Proceedings. First International Workshop on High-Level Programming Models and Supportive Environments*, Tech. Univ. Munchen; Res. Centre Julich; Central Inst. Appl. Math.; 10th IEEE Int. Parallel Process. Symposium; IEEE Comput. Soc. Tech. Committee on Parallel Process.; ACM SIGARCH, IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 1996, pp. 41–49.
- [44] I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, P. Koumoutsakos, PPM – a highly efficient parallel particle-mesh library for the simulation of continuum systems, *J. Comput. Phys.* 215 (2) (2006) 566–588.
- [45] O. Awile, O. Demirel, I. F. Sbalzarini, Toward an object-oriented core of the PPM library, in: *Proc. ICNAAM, Numerical Analysis and Applied Mathematics, International Conference, AIP, 2010*, pp. 1313–1316.
- [46] O. Awile, M. Mitrović, S. Reboux, I. F. Sbalzarini, A domain-specific programming language for particle simulations on distributed-memory parallel computers, in: *Proc. III Intl. Conf. Particle-based Methods (PARTICLES)*, Stuttgart, Germany, 2013, p. p52.
- [47] S. Karol, T. Nett, P. Incardona, N. Khouzami, J. Castrillon, I. F. Sbalzarini, A language and development environment for parallel particle methods, in: *International Conference on Particle-based Methods – Fundamentals and Applications*, Hanover, Germany, 2017, pp. 1–12.
- [48] L. Verlet, Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules, *Phys. Rev.* 159 (1) (1967) 98–103.
- [49] R. W. Hockney, J. W. Eastwood, *Computer Simulation using Particles*, Institute of Physics Publishing, 1988.
- [50] G.-H. Cottet, P. Koumoutsakos, *Vortex Methods – Theory and Practice*, Cambridge University Press, New York, 2000.
- [51] J. Barnes, P. Hut, A hierarchical $O(N \log N)$ force-calculation algorithm, *Nature* 324 (1986) 446–449.
- [52] V. Rokhlin, Rapid solution of integral equations of classical potential theory, *Journal of Computational Physics* 60 (2) (1985) 187–207.
doi:[https://doi.org/10.1016/0021-9991\(85\)90002-6](https://doi.org/10.1016/0021-9991(85)90002-6).
URL <https://www.sciencedirect.com/science/article/pii/0021999185900026>
- [53] P. P. Ewald, Die berechnung optischer und elektrostatischer gitterpotentiale, *Annalen der Physik*.

- [54] G.-H. Cottet, J.-M. Etancelin, F. P  rignon, C. Picard, High order semi-Lagrangian particle methods for transport equations: numerical analysis and implementation issues, *ESAIM: Mathematical Modelling and Numerical Analysis* 48 (4) (2014) 1029–1060.
- [55] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Web page, <http://www.mcs.anl.gov/petsc> (2016).
URL <http://www.mcs.anl.gov/petsc>
- [56] G. Guennebaud, B. Jacob, et al., Eigen v3, <http://eigen.tuxfamily.org> (2010).
- [57] A. Huebl, R. Lehe, J.-L. Vay, D. Grote, I. Sbalzarini, S. Kuschel, M. Bussmann, openpmd 1.0.0: A meta data standard for particle and mesh based data (11 2015). doi:10.5281/zenodo.33624.
- [58] O. Green, R. McColl, D. A. Bader, Gpu merge path: A gpu merging algorithm, in: *Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12*, Association for Computing Machinery, New York, NY, USA, 2012, p. 331–340. doi:10.1145/2304576.2304621.
URL <https://doi.org/10.1145/2304576.2304621>
- [59] N. Research, Cub provides state-of-the-art, reusable software components for every layer of the cuda programming model: Parallel primitives, utilities.
- [60] S. Baxter, moderngpu is a productivity library for general-purpose computing on gpus (2016).
- [61] K. Museth, J. Lait, J. Johanson, J. Budsberg, R. Henderson, M. Ald  n, P. Cucka, D. R. Hill, A. Pearce, Openvdb: an open-source data structure and toolkit for high-resolution volumes, in: *SIGGRAPH '13*, 2013.
- [62] G. Karypis, V. Kumar, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering, *J. Parallel Distrib. Comput.* 48 (1) (1998) 71–95. doi:10.1006/jpdc.1997.1403.
URL <http://dx.doi.org/10.1006/jpdc.1997.1403>
- [63] M. Bergdorf, I. F. Sbalzarini, P. Koumoutsakos, A lagrangian particle method for reaction–diffusion systems on deforming surfaces, *J. Math Biol.* 61 (5) (2009) 649–663. doi:10.1007/s00285-009-0315-2.
URL <http://www.cse-lab.ethz.ch/wp-content/papercite-data/pdf/bergdorf2009a.pdf>
- [64] D. Peng, B. Merriman, S. Osher, H. Zhao, M. Kang, A PDE-based fast local level set method, *J. Comput. Phys.* 155 (1999) 410–438.

- [65] Hdf5 web page [cited 2018].
URL <http://www.hdfgroup.org/HDF5/>
- [66] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, S. Klasky, Adios 2: The adaptable input output system. a framework for high-performance data management, *SoftwareX* 12 (2020) 100561. doi:<https://doi.org/10.1016/j.softx.2020.100561>.
URL <https://www.sciencedirect.com/science/article/pii/S2352711019302560>
- [67] L. B. Schroeder Will, Martin Ken, The Visualization Toolkit (4th ed.), Kitware, 2006.
- [68] A. Utkarsh, The ParaView Guide: A Parallel Visualization Application, ACM Press, 2015.
- [69] X. Hu, N. Adams, A multi-phase sph method for macroscopic and mesoscopic flows, *Journal of Computational Physics* 213 (2) (2006) 844–861. doi:<https://doi.org/10.1016/j.jcp.2005.09.001>.
URL <http://www.sciencedirect.com/science/article/pii/S0021999105004195>
- [70] S. Adami, X. Hu, N. Adams, A generalized wall boundary condition for smoothed particle hydrodynamics, *Journal of Computational Physics* 231 (21) (2012) 7057–7075. doi:<https://doi.org/10.1016/j.jcp.2012.05.005>.
URL <http://www.sciencedirect.com/science/article/pii/S002199911200229X>
- [71] J. J. Monaghan, Smoothed particle hydrodynamics, *Annu. Rev. Astron. Astrophys.* 30 (1992) 543–574.
- [72] P. Gray, S. K. Scott, Autocatalytic reactions in the isothermal, continuous stirred tank reactor - isolas and other forms multistability, *Chem. Eng. Sci.* 38 (1) (1983) 29–43.
- [73] P. Gray, S. K. Scott, Autocatalytic reactions in the isothermal, continuous stirred tank reactor : Oscillations and instabilities in the system $A + 2B \rightarrow 3B$; $B \rightarrow C$, *Chem. Eng. Sci.* 39 (6) (1984) 1087–1097.
- [74] P. Gray, S. K. Scott, Sustained oscillations and other exotic patterns of behavior in isothermal reactions, *J. Phys. Chem.* 89 (1) (1985) 22–32.
- [75] J. Lee, A. Ishihara, J. A. Theriot, K. Jacobson, Principles of locomotion for simple-shaped cells, *Nature* 362 (1993) 167–171.

- [76] J. E. Pearson, Complex patterns in a simple system, *Science* 261 (5118) (1993) 189–192.
- [77] M. Bergdorf, P. Koumoutsakos, A. Leonard, Direct numerical simulations of vortex rings at $Re_\Gamma = 7500$, *Journal of Fluid Mechanics* 581 (2007) 495–505.
URL <http://www.cse-lab.ethz.ch/wp-content/papercite-data/pdf/bergdorf2007a.pdf>
- [78] J. H. Walther, I. F. Sbalzarini, Large-scale parallel discrete element simulations of granular flow, *Engineering Computations* 26 (6) (2009) 688–697.
- [79] L. E. Silbert, D. Ertas, G. S. Grest, T. C. Halsey, D. Levine, S. J. Plimpton, Granular flow down an inclined plane: Bagnold scaling and rheology, *Phys. Rev. E* 64 (2001) 051302.
- [80] K. G. Wilson, Confinement of quarks, *Phys. Rev. D* 10 (1974) 2445–2459.
doi:10.1103/PhysRevD.10.2445.
URL <https://link.aps.org/doi/10.1103/PhysRevD.10.2445>
- [81] C. Bonati, G. Cossu, M. D’Elia, P. Incardona, QCD simulations with staggered fermions on GPUs, *Computer Physics Communications* 183 (4) (2012) 853–863. doi:<https://doi.org/10.1016/j.cpc.2011.12.011>.
URL <http://www.sciencedirect.com/science/article/pii/S0010465511003997>
- [82] Y. Afshar, I. F. Sbalzarini, A parallel distributed-memory particle method enables acquisition-rate segmentation of large fluorescence microscopy images, *PLoS One* 11 (4) (2016) e0152528. doi:10.1371/journal.pone.0152528.
- [83] C. L. Müller, I. F. Sbalzarini, Gaussian Adaptation as a unifying framework for continuous black-box optimization and adaptive Monte Carlo sampling, in: *Proc. IEEE Congress on Evolutionary Computation (CEC)*, Barcelona, Spain, 2010, pp. 2594–2601.
- [84] N. Hansen, S. D. Muller, P. Koumoutsakos, Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES), *Evol. Comput.* 11 (1) (2003) 1–18.
- [85] N. Hansen, The CMA Evolution Strategy: A Tutorial (Apr 2007).
URL <http://www.inf.ethz.ch/personal/hansenn/cmatutorial.pdf>
- [86] C. L. Müller, B. Baumgartner, I. F. Sbalzarini, Particle swarm CMA evolution strategy for the optimization of multi-funnel landscapes, in: *Proc. IEEE Congress on Evolutionary Computation (CEC)*, IEEE, Trondheim, Norway, 2009, pp. 2685–2692.

- [87] C. L. Müller, B. Baumgartner, G. Ofenbeck, B. Schrader, I. F. Sbalzarini, pCMALib: a parallel FORTRAN 90 library for the evolution strategy with covariance matrix adaptation, in: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09, ACM, New York, NY, USA, 2009, pp. 1411–1418.
- [88] C. L. Müller, I. F. Sbalzarini, Global characterization of the CEC 2005 fitness landscapes using fitness-distance analysis, in: Proc. EvoStar, Vol. 6624 of Lecture Notes in Computer Science, Springer, Torino, Italy, 2011, pp. 294–303.