UNIVERSITA' DEGLI STUDI DI PARMA

DOTTORATO DI RICERCA IN

INGEGNERIA CIVILE E ARCHITETTURA

Ciclo XXXII

A general design for a scalable MPI-GPU Shallow Water Equations solver on a multi-resolution grid

Coordinatore:
Chiar.mo Prof. Andrea Carpinteri

Tutor:
Dott. Ing. Renato Vacondio

Co-Tutor:
Chiar.mo Prof. Alessandro Dal Palù

Dottorando:
Massimiliano Turchetto

Anni 2016/2019

1. Reviewer:

2. Reviewer:

Day of the defense:

Signature from head of PhD committee:

Abstract

This thesis presents a multi-GPU implementation of a Finite-Volume solver approximating the $2D$ Shallow Water Equations (SWE) in order to simulate flooding events. There is a consensus in the scientific literature that such events are increasing year by year and the only feasible way to deal with them is to increase the resilience of the territory by making fast and accurate predictions on their evolution. A previous version of the solver, which is the starting point of this thesis, takes a first step in that direction by solving conservation laws across a multi-resolution grid and offloading the whole computation to a *Graphic Processing Unit* (GPU). Notwithstanding the many optimizations adopted inside the kernels, namely the efficient memory representation and the exclusion of the dry cells from the computational time-steps, the single GPU implementation showed evident scalability limitations on the size of the input grids and the simulation times which may take several hours in case of realistic scenarios. The goal of this thesis is to overcome such limitations by distributing the workload across multiple GPUs, each one managed by a single process. The overall design of the code follows the common approach of the High Performance Computing (HPC) applied to the field of Computational Fluid Dynamics in which the input grid is partitioned in different parts communicating their borders using the *Message Passing Interface* (MPI) standard and

performing a global time-step reduction. Two different partitioning algorithms have been considered: the first making mono-dimensional domain subdivisions and the second, more sophisticated, based on Hilbert Space Filling Curves (HSFC). While both proved efficient in the Weak Scalability Test, showing a constant efficiency of $\sim$ 90% between 8 and 64 GPUs, the HSFC partitioning outperformed the $1D$ one in the Strong Scalability Test by reaching an efficiency of 85% on 64 GPUs. The MPI communications have been overlapped with the kernel computations, gaining an efficiency of 10% in the strong scaling up to 32 GPUs. The ever changing nature of the wet and dry fronts in realistic scenarios creates imbalances in the computational load across different MPI ranks. In particular, partitions having less wet cells spend idle times waiting for other processes to terminate their computations in order to perform the global time-step reduction. An heuristic has been designed and implemented to minimize such idle times by migrating grid portions from computationally lighter processes (having high idle times) to heavier ones. The tests showed that the heuristic works well and it is able decrease the time-step duration by gradually lowering the idle times to the order of microseconds.

To my family, Sebastian, Anna and Laura.
Thank you for always being there for me.

Acknowledgements

I would like to thank my tutors, Renato Vacondio and Alessandro dal Palù, for their helpful advice on the work carried out during my PhD. A special thanks to Alessia Ferrari for her precious help in the input models generation. Finally, I thank the Hydraulic Constructions research group for the positive environment they created. The past three years have been a wonderful experience.

Contents

List of Figures

1

Introduction

The climate changes are considered the driving factor for many natural calamities. According to the IPCC reports [1] [2] the climate change is responsible for environmental damages with direct impact on humans and more in general all forms of life. Climate change has many negative effects on the food security, terrestrial ecosystems and contributed to desertification and land degradation of many regions. In the recent years events like droughts, precipitations and floods, increased their frequency, intensity and duration. The human health has been negatively affected by weather phenomena such as extreme heath, hurricanes and storms. There are also multiple climate-related health threats like toxic algas blooms, waterborne diseases due to the increasing water temperatures, dust storms due to land use and land cover changes (i.e., loss of natural areas such as forests) together with increasingly high emissions of CO_2. Even the food security is threatened by the global warming thanks to the different precipitation patterns, frequency and intensity of natural calamities.

Furthermore, we assisted to a shift of the climate zones thanks to global warming with negative consequences on animal species and plants which decreased in abundance and ranges. A lower animal growth rate and productivity have been

observed in pastoral systems together with an increment of pests, diseases and infestations. The global warming together with the urbanization can increase the so called *heat island* effects on urban areas especially during heat related events such as heat waves. These phenomena lead to the intensification of extreme rainfalls which in turn are strictly related to the flooding events.

Flooding events are catastrophic phenomena threatening life and causing huge economics damages to humans and infrastructures around the world. A great amount of studies are relating the global warming with the increase in flooding phenomena causing relevant damages to people and infrastructures [3] [4] [5] [6] [7] In particular, according to [8] we will assist to an increase of extreme events in 21 states in Europe by 2035 with a direct implication on the flooding hazards. These unsettling predictions can be extended to a global scale [9] in which, assuming to limit the global warming to 1.5° compared to the preindustrial levels as ambitiously (and unrealistically [4]) settled in the Paris Agreement [10], the increase of population affected by floods will be of 100-170% compared to the baseline period 1976-2005 with a 120-170% increase in flood-related damages.

The only way we have to deal with such phenomena (once they happen) is to dynamically adapt to them by adopting countermeasures on the fly, avoiding or minimizing the damages to people and infrastructures. In other words, what we can do is to increase the *resilience* of the territory by making fast and accurate predictions on the evolution of the phenomena. According to [11], trying to avoid flooding events by just increasing the infrastructural protections helps to deal with small events but exposes a large amount of people to catastrophic events. With the ability of describing the evolution of an extreme phenomena in real-time (or quasi-realtime) comes the great advantage of limiting the damages to people especially in life-threatening events, for example by making fast evacuation plans. However we can do much more than limiting our actions to simply respond to an extreme

event once it is in progress. We can model realistic case scenarios and simulate them under a number of different conditions.

The term *simulation* will be hereinafter adopted to talk about the process involving the sequence of steps carried out to describe the evolution in time of a physical phenomena. Such phenomena can be already happened in the reality, currently in progress or simply the product of an hypothetical scenario. The first case is particularly useful to assess the accuracy of the simulation by comparing the results with the reality. The second one is useful to make fast predictions as highlighted earlier. The third one is particularly interesting because allows to imagine an event of particular likelihood and observe its effects, developing a deep understanding of the causes and the dynamics of a flood phenomena, feeding important data which can be used to plan smart interventions on the protection and mitigation infrastructures.

Nowadays simulations are carried out by computer programs usually referred to as *solvers*, generally consisting in tens of thousands code lines (in some cases much more) implementing algorithms to simulate any sort of physical event of interest. However, the software implementation is "only" the last step in a sequence of approximations consisting in (i) describing the reality through a system of equations, (ii) choosing a numerical method to approximate the solution of such system and (iii) finally implement the chosen method which is subject to a finite machine precision. The work carried out in this PhD thesis focuses on the third point and consists in a contribution to the implementation of a numerical solver which is able to accurately describe a flooding event.

Events like river inundations, where the flood waves can propagate for thousands of square kilometers can be accurately described through the 2D *Shallow Water Equations* (SWE) [12] [13] [14], consisting in a system of partial differential equations of hyperbolic type deriving from the laws of momentum and mass conservation. As outlined in [15] the SWE represent a fully 2D model created to

overcome the limitations present in other proposed models. In particular, fully 1D models have been discarded due to their inability to model 2D flows outside the river region, for example in case of a river breach; 3D models are computationally too expensive to be adopted for simulations at regional level, where the domain can extend for thousands of square kilometers. Other approaches proposed to use 1D and 2D models together in a decoupled or coupled way. The former has some numerical limitations meaning that it cannot be used to model some important case scenarios. The latter needs some assumptions on the input models to work properly [16], making it less general if compared to the 2D SWE.

In the last decade multiple studies [17] [18] [19] [20] [21] [16] have shown the versatility, accuracy and stability of numerical solvers implementing SWE models and the only questions remaining unanswered are how far these applications can be pushed in terms of performance and scalability and if it is possible to make real-time simulations of a large scale flooding event a real possibility. The answer to the last question is almost certainly affirmative and a lot of scientists around the world are working hard to achieve this goal not only in the field of fluid dynamics but more in general in a large variety of science disciplines. The work carried out in this thesis goes in that direction and it is primarily focused on the parallelization of a Shallow Water Equation solver named PARFLOOD. The main features of such solver are the offloading of the computational part of the simulation to a Graphic Processing Unit and the exploitation of a multi-resolution grid to optimize the memory utilization. It is the goal of this chapter to firstly introduce the numerical (Section 1.1) and computational aspects (Section 1.2) of the SWE model. The chapter concludes by motivating and formalizing the goals of this thesis (Section 1.4).

1.1 Numerical modeling

The differential form of the SWE is defined by the equation 1.1

$$\mathbf{U}_t + \mathbf{F}_x + \mathbf{G}_y = \mathbf{S}(\mathbf{U}) \tag{1.1}$$

where $\mathbf{U}$ is the vector of conserved variables, $\mathbf{F}$ and $\mathbf{G}$ are the physical fluxes in the x and y directions respectively and $\mathbf{S}$ is the source term. The conserved variables, fluxes and the source terms are vectors defined as follows

$$\mathbf{U} = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, \quad \mathbf{F}_x = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}, \quad \mathbf{G}_y = \begin{bmatrix} hv \\ hvu \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}, \quad \mathbf{S}(\mathbf{U}) = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

where h represents the water depth, u and v are the velocities in the x and y directions respectively, g is the acceleration due to gravity and s_1, s_2, s_3 are functions of the conserved variables. The differential form works only assuming smooth solutions meaning that it is not general enough to capture discontinuities such as shock waves. For this reason the Equation (1.1) must be rewritten in integral form. For simplicity of explanation the source term is omitted, thus considering $\mathbf{S}(\mathbf{U}) = 0$. The integral form then becomes

$$\frac{d}{dt}\int_V \mathbf{U}\, dV + \int_\Omega \mathbf{H} \cdot \mathbf{n}\, d\Omega = 0 \tag{1.2}$$

where V represents a control volume, Ω is its boundary, $\mathbf{H} = (\mathbf{F}, \mathbf{H})$ represents the tensor of fluxes in the x and y directions respectively and $\mathbf{n} = [n_1, n_2]$ is the vector normal to Ω. Despite the fact that the *exact solution* of the Equation (1.2) is unique, it is not always possible to find it analytically. For this reason *numeric methods* are adopted to approximate the solution to a system of equations.

PARFLOOD uses a numerical method called *finite volume method* which consists in defining a discretization of the spatial domain in a *set of volumes* having a finite surface. Such set is called *mesh* or *grid*. Since the Equation (1.2) describes

a phenomenon evolving in time, the solution is approximated at each volume at discrete time intervals. Such intervals are often referred to as *time-steps*. The shape and disposition of volumes in the domain defines the type of mesh adopted. In general, if it is possible to map such mesh in a Cartesian space maintaining the neighboring relations between each cell, the mesh is called *structured*. Vice versa if such mapping does not exist, the mesh is called *unstructured*.

When a mesh contains both structured and unstructured sections, it is called *hybrid*. The type of grid used by PARFLOOD falls under the hybrid class and it will be detailed in Section 1.2.1. Such grid is made of square volumes or *cells* which are mapped on the $2D$ Cartesian space even though, in general, they do not preserve the neighboring relations. To distinguish between the original arrangement of cells in space and the one obtained with the mapping, the former will be called *physical space* and the latter *logical space*. Since the logical space is a Cartesian space, each cell can be referred to by using an index (i, j). This is important because the formulation of the FV method that follows, approximates the solution of Equation (1.2) at each cell (i, j) in the physical space assuming a Cartesian domain discretization. The problems arising by adapting this method to a multi-resolution grid (having a Cartesian representation in a logical space) stand at the basis for understanding the work carried out in this thesis.

The term V in Equation (1.2) indicating a generic cell, can be replaced by V_{ij}, referring to the cell having coordinates (i, j) in the logical space. The vector of conserved variables $\mathbf{U}_{ij}$ is averaged for each cell by the vector $\hat{\mathbf{U}}_{ij}$ defined as

$$\hat{\mathbf{U}}_{ij} = \frac{1}{|V_{ij}|} \int_{V_{ij}} \mathbf{U} \, dV_{ij} \tag{1.3}$$

By substituting (1.3) in (1.2) we obtain the equation

$$\frac{d}{dt}\hat{\mathbf{U}}_{ij} = -\frac{1}{|V_{ij}|} \int_{\Omega} \mathbf{H} \cdot \mathbf{n} \, d\Omega \tag{1.4}$$

Recalling that the physical domain is discretized by means of square cells (see the Figure 1.1), we can introduce a variable v_s for $s = 1, \ldots, 4$ used to number the four vertices of each cell. In this way we can rewrite the line integral on the left hand side of the Equation (1.4) in the following way

$$\int_{\Omega} \mathbf{H} \cdot \mathbf{n} \, d\Omega = \sum_{s=1}^{4} \int_{v_s}^{v_{s+1}} \mathbf{H} \cdot \mathbf{n}_s \, dv \tag{1.5}$$

where, by using a simple abuse of notation, $s+1$ is assumed to be equal to 1 when $s = 4$. PARFLOOD assumes that the physical representation of the input grid is made of cells which edges are aligned with the Cartesian axes. For this reason, the vector $\mathbf{n}_s$, defined as the unit vector normal to the surface of the edge $[s, s+1]$, assumes the four values $[0, -1], [1, 0], [0, 1]$ and $[0, -1]$, corresponding to the edges $[1, 2], [2, 3], [3, 4]$ and $[4, 1]$ respectively, as depicted in the Figure 1.1. The four resulting integrals in the right hand side of the Equation (1.5) becomes

$$\int_{v_1}^{v_2} (\mathbf{F}, \mathbf{G}) \cdot [0, -1] \, dv = -\Delta_x \mathbf{G}_{ij-\frac{1}{2}} \qquad \int_{v_2}^{v_3} (\mathbf{F}, \mathbf{G}) \cdot [1, 0] \, dv = \Delta_y \mathbf{F}_{i+\frac{1}{2}j}$$

$$\int_{v_3}^{v_4} (\mathbf{F}, \mathbf{G}) \cdot [0, 1] \, dv = \Delta_x \mathbf{G}_{ij+\frac{1}{2}} \qquad \int_{v_4}^{v_1} (\mathbf{F}, \mathbf{G}) \cdot [-1, 0] \, dv = -\Delta_y \mathbf{F}_{i-\frac{1}{2}j}$$

where the resulting $\mathbf{F}_{i\pm\frac{1}{2},j}$ and $\mathbf{G}_{i,j\pm\frac{1}{2}}$ are called *intercell numerical fluxes* or simply *numerical fluxes*, and need to be approximated. The values Δ_x and Δ_y represent the length of the cell edges in the x and y directions respectively. The product $\Delta_x \Delta_y$ is also called *resolution*.

Finally the Equation (1.4) becomes

$$\begin{aligned}\frac{d}{dt}\hat{\mathbf{U}}_{ij} &= -\frac{1}{\Delta_x\Delta_y}\sum_{s=1}^{4}\int_{v_s}^{v_{s+1}}\mathbf{H}\cdot\mathbf{n}_s\,dv \\ &= -\left[\frac{\Delta_y\left(\mathbf{F}_{i+\frac{1}{2},j}-\mathbf{F}_{i-\frac{1}{2},j}\right)}{\Delta_x\Delta_y}+\frac{\Delta_x\left(\mathbf{F}_{i,j+\frac{1}{2}}-\mathbf{F}_{i,j-\frac{1}{2}}\right)}{\Delta_x\Delta_y}\right] \\ &= -\left[\frac{\mathbf{F}_{i+\frac{1}{2},j}-\mathbf{F}_{i-\frac{1}{2},j}}{\Delta x}+\frac{\mathbf{F}_{i,j+\frac{1}{2}}-\mathbf{F}_{i,j-\frac{1}{2}}}{\Delta y}\right] \qquad (1.6)\end{aligned}$$

The last step consists in removing the time derivative on the left hand side of the Equation (1.6). This can be done using a *forward approximation* in time, thanks to which the equation becomes

$$\mathbf{U}_{ij}^{t_n} = \mathbf{U}_{ij}^{t_{n-1}} - \Delta t\left[\frac{\mathbf{F}_{i+\frac{1}{2},j}^{t_{n-1}}-\mathbf{F}_{i-\frac{1}{2},j}^{t_{n-1}}}{\Delta x}+\frac{\mathbf{G}_{i,j+\frac{1}{2}}^{t_{n-1}}-\mathbf{G}_{i,j-\frac{1}{2}}^{t_{n-1}}}{\Delta y}\right] \qquad (1.7)$$

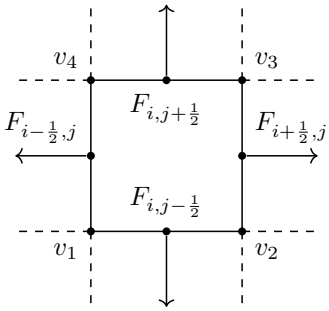
A numerical method is usually evaluated in terms of *convergence* and *order of accuracy*. It is said to be *convergent* if by decreasing the cell size (or equivalently increasing the resolution) the numerical solution approaches the exact one. The order of accuracy gives information on how fast the method converges by increasing the resolution.

The Equation (1.7) is completely specified once the numerical fluxes $\mathbf{F}_{i\pm\frac{1}{2},j}$ and $\mathbf{G}_{i,j\pm\frac{1}{2}}$ are specified. In general, a numerical flux $\mathbf{F}_{i+\frac{1}{2},j}$ is computed as a function H of $l+r+1$ arguments

$$\mathbf{F}_{i+\frac{1}{2},j}^{t_n-1} = H(\mathbf{U}_{i-l,j},\ldots,\mathbf{U}_{i,j},\ldots,\mathbf{U}_{i+r,j}) \qquad (1.8)$$

The arguments $\mathbf{U}_{i_x,j}$, for $i_x \in [i-l, i+r]$, are vectors of conserved variables, spanning l positions on the left (on the x axys) of the cell (i,j) and r positions on its right. In this case, the approximation scheme is said to be *explicit* if the intercell numerical flux at the time-step $t_n - 1$ is specified entirely as a function of the conserved variables at the time-step t_{n-1} (i.e., $\mathbf{U}_{i_x,j}^{t_{n-1}}$ for $i_x \in [i-l, i+r]$). If some of the arguments of the function H are computed at the time-step t_n, the

Figure 1.1: Representation of a square computational cell of coordinates (i, j) having vertexes $v_1, \ldots, v_4$. The four vectors represent the intercell numerical fluxes.



approximation scheme is called *implicit*. A similar approximation is done also for the remaining numerical fluxes (i.e., $\mathbf{F}_{i-\frac{1}{2},j}$ and $\mathbf{G}_{i,j\pm\frac{1}{2}}$).

1.2 Computational challenges

The Section 1.1 discussed how to approximate a physical model defined by a system of PDEs by means of a finite volume method. This process consists in discretizing a physical domain through a set of finite volumes, or computational cells, which content is computed at a discrete time intervals. From a computational point of view, the main task consists in computing the intercell numerical fluxes which in turn are defined as a function of the conserved variables surrounding a specific cell. In the Equation (1.8) we put our self in a simplified position because the physical domains is discretized using a Cartesian grid. Although this is a good starting point to describe a numerical method thanks to the simplicity of its description and implementation, such type of grid cannot always be used to solve practical problems. The section 1.2.1 discusses the limitations of the Cartesian grids and a few known alternatives. The FV method described by the integration time Equation (1.7), can be implemented using an iterative algorithm running indefinitely until a certain time threshold T_{Max} is reached. During each

time-step the content of each cell is updated based on the content of a set of conserved variables at the previous time-step (recall that explicit methods are assumed). A crucial aspect of a simulation algorithm, besides its correctness, is the *performance*, which generally describes how fast a simulation can be run. From a mathematical point of view, to obtain greater performance, a possibility is to work on the approximation scheme by searching for a way of performing less operations without losing the accuracy. From a computational point of view the algorithm can be optimized for the underlying hardware architecture. The latter is the main focus of this thesis, and in section 1.3 describes the main techniques adopted during an optimization process.

1.2.1 Types of grids

The implementation of explicit schemes based on Cartesian grids is straightforward. The reason behind that is the memory representation of the grid which preserves the neighboring relations between physical cells and logical (computational) ones, meaning that there is a one-to-one mapping between the physical domain and the logical one. This property simplifies the implementation of the neighbor retrieval operation (NRO). In particular, a cell of (i, j) has the same coordinates in both spaces (i.e., physical and logical) and its neighbors can be found as a function of its coordinates by varying i along the x axys and j along the y one. This kind of relation between (i, j) and its neighbors is referred to as *spatial locality*. Besides being very simple to implement, using this type of grid is also very efficient from a computational point of view since the physical representation is the same has the logical representation. It follows that no additional data structures need to be queried in order to implement the NRO because the memory representation preserves the data locality between neighboring cells.

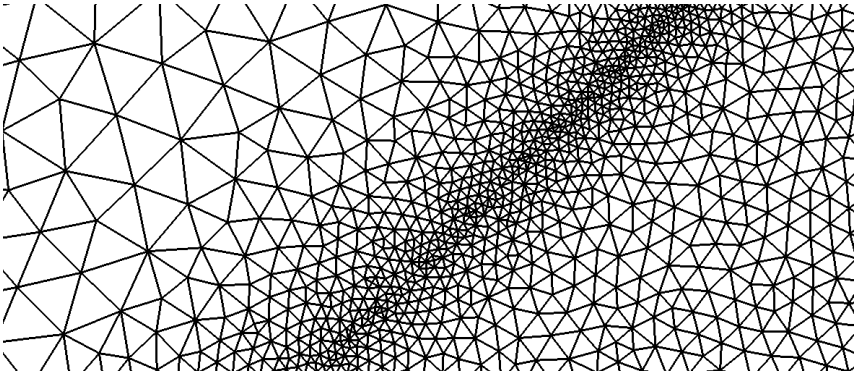
The term *resolution* refers to the size of a cell: the smaller the cell, the higher is its resolution. If from one hand, the Cartesian grid is algorithmically simple

to handle, from the other hand the entire domain must be discretized using the same level of resolution, resulting in an large number of memory cells to be stored and processed. This happens when high resolution is needed in order to obtain high accuracy in some specific domain points, forcing the rest of the domain to be discretized with the same (high) resolution level. Even if a single cell can be processed very rapidly, in practical cases the number of cells to be processed by a numerical algorithm is often too high to justify the use of a Cartesian grid.

A common alternative to a Cartesian grid is the *unstructured grid*, in which volumes do not have a regular shape and size. An example of unstructured grid is depicted in Figure 1.2. In general, it is not possible to store an unstructured grid into memory maintaining the data locality between neighboring cells. For this reason, to find the neighbors of a volume, a graph based data structure must be queried all the times, making the NRO computationally less efficient because, unlike Cartesian grids, data locality cannot be exploited. However, thanks to the variable size of each volume, unstructured grids allow to adopt higher resolution only on domain section of particular interests, and impose lower lever of resolution elsewhere, reducing considerably the number of volumes to be processed, allowing simulation algorithms to obtain a greater performance if compared to those working on Cartesian grids.

PARFLOOD adopts an *hybrid* grid obtained by discretizing the physical domain into a set of blocks with variable resolution. In previous works [18], the grid has been has been referred to has *Block Uniform Quadtree grid* (BUQ) because the spatial subdivision has a Quadtree-like structure which remains constant in time. As depicted in Figure 1.3, the grid tiles a set of square matrices with different resolution levels and constant number of elements (e.g. $2^k \times 2^k$, $k = 4$). These matrices are named *blocks* and their elements are the computational cells of the FV method described in Section 1.1. The side of a block is assumed to be parallel to one of the Cartesian axes. The numerical model that handles multi

Figure 1.2: A triangle mesh.



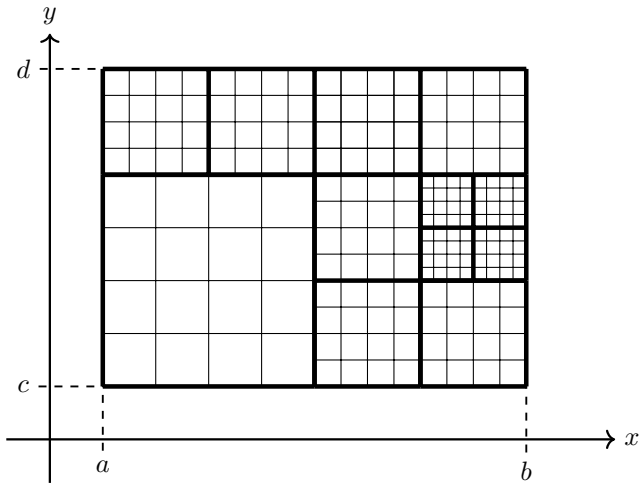
resolution cells requires three constraints on the grid. The first imposes that if two blocks share one side, they must share at least a corner (block alignment). Blocks sharing at least a point are named *neighbors*. The second constraint forces the length of a block side to be 2^{k+r}, where $r \geq 0$ is an integer named *resolution*. The third and last constraint allows two neighbors to change their resolution by at most one unit. Formally, given a pair of neighbors s and t with resolution r_s and r_t respectively, then $|r_s - r_t| \leq 1$.

Any memorization of the set of blocks cannot preserve the logical locality of neighbors, due to the complex multi resolution tiling. The memory representation designed for this implementation is presented in Section 3.2 whilst the neighbor retrieval operation between adjacent cells is detailed in Section 3.3 .

1.3 Need for HPC

The numerical model resulting from the Equation (1.7) is implemented through a simple simulation algorithm which has the structure highlighted in the Listing 1.1. In particular, the outer `while` loop advances the time-step of a discrete value Δ_t once all the numerical fluxes have been computed between each pair of adjacent

Figure 1.3: A bounding-box $[a, b] \times [c, d]$ discretized through a BUQG having three different resolution levels and containing 12 blocks.



cells and their respective conserved variables have been updated inside the `for` inner loop.

Listing 1.1

```
1   t ← 0
2   // let G be a domain discretization
3   while(t < T_max){
4     // compute the minimum Δ_t
5     for( v, z ∈ G ){ // for each pair of adjacent volumes in G
6         // compute the numerical flux between v and z
7         // update the conserved variables
8       }
9       t ← t + Δ_t // update Δ_t
10    }
11  }
```

Notice that the inner loop performs the same sequence of operation between each

pair of adjacent volumes and every iteration is completely independent from the others, meaning that the `for` loop can be parallelized. More in general, each time the same sequence of operations is performed over different (unrelated) data, we can say it is parallelizable. Thanks to the relevance of the problems being solved in a multitude of science disciplines (see Chapter 2), an entire field named *High Performance Computing* (HPC) has been created in recent years with the specific goal of building hardware architectures and exploit them through highly optimized algorithms. Two important metrics often used in HPC to evaluate numerical algorithms are *performance* and *scalability*. The performance compares the speed of the algorithm in relation to the underlying hardware used (e.g., an algorithm optimized for GPU can *perform* better than the same algorithm optimized for CPU, or vice-versa). Different mathematical models can lead to algorithms with different performances, but this is beyond the focus of this thesis. The scalability can be classified in *strong scalability* and *weak scalability*. The former evaluates the speedups obtained by increasing the computational resources (e.g., increasing the number of CPUs) while keeping constant the input size. The latter evaluates the algorithmic overheads introduced to deal with multiple computational units.

1.4 Goals of this thesis

The solver tries to compensate the memory limitations of a single GPU by using the multi-resolution grid defined in Section 1.2.1. Despite having achieved excellent results, the single GPU implementation lacks in *scalability* (see 1.3).

The study carried out in [18] gives an idea on the performance achieved by the single GPU implementation. In particular such version has been used to simulate a realistic (still hypothetical) case scenario in which a levee breach in the Secchia River (northern Italy) inundates a domain having an extension of $840km^2$. Such domain is discretized with a BUQ grid having $7,3 \times 10^6$ cells which resolution

varies from $[5 \times 5]m$ to $[40 \times 40]m$. Such simulation took 22 hours to simulate an event lasting $80h$, thus reaching a simulation time to physical time of $1/4$. By discretizing the same domain using a Cartesian grid, approximately 100×10^6 cells are needed, making the domain size too big to be stored even on the most recent GPUs (i.e., 16 to 32 GB on NVIDIA $V100$). Even if the BUQ grid extends the domain size and/or the simulation speed compared to the Cartesian grid, there is a visible limitation on the performance and scalability which can be obtained by the solver.

In short, the motivation behind the work of this thesis is that by using a single GPU, both the size of the input grids and the simulations speedups cannot be increased any further.

The goal of this thesis is to overcome such limitations as described in Section 1.4.

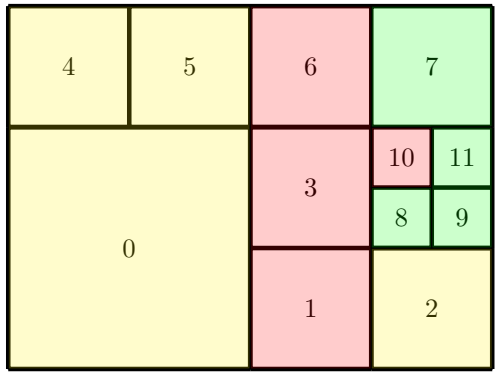
The goal of this thesis is to extend the code of PARFLOOD, allowing it to perform simulations on multiple GPUs, abstracting from their locality (i.e., GPUs may or may not belong to the same machine).

To achieve this, it is necessary to perform a *domain partitioning* (Section 1.4.1), implement a *border communication* (Section 1.4.2) to allow adjacent partitions to exchange border data (e.g., conserved variables), and finally guarantee that all partitions are *computationally balanced* by implementing a *dynamic load balancing* heuristic (Section 1.4.3). These are the goal of the present work and they will be detailed in the following three sections.

1.4.1 Domain partitioning

In this context, the *domain partitioning* is the process in which the input grid (i.e., the physical domain discretization) is partitioned in different pieces, which are called partitions. More in detail, the first step is to compute the barycenter

Figure 1.4: A simple domain decomposition in three partitions, each one with a different color. $P_1 = \{0, 2, 4, 5\}$, $P_2 = \{1, 3, 6, 10\}$ and $P_3 = \{7, 8, 9, 11\}$



of each block (recall that a block is a square matrix of cells), represented by a point in the $2D$ Cartesian space, and then resulting points are used as input to a partitioning algorithm. Formally, given a set $P = \{p_1, \ldots, p_n\}$ where $p_i \in \mathbb{F} \times \mathbb{F}$ is a pair numbers belonging to the floating point domain ($\mathbb{F}$), the domain partitioning computes a sequence of sets (partitions) $P_1, \ldots, P_k$, such that $P_i \cap P_j = \emptyset$ for $i \neq j$, and $\cup P_i = P$ for $i = 1, \ldots, k$. The Figure 1.4 shows a simple partitioning in three different parts of the grid shown in Figure 1.3 from which the cells are omitted and the blocks are uniquely identified by an integer number. Since there is a unique mapping between the block barycenter and the block identifier, we can represent the partitions in a more compact way using the latter.

Each partition represents a portion of grid which is then assigned to a different (dedicated) GPU. Notice that the partitioning is performed block-wise, meaning that all the computational cells belonging to the same block are assigned to the same partition (GPU).

1.4.2 Border communication

As discussed in Section 1.2, from an implementative point of view the main problem of a finite volume method, consists in computing the intercell numerical fluxes between each pair of adjacent cells. Partitions having at least a cell needing information from a remote neighbor (i.e., a neighbor cell located on another partition) are called *adjacent partitions*, whilst those cells needing information from a remote neighbor are called *border cells*. For example, the partition P_1 shown in Figure 1.4 is adjacent to both P_2 and P_3 and the only block in the entire gird without border cells is the block 4. A further complication is that PARFLOOD implements conservation laws between volumes having different resolution levels, meaning that some interpolations need to be performed as it will be discussed in Section 3.3.

In this context, the *border communication* consists in allowing adjacent partitions to exchange their border cells. Since the communications between adjacent partitions, in general, are carried out over a network, it is important to make this process as efficient as possible to minimize any overhead due to the network delays. The communication between adjacent partitions is carried out using Open-MPI [22], an implementation of the *Message Passing Interface* standard [23] which provides an high level set of library calls for exchanging messages between a group of processes. The Sections 3.5 and 3.6 provide a detailed description on how MPI has been used to implement the border communication.

1.4.3 Dynamic load balancing

The *dynamic load balancing* (DLB) aims to guarantee that all partitions have (more or less) the same time-step duration. If such property is not achieved, some partitions requiring less "computational work" will terminate the time-step faster than others. In this way the elapsed time of the simulation is determined entirely by the partition having the longest time-step. From an hardware point

of view, this means that some computational units (in our case the GPUs) are busier than others which in turn spend idle time waiting for the time-step to end. From a computational point of view, DLB can be defined as the process aiming at distributing the same amount of computational work to each GPU involved in the simulation. Clearly this statement is true under the assumption that all GPUs are identical.

Since the computational work of a GPU is mainly based on the number of cells to be processed, it is important to guarantee that (more or less) each device processes the same number of cells. Intuitively, this means that it is possible to obtain a balanced situation from an unbalanced one by moving cells from overabundant partitions to lightweight ones. In our case, the input domain can be partitioned in a balanced way from the start of the simulation, however, this situation is not stable because in flood propagation over initial dry bathymetries the number of cells which need to be processed varies at runtime.

For this reason it is necessary to dynamically balance the workload among partitions as the simulation advances. The computational balance is slightly influenced by other minor factors such as the network uneven performances and the differences in the computational load arising from the divergence of the computational paths among different partitions. All this factors will be addressed in Section 3.9 once the GPU architecture will be introduced in Section 2.5.

2

The HPC Environment

High Performance Computing (HPC) is a general term referring to the exploitation of large computational resources to efficiently solve a problem. Such resources are represented by large clusters made of hundreds or thousands of nodes interconnected with each other and containing the most recent hardware architectures. For this reason there is a big competition between hardware vendors which are interested in putting their products on the most powerful machines. According to the Top-500 [24], a ranking of the 500 most powerful clusters on Earth, the top supercomputers are currently working in the *petascale*, meaning that they are able to perform an order of 10^{12} floating point operations per seconds (FLOPS), and it seems to be only matter of time before we can finally reach the *exascale*.

HPC is not only concerned with the construction of supercomputers but also with the implementation of programs capable of exploiting the underlying hardware architectures. To this end in recent years we assisted to the birth of research centres, often referred to as *supercomputing centres*, where supercomputers are hosted and maintained. This places represent the middle ground between the academic and the corporate worlds in that companies submit practical problems to research centres which in turn work out an efficient solution. On the other hand

research centres work along side with the academic world to solve new challenging problems and implementing new ideas.

The following examples highlight the rich variety of problems which are solved everyday in these centres. In a Molecular Dynamics (MD) study [25] a supercomputer has been used to develop a deeper understanding of the process behind the gastric acid production in the interior stomach lining. In the field of Physics, a new simulation method [26] has been developed in order to get an insight on the complex processes happening inside the neutron stars. A Geophysics study [27] have been carried out using a supercomputer in order to examine the seismic waves traveling through Mars with the goal of investigating the internal structure of the red planet. The data was provided by a seismometer installed on the InSight space probe launched on May $5th$ 2018. An Earth Science study [28] reconstructed the history of glaciation in the Alps to better understand the mechanisms behind the glaciation.

All these and a lot of other problems are solved with the aid of a supercomputer executing highly optimized codes. Such codes usually simulate some sort of realistic event, depending on the discipline, and giving important insights on the topic which are otherwise impossible to obtain on a normal computer due to the large computational time it would require and the lack of computational resources.

HPC applications are usually implemented by *computational scientists* (i.e., scientists having programming skills) and are able to scale up to hundreds or thousands of nodes, still maintaining an acceptable level of efficiency. Unfortunately nowadays we don't have high level programming languages capable of transparently exploit the underlying hardware architecture. Even though some languages have been created with this goal in mind [29] [30] [31] [32] [33] [34] [35], today due to the variety of parallel architectures and applications present in the HPC world, there isn't any widely used standard language which can be used to write highly optimized and portable programs for different hardware architectures. For this

reason computational scientists serves as support to the science world, providing software packages and high-level libraries which allows a broader set of people to take advantage of supercomputers.

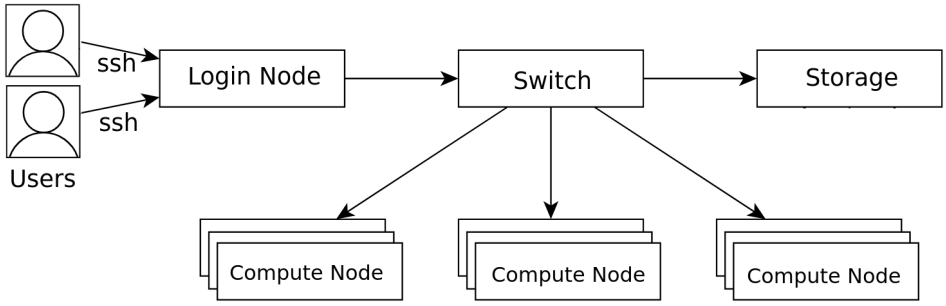
This chapter provides a brief introduction to the HPC environment, starting from a general description of the cluster architectures in Section 2.1 and highlighting the main performance metrics used to evaluate an HPC program in Sections 2.2 and 2.3. The following sections provide the reader with a more detailed discussion of the hardware architectures inside a compute node, starting from the memory hierarchy exploited by the CPU (Section 2.4) thanks to various parallelism levels, together with a brief introduction to the NVIDIA GPU parallelism model (Section 2.5). The chapter concludes with a quick discussion on MPI (Section 2.6), which is the standard way to implement multi-node multiprocessing in HPC systems.

2.1 General Architecture of HPC Clusters

An HPC *cluster* mainly consists of a set of *compute nodes* interconnected by a fast network (e.g., Ethernet [36] and Infiniband [37]) and working together as a single system. Generally, each compute node runs a lightweight operating system that: 1. enables scientific applications; 2. obtains the best performance possible out of the underlying hardware; 3. minimizes the overheads of the operating system's services. Usually the compute nodes are identical systems, both at the software and hardware level, as it facilitates maintenance operations such as packages installation and system configuration. Usually a *login node* (or *master node*) is used as front-end and it is accessible by users through the *Secure SHell* (SSH) protocol as shown in Figure 2.1. Such node is used to load the *environment modules* containing the software needed to compile the code, launching *job scripts* and it is also used by system administrators to perform maintenance operations, system monitoring and configuration. For large clusters, more than one master node can

t

Figure 2.1: General Structure of an HPC cluster.



be present, for example one for login and one for administration purposes. The key aspect is that a master node is not intended to perform computations, which instead are demanded to compute nodes. Usually compute nodes have little or no storage space and the task of storing and serving data among the compute nodes is demanded to a network of *storage servers* having high bandwidth and easily scalable by adding new storage nodes. There is a software layer, generally referred to as *parallel file-system*, making the storage servers working together as a single file-system. The user is offered with a transparent way of storing data and managing his own directories without worrying about the underlying network of storage servers. Two examples of parallel file-system are LUSTRE [38] and IBM GPFS [39].

Both login and compute nodes share multiple file-systems (i.e., multiple storage areas) used to store user data such as input/output files, post-processing files, source codes and scripts. From the point of view of the user, switching from one file-system to another one is as easy as change directory in a desktop computer. Two common file-system present on most HPC clusters are called *home* and *scratch*. The former is the file-system where usually the user logs-in and can be used to (almost) persistently store application data such as programs and input-

output files in a *reliable* way. The latter prioritizes *performance* over *reliability* making it ideal to run applications. Usually *scratch* file-systems are periodically cleaned up by a system script, forcing the user to move important data (if any) produced by the application to some reliable file system such as *home*. In some clusters the *scratch* is used to store only small amount of data, such as programs and scripts and provide another storage area to save larger files such as input/output data.

2.1.1 Architectures and Parallelism Levels

At the *hardware level* the structure of a compute node varies depending on the vendor's design, however, nowadays HPC clusters are usually equipped with *superscalar multiprocessors* and/or general purpose *Graphic Processing Units* (GPUs). The former are CPU architectures, thus capable of running an operating system and containing a variable number of cores, depending on the vendor design. The latter is a CPU *co-processor*, thus unable to work without the aid of a CPU.

Most CPUs implement a *Data Level Parallelism* (DLP) in the form of *Single Instruction Multiple Data* (SIMD) instructions, which will be detailed in Section 2.4.1. An *Instruction Level Parallelism* (ILP) is also available through various techniques such as pipelining, out-of-order execution and the presence of multiple execution units. These techniques are mainly supported by the compiler and the CPU hardware units [40]. The best way for an application to take advantage of multiple cores is by using (where possible) a *thread level parallelism* (TLP) in which multiple *threads* work concurrently sharing the same address space. This kind of parallelism will be detailed in Section 2.4.2.

A different type of architecture is represented by the GPU, which is based on the *Single Instruction Multiple Thread* (SIMT) parallelism having its differences and similarities with the CPU SIMD and multi-threading. SIMT is detailed in Section 2.5. SIMD, SIMT and multi-threading are all optimization happening

inside a single compute node. The Section 2.6, describes how an application can run on *multiple compute nodes* using the *Single Program Multiple Data* (SPMD) paradigm.

2.1.2 Job Scheduler

Since an HPC cluster is a multi-user system where each user wants his simulations to run as efficiently as possible, it is important to prevent different applications to use the same hardware resources at the same time, which leads to an application slowdown. For this reason, on HPC systems there is a program referred to as *job scheduler* or *queue manager* which takes in input a user script specifying the set of *resources* needed to run the user application. System resources can include *hardware resources*, for example the number of CPUs to use and the number of processes to spawn, or *time resources* specifying the user-estimated duration of the program execution. If the user-requested resources are not available at the submission of the script it's because those resources have been assigned already to other users which are currently running their jobs. For this reason the *job scheduler* inserts user requests which cannot be immediately satisfied, inside a queue containing (in general) requests from other users. As soon as the resources are available, the job scheduler extracts a satisfiable user request from the queue, and runs the user-specified application on the hardware resources requested. Thanks to the job scheduler, HPC users don't need to wait for the resources to become available, as their jobs are automatically started when the system is ready.

The Listing 2.1 contains an example of job script for the open source SLURM queue manager [41] written inside a file called `example_script.sh`. Each line starting with the keyword `#SBATCH` contains a resource request. For example the first line asks the queue manager to run the user application on a single compute node while the second specifies that the application should be run as a single process. As the `--constraint=gpu` option specifies, the program takes advantage

of a single GPU and with the `--time=00:10:00` option the time limit is set to ten minutes. If after the time limit the application is still running, it will be terminated by the system. SLURM allows for particularly fine-grained resources requests. To submit a job-script in slurm the program `sbatch` is used, which takes care of reading the input file containing the script and submitting the job to the queue manager, which usually responds with the job id, in this case the id is `15407622`. When the job is completed, unless otherwise specified, the output is saved into a file containing the job id in its name (e.g., in this case `slurm-15407622.out`).

Listing 2.1

```
> cat example_script.sh
#!/bin/bash -l

#SBATCH --nodes=1 # Number of compute nodes
#SBATCH --ntasks=1 # Number (SPMD) processes to run
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks-per-core=1
#SBATCH --cpus-per-task=1
#SBATCH --constraint=gpu # 1 GPU per node
#SBATCH --time=00:10:00 # Time limit

# running the user application
# < command to execute the user program >
> sbatch example_script.sh
Submitted batch job 15407622
> ls
slurm-15407622.out
```

2.2 Performance and Scalability

As briefly mentioned in Section 1.3 the two main metrics to evaluate the implementation of HPC applications are *performance* and *scalability*. From a computational

point of view, the performance evaluates the level of optimization of an algorithm in relation with the underlying hardware architecture. In the HPC environment the most common metric evaluates the top speed, also called *peak performance*, of an hardware device (e.g., CPU or a GPU) in terms of *Floating Point Operations Per Seconds* (FLOPS). Another common metric is the *memory bandwidth* which is used to quantify the amount of data (e.g., *bytes*) which can be exchanged in a time unit between two devices. Another common metric evaluates the average *Instructions Per Cycle* (IPC) of a CPU to compare the amount of time spent in computation and memory operations.

The performance metrics highlighted so far do not provide any information on how well the HPC system is exploited as a whole. Such metrics provide information on how well the various hardware architectures are exploited considering them as separate units. However, an HPC application (in general) consists in a set of software modules which can be optimized for a number of different hardware architectures such as many-cores CPUs and haywire accelerators like GPUs, all of them spread across a network of compute nodes. For this reason it is important to take into account the interaction between the different hardware components and not just the optimization level applied to each one of them. A typical example in a CPU-GPU application can be the lack of overlapping of the CPU tasks with the GPU ones where possible, which creates idle times for both devices. Another example can be the presence of bottlenecks in a multi-process application arising from blocking communications between processes. To asses the performance of an HPC application, an important metric is the *scalability* which can be divided in *weak* and *strong scalability*.

The *strong scalability* test evaluates the speedups obtained by running the same application a number of times by keeping constant the input size and increasing the processing units across different runs. In this context, the term *processing unit* (PU) refers to an hardware unit, whether is a CPU core, an entire CPU or a GPU.

A typical example of strong scaling from the field of Computational Fluid Dynamics (CFD) consists in splitting an input grid in n parts where n is the number of PUs, each one performing a computation on a different partition of the grid. By indicating with T_i the running time of the algorithm using i PUs, the speedup over the same application run on 1 PU is defined as the ratio T_1/T_i. In normal conditions, such ratio spans in the interval $[1, i]$, with 1 and i being the worst and the best possible values respectively. The quality of the strong scalability can be measured through the efficiency $\mathrm{SS}_{\mathrm{Eff}}$, and expressed as a percentage by the formula 2.1.

$$\mathrm{SS}_{\mathrm{Eff}} = \frac{T_1}{i \cdot T_i} \cdot 100 \tag{2.1}$$

If a solver strong scales at 100% on i processing units and i partitions, it means that the speedup is equal to i. By increasing the the number of hardware units, sooner or later the efficiency drops to a point in which the software does not scale anymore. From that point on, increasing the number of hardware units implies a waste computational resources and a power consumption in general.

The strong scalability test represent a good starting point to asses the amount of computational resources which should be used for a given input size and to evaluate the sources of inefficiency through a deep *profiling* (i.e., analysis of the timings of the various code parts to evaluate their efficiency) of the code. In the field of CFD a common loss of efficiency in the strong scaling is related to synchronization steps such as the global time-step reduction (i.e., the Δ_t reduction) (see 3.7) as well as overheads arising from the border communication between adjacent partitions (see Chapter 4).

The *weak scalability* is computed by increasing the number of processing units together with the number of partitions, keeping constant the partition size. Thus a simulation involving n hardware units is performed on an input grid having $n \cdot C$ cells, where C is the (constant) partition size. The weak scaling efficiency ($\mathrm{WS}_{\mathrm{Eff}}$)

is expressed as a percentage using the formula 2.2.

$$\mathrm{WS}_{\mathrm{Eff}} = \frac{T_1}{T_i} \cdot 100 \tag{2.2}$$

Since the partition size is constant, if the efficiency is less than 100% it means that by increasing the hardware units some overheads are introduced by the parallel implementation of the code (see Chapter 4).

2.3 Load Balance

The Section 2.2 silently assumes that the computational load among different partitions, both in the strong and weak scaling tests, is balanced. In other words, each hardware unit performs the same amount of work throughout the simulation. This is a rather unrealistic situation in HPC applications. The fact that input data is equally split among different hardware units does not imply that all of them perform the same amount of computational work.

An example from Particle Physics [42] applications involves the spatial mobility of particles in space which implies the possibility for such particles to move across different partitions. In such cases, noticeable differences in the computational load may arise among processes due to the different number of particles to be processed, making the load balancing a priority in order to obtain a scalable application.

To give another example, the multi-GPU implementation of PARFLOOD, deals with the imbalances arising from the evolution of *dry-wet* fronts, emulating the water propagation on the domain surface together with the possibility for some zones to dry up. In this case some imbalances arise because the code is optimized to exclude dry cells not adjacent to the wet ones from the computation (see Section 3.4 for more details) meaning that partitions having more dry cells are computationally lighter than the others.

Another assumption which has been made so far is the hardware uniformity between processes, meaning that each parallel process takes advantage of one or more processing units of the same type and architecture. For example in the case of PARFLOOD, a single GPU is managed by a single MPI process, offloading the computational kernels entirely on the GPU. However, nowadays supercomputers are made of nodes containing both super-scalar CPUs and accelerators (mostly NVIDIA GPUs).

For this reason, requesting both a CPU and a GPU and then running the majority of the simulation on just one of them seems a rather inefficient waste of computational resources. The ideal case would be to exploit both architectures at runtime, raising once more the load balancing problem following from the need to distribute the workload across PUs functioning at different speed (refer to [43] for a complete survey on the topic).

The implementation of the dynamic load balancing is generally an hard task to accomplish from an implementative point of view, and there isn't a standard way to do it, mainly because it is strongly application dependent and it is difficult to find an optimal solution for every implementation. A rather general (not necessarily optimal) approach in the field of CFD seems to be the minimization of *idle times* caused by some *synchronization barrier*. This heuristic simply gathers information on the time spent by each process in an idle status (i.e., the time spent without performing any task). Intuitively, processes spending more time stuck at the barrier are wasting their computational resources which can be lent to other processes in the same node [44].

A different approach (adopted inside PARFLOOD), consists in using MPI to migrate grid portions from the processes having low idle times to the ones having an higher idle time. The Section 3.9 gives all the details and the Chapter 4 presents the main results.

2.4 CPU Parallelism levels

CPUs take advantage of multiple memory levels which can be exploited by different types parallelism. At the lowest level (core level) we can find the the CPU *registers* usually storing the operands of the assembly instructions executed by the CPU. As detailed in Section 2.4.1, modern CPUs include special set of registers exploiting a *Data level Parallelism*. Above the register memory there is the *cache memory*, usually further divided in multiple levels and representing the middle ground between the main memory (i.e., *Random Access Memory* RAM) and the CPU. Since the RAM fetches are very slow compared to the CPU speed, the caches represent a way to mitigate and possibly overcome the memory overheads. The cache memory is by far smaller than the RAM in terms of storage capacity but at least an order of magnitude faster in terms of bandwidth (usually given in *Gigabytes per second*, GB/s).

The cache is located near the CPU and is usually divided into three different levels which are L_1, L_2 and L_3. The first level (L_1) is usually divided into two different areas, one to store the code instructions ($L1_i$) and one for the data ($L1_d$). The second level (L_2) is bigger than the first one, but still fast. Each CPU is usually made of a number of compute units called *CPU cores*, each one having its private L_1 and L_2 caches while the third level (L_3) is shared among different cores. Some realistic capacities associated with the three levels of caches are $64KB$ for L_1, $256KB$ for L_2 and $4MB$ for L_3.

When the CPU asks for data which are not located inside the cache memory (i.e., *cache miss*), a RAM fetch is performed, which is an expensive operation (see [45] for an example of memory latencies on Intel architecture). In *memory-bounded* applications, where memory fetches are more expensive than calculations, it is important to minimize the cache misses. CFD applications belong to such class [40] due to their need to update their computational units (whether they are grid

cells or particles) by constantly gathering information from their neighborhood in space.

Cache misses can be decreased with a careful design of the data structures by the programmer and a few optimizations performed at compile time [46]. The cache memory can be exploited by *multi-threading* parallelism, where multiple *threads*, usually assigned to different CPU cores, "live" in the same address space and share the level L_3, making the communication between threads very fast. In the HPC environment multi-threading is usually associated with OpenMP which offers an easy way to parallelize a region of code (usually a loop) by spawning a number of threads executing concurrently (see Section 2.4.2 for an example).

As said earlier the main memory (RAM) is much larger if compared to the caches (e.g., generally in the order of 10^2 GB) with the drawback of being slower. If two or more CPUs are present on the same node, the usual approach follows the *Non Uniform Memory Access* (NUMA) design, in which the memory is partitioned in different pieces, each one associated to one CPU, allowing them to access their memory partition concurrently. This memory level (RAM) can be exploited by *multi-processing*, where different CPU processes (i.e., not sharing the same address space) communicate by exchanging messages. Multi-processing (in HPC) is usually exploited through the *Message Passing Interface* (MPI) standard, consisting in a set of high-level library calls facilitating the implementation of the communications between parallel processes (see Section 2.6 for an example).

A further level of memory exploitable by MPI is the *storage memory*, a non-volatile type of memory usually associated to input/output (I/O) operations. As already mentioned in Section 2.1, an HPC system generally deploys a network of *storage servers* which is transparent to the user (i.e, is seen as a single storage unit, or a group of units in case of multiple file-systems). In the HPC environment applications usually perform frequent I/O operations for reading input data and periodically store the outputs. Due to the high latencies of the storage memory [45]

(i.e., can be 10^2 to 10^3 times slower compared to the main memory) considerable bottlenecks may arise at runtime. For this reason it is mandatory to optimize the I/O process as much as possible, otherwise the memory access optimizations performed at the other levels become pointless.

A common source of inefficiency arising during the I/O stage is represented by the serialization of memory operations among processes. This means that I/O operations are either performed by one process at the time or just one *master* process performs them while the others are idle. Provided that there are no race-conditions (i.e., logical dependencies) between I/O operations it is possible to exploit MPI to enable processes to perform readings and writings on the same file in a parallel fashion. The Section 2.6 gives a simple example of parallel output on a file.

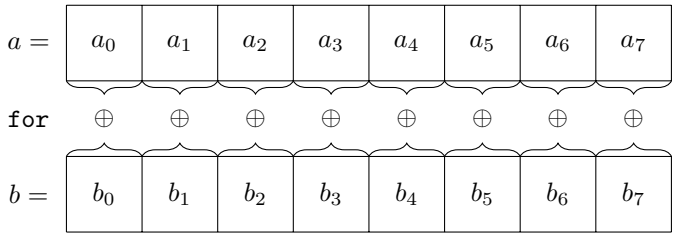
The following sections give a more in-depth description of the parallelism levels exploited by the CPU inside a compute node. In particular, the Section 2.4.1 describes the data level parallelism exploited by CPU registers whilst the Section 2.4.2 highlights the main features of the multi-threading parallelism using OpenMP.

2.4.1 SIMD instructions

Single Instruction Multiple Data (SIMD) is a type of parallelism performed by each core and simply consists in using special CPU registers to perform the same operation over multiple operands (data). CFD applications usually perform the same operation over a lot of different memory regions, whether they represent grid cells or particles, it is convenient to find a way of packing data in contiguous chunks of memory (CPU registers) and process each chunk with a single assembly operation. Such approach is usually referred to as *vectorization* and the following part of this section discusses an example showing how to activate it and what happens at the low level assembly code once it is active.

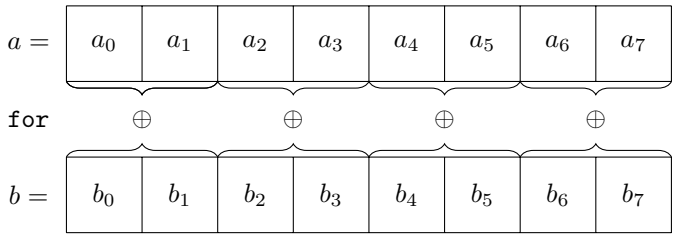
The common situation in which vectorization is applicable is depicted in Figure 2.2 in which a generic operation $\oplus$ (e.g., an addition) is performed between the entries of two arrays of the same size (i.e., a and b) in an aligned fashion.

Figure 2.2: A classical vector sum implemented with a simple `for` loop.



A serial implementation of such operation consists in a `for` loop computing $a_i \oplus b_i$ between each entry pair. It follows that the number of operations is equal to the size of the vectors (i.e., eight in the Figure 2.2). The goal of the vectorization is to decrease the number of operations $\oplus$ by exploiting special CPU registers, called *vector registers*, to perform the same operation over multiple data with the same assembly instruction. For example in Figure 2.3 the operation $\oplus$ is iteratively carried out between pairs of entries, meaning that instead of computing $a_i \oplus b_i$ and then $a_{i+1} \oplus b_{i+1}$, the CPU computes directly $[a_i, a_{i+1}] \oplus [b_i, b_{i+1}]$ thus halving the number of $\oplus$ operations performed.

Figure 2.3: Vector sum obtained by activating the vectorization.



A more in-depth analysis of what really happens to a program when the vector-

ization is active can be carried out by using a debugger. Substantially, a debugger is a software which allows a programmer to inspect (and possibly modify) the content of the memory as well as the content of the CPU registers during a program execution. With that in mind, in the following part of this section a debugger is used to highlight the key aspects of the SIMD parallelism and the vectorization by showing what are the differences between a serial code and a vectorized one.

Consider the program contained in the Listing 2.2 which performs the sum of two vectors `a` and `b`, the former containing odd numbers and the latter containing even ones, both in the interval $[0, 15]$. The result of the sum is stored in `c` (line 11). The goal of this example is to show that when vectorization is not active, the program loops over the input arrays `a` and `b`, performing one addiction at the time. Instead by activating the vectorization, multiple operands of `a` are added to the corresponding operands of `b` in a single operation, thus speeding up the computation of `a+b` of a factor proportional to the number of operands added simultaneously.

Listing 2.2

```
1  #include <stdio.h>
2  #define LEN 8
3  typedef double FP;
4
5  FP a[] = {1,3,5,7,9,11,13,15};
6  FP b[] = {0,2,4,6,8,10,12,14};
7  FP c[LEN];
8
9  int  main(){
10 for(int i=0; i<LEN; i=i+1){
11         c[i] = a[i] + b[i]; // Compute a + b
12 }
13
14 for(int i=0; i<LEN; i=i+1){
```

```
15          printf("c_%d is %f\n", i,c[i]);
16  }
17
18  return 0;
19  }
```

Let's examine what happens by normally compile this code without any optimization

Listing 2.3

```
1   > lscpu | grep "Model name"
2   Model name: Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
3   > gcc --version
4   gcc (GCC) 6.2.0 20160822 (Cray Inc.)
5   Copyright (C) 2016 Free Software Foundation, Inc.
6   # Omitting the full output ...
7   > gcc -o stdsum vectorization.c
8   > objdump -M intel -D stdsum --no-show-raw-insn \
9   | grep "<main>" -A20
10  0000000000400526 <main>:
11    400526:  push   rbp
12    400527:  mov    rbp,rsp
13    40052a:  sub    rsp,0x20
14    40052e:  mov    DWORD PTR [rbp-0x4],0x0
15    400535:  jmp    400569 <main+0x43>
16    400537:  mov    eax,DWORD PTR [rbp-0x4] # loop-start
17    40053a:  cdqe
18    40053c:  movsd  xmm1,QWORD PTR [rax*8+0x601060]
19    400545:  mov    eax,DWORD PTR [rbp-0x4]
20    400548:  cdqe
21    40054a:  movsd  xmm0,QWORD PTR [rax*8+0x6010a0]
22    400553:  addsd  xmm0,xmm1 # sum
23    400557:  mov    eax,DWORD PTR [rbp-0x4]
24    40055a:  cdqe
25    40055c:  movsd  QWORD PTR [rax*8+0x601100],xmm0
```

```
26    400565:  add    DWORD PTR [rbp-0x4],0x1
27    400569:  cmp    DWORD PTR [rbp-0x4],0x7
28    40056d:  jle    400537 <main+0x11>  # loop-end
29    40056f:  mov    DWORD PTR [rbp-0x8],0x0
30    400576:  jmp    4005a6 <main+0x80>
```

The Listing 2.3 assumes our program being written inside a source file named `vectorization.c` which was compiled using `gcc` without providing any optimization option. The CPU in use on the system is an Intel Xeon E5-2650 v3. The tool `objdump` is used in conjunction with *grep* to have a quick look at the first 20 lines of assembly code associated with the main function and contained inside the binary file produced by the compilation (i.e., `stdsum`). The option `-M` is used to set the assembly language to Intel. The for loop executing the sum is contained between the addresses `0x400537` (line 16) and `0x40056d` (line 28). The instruction at the address `0x400569` checks if $i \leq 7$ (i.e., the Boolean condition inside the `for` loop), in which case the instruction `jle` makes the execution jump to the address `0x400537`. The addition is actually performed at `0x400553` (line 22) by the instruction `addsd` which add the content of register `xmm0` to `xmm1` and stores the result into `xmm0`. The instruction `addsd` belongs to the *Streaming SIMD Extension* (SSE) designed by Intel and its operands are 128 bits vector registers. Since no optimization is activated during the compilation let's use the *GNU DeBugger* `gdb` to examine the content of such registers during the execution (Listing 2.4).

Listing 2.4

```
1  > echo "set disassembly intel" > gdb_config # Intel assembly
2  > gdb stdsum -q -x gdb_config
3  Reading symbols from stdsum...done.
4  (gdb) break *0x400553 # setting the breakpoint
5  Breakpoint 1 at 0x400553
6  (gdb) r
7  Breakpoint 1, 0x0000000000400553 in main ()
```

```
8  (gdb) x/i $rip # examine the current instruction
9  => 0x400553 <main+45>:        addsd  xmm0,xmm1
10 (gdb) p $xmm0.v2_double
11 $1 = {0, 0} # content of xmm0
12 (gdb) p $xmm1.v2_double
13 $2 = {1, 0} # content of xmm1
14 (gdb) nexti
15 0x0000000000400557 in main ()
16 (gdb) p $xmm0.v2_double
17 $4 = {1, 0} # a[0] + b [0] = 1
18 (gdb) c
19 Continuing.
20 Breakpoint 1, 0x0000000000400553 in main ()
21 (gdb) p $xmm0.v2_double
22 $5 = {2, 0}
23 (gdb) p $xmm1.v2_double
24 $6 = {3, 0}
25 (gdb) nexti
26 0x0000000000400557 in main ()
27 (gdb) p $xmm0.v2_double
28 $7 = {5, 0} # a[1] + b[1] = 5
```

At the start the assembly language displayed by `gdb` is set to Intel (line 1). From the Listing generated with `objdump` we can see that the addition is performed at the address `0x400553`, thus we set a breakpoint at that address using the `break` instruction (4). The execution is started using the command `r` and stops as soon the breakpoint is hit. The command `xamine` (`x`) is used to inspect as an instruction (`i`) what's inside the *instruction pointer* register (`rip`) to make sure the execution stops at the right place (line 8). Using the command `print` (`p`) the content of `xmm0` and `xmm1` can be inspected (e.g., the operands of the addition) and interpreted as a pair of `double` numbers (`v2_double`). Since `xmm0` contains $\{0, 0\}$ (line 11) and `xmm1` contains $\{1, 0\}$ (line 13) it seems that only the first half of each register is

actually used. After advancing the execution of one instruction `nexti` we can then inspect again the content of `xmm0` containing the result of the sum as expected. With the command `continue` (`c`) the execution is resumed until the breakpoint is hit again. By inspecting once more the content of the `xmm` registers the behavior of the program should be clear: the first half of `xmm0` is used to store contiguous numbers from the array `b` while `xmm1` is half loaded with contiguous numbers from the array `a`. As expected, even if vector registers are used, no optimization is performed. In `gcc`, vector registers `xmm` should be used by compiling the code with the option `-O2` in conjunction with `-ftree-vectorize`. Let's have a look at the assembly code generated by activating this options.

Listing 2.5

```
1  > gcc -O2 -ftree-vectorize -o vecsum vectorization.c
2  > objdump -M intel -D vecsum --no-show-raw-insn \
3  | grep "<main>" -A10
4  0000000000400430 <main>:
5   400430:  push   rbx
6   400431:  xor    eax,eax
7   400433:  movapd xmm0,XMMWORD PTR [rax+0x601060] # loop-start
8   40043b:  add    rax,0x10
9   40043f:  addpd  xmm0,XMMWORD PTR [rax+0x601090] # SIMD sum
10  400447:  movaps XMMWORD PTR [rax+0x6010f0],xmm0
11  40044e:  cmp    rax,0x40
12  400452:  jne    400433 <main+0x3> # loop-end
13  400454:  xor    ebx,ebx
14  400456:  nop    WORD PTR cs:[rax+rax*1+0x0]
```

The assembly code contained in the Listing 2.5 looks different from the previous one (2.3). That's because by activating the optimizations, the program code gets modified by the compiler, resulting in a (different) more efficient assembly code. This time, the for loop spans between the addressed `0x400433` (line 7) and

`0x400452` (line 12) while the addition is carried out by `addpd` (i.e., *add packed doubles* in line 9) at `0x40043f`. The second operand is a memory region specified as a relative address which can be found by summing to the content of the register `rax` the offset `0x601090`. The keywords `XMMWORD PTR` indicate that the square brackets contain a pointer to a memory region which should be interpreted as a 128 bit register (`xmm`) containing (in our case) two contiguous double words. At this point we should examine the content of the operands to see if the optimization is actually performed.

Listing 2.6

```
1  > gdb -q -x gdb_config vecsum
2  Reading symbols from vecsum...done.
3  (gdb) break *0x40043f
4  Breakpoint 1 at 0x40043f
5  (gdb) r
6  Breakpoint 1, 0x000000000040043f in main ()
7  (gdb) p $xmm0.v2_double
8  $1 = {0, 2} # xmm0 contains b[0] and b[1]
9  (gdb) x/2gf $rax+0x601090
10 0x6010a0 <a>:  1  3 # a[0] and a[1] fetched from a
11 (gdb) nexti
12 0x0000000000400447 in main ()
13 (gdb) p $xmm0.v2_double
14 $2 = {1, 5} # a[0,1] + b[0,1] = {1,5}
15 (gdb) c
16 Continuing.
17 Breakpoint 1, 0x000000000040043f in main ()
18 (gdb) p $xmm0.v2_double
19 $3 = {4, 6}
20 (gdb) x/2gf $rax+0x601090
21 0x6010b0 <a+16>:  5  7
22 (gdb) nexti
23 0x0000000000400447 in main ()
```

```
24  (gdb) p $xmm0.v2_double
25  $4 = {9, 13}
```

This time `xmm0` is loaded with two doubles from the array `b` (i.e., 0 and 2 at line 8) while 1 and 3 are fetched directly from the array `a` (line 10). The options `2gf` passed to `xamine` tell `gdb` to print two *double words*, or *giants* (`g`), and to interpret them as floating point numbers `f` (a *word* is usually 32 bits long). The content of `xmm0` after the sum (line 14) contains the expected result (i.e., $\{1, 5\}$). At this point the rest of the code should be clear and we can conclude that the vector registers are actually used to speedup the computation as highlighted by the Figure 2.3. Notice that if we decide to use single precision floats instead of doubles, the `xmm` registers can be used to contain four operands instead of two. Even though there are a lot of more compile options and optimizations that can be tried and examined with `gdb` they won't be covered here as the main focus of this section is to give the main idea underlying the SIMD instructions (i.e., exploit special registers to perform the same operation over multiple data). For more advanced optimizations from Intel see AVX2 [47]and AVX-512 [48]which extend the register length to 256 and 512 bits respectively.

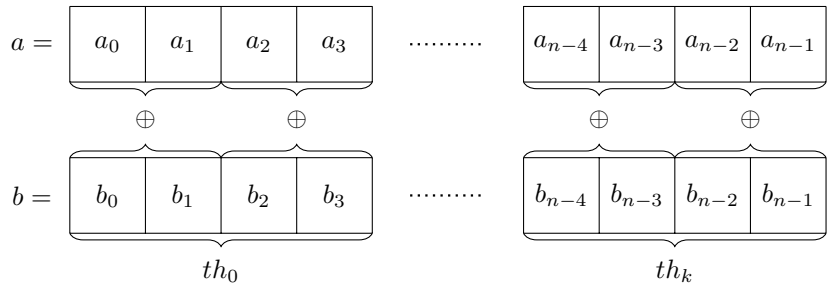
2.4.2 SMP with openMP

The SIMD optimization happens at the core level, meaning that each core has its separate set of registers, included the vector ones. Modern super-scalar CPUs are equipped with many cores. For example the Xeon-E5 2695v4 [49] has 18 cores, each one able to run SIMD instructions using vector registers. In normal conditions, a serial program in capable of exploiting a single core meaning that the other ones are computationally idle. One of the challenges of HPC is to write programs capable of fully exploit the capabilities of the underlying hardware, to solve computationally intensive problems. To do that, a good solution is to adopt

the *simultaneous multi-threading* (SMT), in which some parallelizable portions of the code are executed in parallel by a number of threads, each one associated to a single core.

Provided that an application performs some kind of operation a number of times, the idea of SMT is to distribute the workload among threads executing concurrently on different cores. It is up to the programmer deciding which operations each thread will carry out. However, in HPC applications, usually threads execute the same operations on different data (i.e., memory regions). Once located the region of the program to parallelize, the main execution thread, which can be called *master thread*, spawns a group of *child* threads, indicating the point of the code in which each thread starts its execution. This process of spawning threads, also known as the *fork* operation, can be hidden to the programmer by using the *openMP intrinsics*, consisting in a set of special lines of code, called *pragmas*, in which the programmer specifies the general requirements in order execute the underlying region of code using multi-threading. During the compilation process, the compiler turns the pragmas into the code to manage different threads. In this way it is much more easier for the programmer to write multi-threaded code for HPC applications.

Figure 2.4: Multi-threaded vector sum: different threads perform a vector sum between two aligned sub-vectors. Each thread exploits the vectorization.



Imagine to extend the example shown in Figure 2.3 using openMP where a sequence of SIMD operations is performed between the arrays a and b. In this case the length of such arrays is assumed to be $n = 4k$ for $k \geq 0$. Using a multithreaded approach it is possible to further parallelize this process by spawning a number of threads, each one operating on a different part of the input data (i.e., in this case a and b). The Figure 2.4 illustrates the idea: each thread th_i iterates the $\oplus$ operation between two aligned chunks of length $k = 4$ between the arrays a and b. In this way all the cores of a CPU can be efficiently exploited by spawning multiple threads, one for each core, and let them execute concurrently on a different part of the input data. As already mentioned, openMP simplifies the process of spawning threads thanks to the openMP pragmas, consisting in a number of special strings, informing the compiler on how to parallelize the underlying region of code. In the following part of this section the reader is provided with a simple example highlighting the multi-threading parallelism using OpenMP.

The code contained inside the Listing 2.7 extends the vector sum of the Listing 2.2 with OpenMP. The main idea is to split the input vectors `odd` and `even` in different (aligned) chunks and compute their sum in parallel using multiple threads. Since there are no data dependencies between different chunks, threads can work in parallel. The program expects as input the length of the vectors to be added and stores it in the variable `LEN`. The procedure `init_vectors` simply fills the arrays `odd` and `even` with the i-th odd and even number respectively, for $i = 0, \ldots,$`LEN`-1. After the allocation and the initialization (lines 23,24,25 and 27) the OpenMP pragma is inserted at line 30 right before the `for` statement computing the vector sum. The keywords `parallel for` indicate that the following loop is distributed across all available threads, provided that the vectors are long enough to guarantee at least one sum per thread. Otherwise only a subset of threads will be spawned. The `simd` keyword enables the SIMD parallelism inside each thread and `default(shared)` indicates that the program variables are shared among

active threads. The `#if` block surrounding the pragma means that the code is compiled only if the option `-fopenmp` is passed to `gcc`. The same block is used to wrap the `printf` call at line 35 for portability reasons: the code should be able to run even on those environments not having OpenMP. The `printf` statement is called by each thread, printing out the thread id, the total threads number and the current entry i of the vectors being added.

Listing 2.7

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  typedef double FP;
6  FP * odd, *even, *res;
7
8  void init_vectors(FP * odd, FP * even, int len ){
9  for(int i=0;i<len;i++){
10 odd[i]=2*i+1;
11 even[i]=2*i;
12 }
13 return;
14 }
15
16 int  main(int argc, char ** argv ){
17 int LEN=40;
18 if(argc==2){
19   LEN = atoi(argv[1]);
20 }
21
22 size_t bytes = LEN*sizeof(FP);
23 odd = (FP *) malloc(bytes);
24 even = (FP *) malloc(bytes);
25 res = (FP *) malloc(bytes);
```

```
26
27  init_vectors(odd,even,LEN);
28  // Compute odd + even and add an openMP intrinsics
29  #if defined (_OPENMP)
30  #pragma omp parallel for simd default(shared)
31  #endif
32  for(int i=0; i<LEN; i=i+1){
33      res[i] = odd[i] + even[i];
34      #if defined (_OPENMP)
35      printf("Thread %d out of %d. Computing Index %d\n",
36      omp_get_thread_num(), omp_get_num_threads()-1,i);
37      #endif
38  }
39
40  for(int i=0; i<LEN; i=i+1){
41    printf("res_%d is %f\n", i,res[i]);
42  }
43  return 0;
44  }
```

As shown in the Listing 2.8, the example is run on an Intel `Intel(R) Xeon(R) E5-2650 v3` CPU having 19 cores, each one managing one thread. The code is compiled in a binary called `omp_sum` and run on vectors of length 100 (lines 10 and 16).

Listing 2.8

```
1  > lscpu | grep 'On-line\|Thread(s)\|Model name'
2  On-line CPU(s) list:   0-19
3  Thread(s) per core:    1
4  Model name:   Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
5  > gcc -Ofast -march=native -ftree-vectorize -fopt-info-vec
6  -fopenmp -o omp_sum openmp_vec.c
7  openmp_vec.c:9:1: note: loop vectorized
8  # omitting full output ..
```

```
# the printf prevents OpenMP loops from being vectorized
> ./omp_sum 100 | grep "Thread 0" # Filters the thread 0
Thread 0 out of 19. Computing Index 0
Thread 0 out of 19. Computing Index 1
Thread 0 out of 19. Computing Index 2
Thread 0 out of 19. Computing Index 3
Thread 0 out of 19. Computing Index 4
> ./omp_sum 100 | grep "Thread 19" # Filters the thread 19
Thread 19 out of 19. Computing Index 95
Thread 19 out of 19. Computing Index 96
Thread 19 out of 19. Computing Index 97
Thread 19 out of 19. Computing Index 98
Thread 19 out of 19. Computing Index 99
> # Commenting out the #if block in openmp_vec.c
> gcc -Ofast -march=native -ftree-vectorize
-fopt-info-vec  -fopenmp -o omp_sum openmp_vec.c
openmp_vec.c:33:12: note: loop vectorized
# omitting the full output ..
```

During a multithreaded execution it is important to prevent a thread to read data from a region of memory which another thread modifies. This phenomenon is known as *race condition* and can arise in a number of different ways [50].

In general the compiler is not able to detect the race conditions in a multithreaded portion of code. It is up to the programmer to identify the specific region of code containing a race condition (i.e., a *critical* region) and deal with it. Some pragmas exists to avoid the race conditions such as *omp critical* or *omp atomic* which avoid the race conditions by serializing the code, with a negative impact on the performance. In some cases, it is useful to use an openMP *reduction* pragma, enabling each thread to accumulate a partial result of a loop computation in a private variable for each thread and then computing the final value as a global reduction of the private variables.

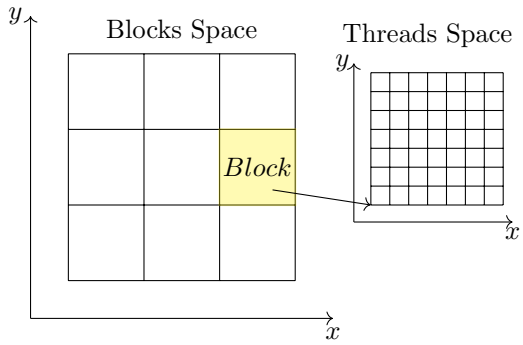
2.5 SIMT with NVIDIA GPUs

So far two types of parallelism exploited by the CPU have been described. The first one is the SIMD parallelism, which allows to perform an operation between vector registers using a single assembly instruction. The second level of parallelism is SMT which allows multiple threads to execute concurrently on different cores. This section briefly introduces the SIMT parallelism implemented by NVIDIA GPUs, which can be located somewhere in between SIMD and SMT as it shares its differences and similarities with both these models.

In particular SIMT is akin to SIMD in that a single instruction is issued to multiple processing elements but while in the SIMD model there is a thread executing on a CPU core having its private set of vector registers, in the SIMT model there are multiple threads, each one executing the same instruction on a different core. It follows that SIMT is similar to SMP in that multiple threads execute concurrently but while CPU threads are very few even in modern superscalar CPUs, NVIDIA GPUS can easily manage tenths of thousands threads. Depending on the hardware, a considerable subset of such threads can be executed in parallel.

The main difference between SMP on CPUs and SIMT on GPUs is that CPUs privilege the execution speed of a single thread by minimizing the idle times. To do that CPUs exploit a complex cache hierarchy together with advanced techniques such as *branch-prediction*, *out-of-order execution* and *speculative execution* with the goal of keeping the cores busy. On the other side GPUs try to maximize the throughput by managing a larger (compared to CPUs) number of threads, hiding the latencies by constantly switching between groups of threads (called *warps*) executing concurrently. Summarizing, GPUs support the parallel execution of a large set of slow threads while superscalar CPUs privilege the parallel execution of few very fast threads.

Figure 2.5: 2D representation of the CUDA block and thread spaces. The same organization can be generalized to a $3D$ space.

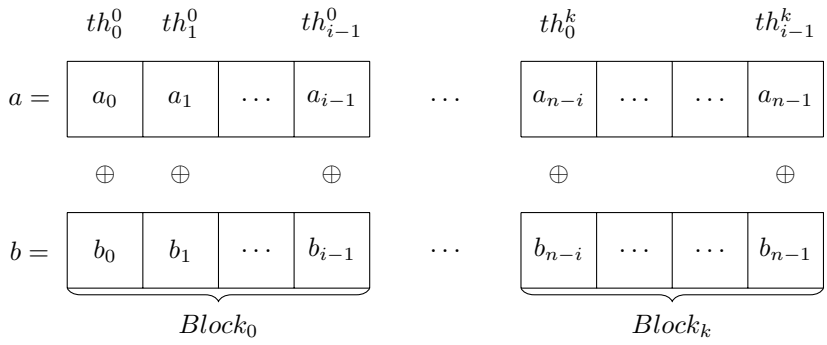


From a software point of view the programmer writes SIMT codes using a programming language called CUDA (supported by NVIDIA GPUs) [29]. In such language it is possible to specify the thread code thanks to a series of builtin constants which are used to control the execution path of single threads. In particular each thread belongs to a group of threads called *block*. Inside a block each thread can be identified by a three-dimensional coordinate system using the builtin values *threadIdx.x, threadIdx.y* and *threadIdx.z*.

When the programmer launches a GPU *kernel* (i.e., the name of a procedure executing on a GPU) specifies the number of blocks and the blocks size, which is the same for all blocks. To distinguish between threads belonging to different blocks the CUDA model defines three additional builtins: *blockIdx.x, blockIdx.y* and *blockIdx.z*. It follows that the set of all threads executing the code of a given kernel can be represented by a $3D$ Cartesian grid having two different levels of subdivision. In the first level there is a grid of blocks whilst in the second level, each block contains a grid of threads. This concept is highlighted in Figure 2.5 in two dimensions for ease of representation.

From an hardware point of view, GPUs feature a series of superscalar multipro-

Figure 2.6: Vector sum using the SIMT parallelism defined by the CUDA programming model.



cessors called *streaming multiprocessors* (SMs) each one having the same number of cores which depends on the specific GPU model. Inside each SM, threads execute in groups of size 32 called *warps* and the threads contained in each warp belong to the same block. The threads in the same warp execute the same instructions in parallel, thus a GPU code can be considered efficient if there are a lot of threads sharing the same execution path. Divergences (i.e., code branches) between threads inside the same warp are solved by serializing the execution between the groups of threads having a different execution path. For this reason a GPU code having a lot of branches is generally not recommended [51].

CPU and GPU can work asynchronously, provided that there is no logical dependence between their respective computations, otherwise specific synchronization calls can be used to force the CPU to wait for a kernel termination. For analogy with the previous Sections 2.4.1 and 2.4.2, let's see how a simple operation $\oplus$ can be performed between two vectors a and b of the same length, using the SIMT model. The main idea is to ideally split a and b into contiguous chunks of the same size and associate to each chunk a separate block. The Figure 2.6 should clarify the concept. The notation th_r^t is used to indicate that the thread of coor-

dinate r belongs to the block t. Since vectors are mono-dimensional structures, a single index can be used to identify the threads inside each block, and the blocks space is mono-dimensional as well.

To express the sum between a and b (Figure 2.6) in a SIMT fashion, the CUDA programming language adopts a *scalar* notation which expresses the execution path of a single thread. To this end it is necessary to define a mapping from the blocks and the thread spaces into $\mathbb{N}$ (i.e., a single integer) which can be used as a global offset inside the GPU memory. In our example, a intuitive way to do it, is to assign the thread r belonging to the block t to the memory cell located at the index $idx = i \cdot t + r$, where i denotes the block length in the Figure 2.6. More in general, each thread computes idx as a function of the *blockIdx* and *threadIdx* builtins defined earlier in this section. The block size is defined by the programmer before launching the GPU kernel. At this point each thread can compute $a[idx] \oplus b[idx]$.

The following part of this section provides the reader with an example implementing a vector sum using the SIMT parallelism. The code contained inside the Listing 2.10 implements the sum of two vectors a and b, containing the first n even and odd numbers, respectively. The sum is implemented exactly as highlighted in Figure 2.6 with the difference that this time n can be any (positive) integer number. As usual the program starts at line 23, where the main function begins. Between lines 25 and 30 the program checks that the user correctly inputs a single positive integer to the program. Such integer will be used to determine the size of the arrays a and b, which are then allocated and initialized together with an array c containing the final results (lines 33 to 37). These allocations and initializations are performed on CPU, but to being able to compute the sum on GPU it is necessary to allocate and initialize the GPU counterparts of a, b and c (i.e., da, db and dc). The allocation is carried out between lines 41 and 43 thanks to `cudaMalloc`. The third parameter of such call specifies the direction of the copy

(in this case from CPU *host* to GPU *device*). The initialization simply consists in copying the content of a, b and c to da, db and dc using the `cudaMemcpy` function.

The next step consists in setting up the kernel launch parameters by computing the size of each block and the number of blocks. For simplicity, the block size is fixed to the warp size, which usually is 32 and is stored in the variable `WARP`. The number of blocks to spawn is computed in line 48 as $bx = \lceil n/32 \rceil$ to ensure that at least one block is spawned, even for $n < 32$. The following line prepares the two launch parameters, having type `dim3` and containing the grid dimension $(bx, 1, 1)$ and the block dimension $(32, 1, 1)$ respectively. Al line 51 the kernel `simtSum` is launched on the vectors da, db and dc. The kernel code is defined between lines 6 and 13. Since the kernel code is executed by each thread it is necessary to map the thread execution on the right memory region. To this end we need to compute the memory offset of each thread as shown in Figure 2.6; in particular each thread firstly determines the chunk of memory of competence of the blocks its belongs to. This is done by multiplying the block dimension (in this case 32), stored into the predefined variable `blockDim.x`, for the block index stored into `blockIdx.x`. Finally it is sufficient to add to that offset the thread coordinate stored into the predefined variable `threadIdx.x`. Since threads may outnumber the memory cells we need to prevent threads having and offset greater than $n - 1$ from accessing the memory by terminating their execution (line 11) thus avoiding segmentation faults. The computation of $a + b$ is carried out in the following line as expected. The result vector is copied from the GPU memory dc to the CPU one (c) using another call to `cudaMemcpy`.

Listing 2.9

```
1  #include <stdio.h>
2  typedef float FP;
3  const int WARP = 32;
4
```

```
5  __global__
6  void simtSum(FP * a, FP * b, FP * c,int n){
7      int k;
8      /* Map a thread into a memory index */
9      offset = blockDim.x*blockIdx.x + threadIdx.x;
10     /* Discard threads without a memory counterpart */
11     if(k>n-1) return;
12     c[offset] = a[offset] + b[offset]; // a + b
13 }
14
15 void init_vec(FP * a, FP * b, FP * c, int n){
16   for(int i=0; i<n; i++){
17     a[i] = 2*i;
18     b[i] = 2*i+1;
19     c[i] = 0;
20   }
21 }
22
23 int main(int argc, char ** argv){
24 /* Input check*/
25 if(argc != 2){
26   printf("Wrong number of args. Expected one integer\n");
27   exit(0);
28 }
29 int n = atoi(argv[1]);
30 if(n==0) return 1;
31
32 /* CPU allocations */
33 size_t bytes = n*sizeof(FP);
34 FP * a = (FP *) malloc( bytes );
35 FP * b = (FP *) malloc( bytes );
36 FP * c = (FP *) malloc( bytes );
37 init_vec(a,b,c,n);
38
```

```
39  FP *da, *db, *dc;
40  /* GPU allocations */
41  cudaMalloc((void **)&da, bytes);
42  cudaMalloc((void **)&db, bytes);
43  cudaMalloc((void **)&dc, bytes);
44  cudaMemcpy(da,a,bytes,cudaMemcpyHostToDevice);
45  cudaMemcpy(db,b,bytes,cudaMemcpyHostToDevice);
46  cudaMemcpy(dc,c,bytes,cudaMemcpyHostToDevice);
47  /* Compute the number of blocks and threads needed */
48  int bx=ceil( ((float) n)/ ((float)WARP) );
49  dim3 threads(WARP,1,1), blocks(bx,1,1);
50  /* Kernel launch */
51  simtSum<<<blocks,threads>>>(da,db,dc,n);
52  cudaMemcpy(c,dc,bytes,cudaMemcpyDeviceToHost);
53  /* Print the result */
54  for(int i= 0; i<n; i++){
55    printf("c[%d]=%f\n", i,c[i]);
56  }
57
58  return 1;
59  }
```

To compile and run the code it is necessary to install the cuda-toolkit being in possession of a CUDA capable device.

Listing 2.10

```
> lspci | grep -i nvidia
01:00.0 3D controller: NVIDIA Corporation GM107M
[GeForce GTX 960M] (rev a2)
> nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Fri_Feb__8_19:08:17_PST_2019
Cuda compilation tools, release 10.1, V10.1.105
> nvcc -o simtSum simt.cu
```

```
> ./simtSum
Wrong number of args. Expected one integer
> ./simtSum 10
c[0]=1.000000
c[1]=5.000000
c[2]=9.000000
c[3]=13.000000
c[4]=17.000000
c[5]=21.000000
c[6]=25.000000
c[7]=29.000000
c[8]=33.000000
```

Summarizing, in this section described the general structure of a SIMT program using the CUDA programming model. It is important to keep in mind that a GPU is a CPU co-processor well suited to solve tasks having high arithmetic intensity. In the Listing 2.10 the sum of two vectors is implemented, using the CUDA programming model which is a language used to write programs executed in a SIMT fashion. The portions of code offloaded by the CPU to a GPU are called kernels, and contain the code executed concurrently by a number of threads grouped in blocks. The number of blocks and the size of each block is specified by the kernel launch parameters. It is up to the programmer the task of defining a mapping between the thread and blocks coordinates to the memory space and even though there are multiple ways to do it, a good practice is to coalesce the memory accesses by making sure that contiguous (in term of coordinates) threads access contiguous (in term of offsets) memory cells. This practice is known as *coalesced memory accesses* and aims to minimize the number of memory transactions.

2.5.1 A quick comparison between CPUs and GPUs

Nowadays GPUs seem to be more suitable than CPU to solve problems where the same operation is repeated number of time over a large amount of data. CPUs deal

with such problems using few very fast and architecturally complex cores, each one using SIMD parallelism and exploiting various techniques such as prefetching, out-of-order execution, branch prediction and a complex hierarchy of caches to hide memory latencies. On the other hand GPUs use many simpler cores grouped inside streaming multiprocessors executing in a SIMT fashion. GPU cores are much slower than the CPU ones, and the memory latencies higher. However, GPUs are able to hide this latencies by constantly switching between warps to keep the cuda cores busy. For this reason, an application can achieve a greater performance by using the GPU only if there is a sufficient amount of data which can be efficiently processed using SIMT. A comparison between CPU and GPU has been carried out in [52] where a number of benchmarks have been run on both architecture with a great level of optimization. According to such study, the exploitation of GPUs to achieve an order of $100x$ speedups is not feasible when the benchmarks are highly optimized both on CPU and GPU. Instead, the study shows how the speedups obtained by using GPUs are of $2,5\times$ on average. Some benchmarks benefit from a speedup around $10\times$ such as SAXPY, LBM and GJK.

The calculation of the *theoretical peak performance* for two state of the art devices can be useful to give an idea of the difference between CPU and GPUs in term of *Floating Point Operations Per Second* FLOPS. Consider as an example an Intel CPU Xeon-E5 2695v4 [49], which contains 18 cores and is able to work at a maximum clock speed of 3,3GHz (i.e., $3,3 \cdot 10^9$ cycles per second in Turbo-Boost mode). Such CPU supports the *Advanced Vector Extension* (AVX2) meaning that it contains special registers of 256 bits (32 Bytes) in which 4 double precision or 8 single precision floating point numbers can be stored. With this extension the CPU is able to perform an operation on the AVX registers with a single instruction. According to Intel [53]the ideal maximum number of *Instructions Per Cycle* (IPC) is 4, however for a real HPC application, an average IPC of 1 is considered acceptable. Since in a HPC application the number of floating point

instructions should be much more higher than any other instructions in the code, the IPC can be approximated with the actual number of floating point operations per cycle. It follows that the theoretical peak performance of the Intel Xeon-E5 2695v4 is

$$\text{IPC}[\tfrac{FLO}{Cycle}] \times \text{ClockSpeed } [Hz = \tfrac{Cycle}{s}] \times \text{Cores} \times \text{AVX-Opt} =$$
$$1 \times 3,3 \times 10^9 \times 18 \times 4 = 237,6 \times 10^9 \ [\tfrac{FLO}{s}] = 237,6[\text{GFLOPS}]$$

Instead, considering an IPC of 4 the result is $950,4$ GFLOPS. An NVIDIA P100 GPU has a theoretical peak performance of $5,3$ Tera FLOPS [54], thus much more faster, in term of FLOPS, than the Intel CPU detailed above. Furthermore the Intel Xeon-E5 2695v4 has a power consumption of $120W$ whereas a the P100 consumes $300W$, meaning that it can (theoretically) work between $5,57$ and $22,3$ times faster than the CPU, depending on the IPC being resp. 1 or 4, consuming $2,5$ times as much. In conclusion GPUs seem to be the best choice for executing tasks with *high arithmetic intensity* (i.e. the ratio between arithmetic operations and memory operations) [55].

2.6 SPMD with MPI

The above sections presented the main approaches which can be taken to parallelize an HPC application. A CPU based approach takes advantage of a data level parallelism using SIMD instructions; a thread level parallelism exploits multiple cores, each one executing a dedicated thread. A GPU based approach uses the SIMT parallelism which is realized by groups of threads executing the same instructions (warps) over an array of streaming multi processors. An hybrid approach is also common in the HPC environment and consists in distributing the workload and consequently the data among CPUs and GPUs and make them work asynchronously where possible, performing synchronizations where needed.

Since these approaches take place inside a compute node and are controlled by a single process, now the question is how to deal with increasingly large inputs, or similarly, how to further speedup the computation. On a single compute node, the amount of main memory (both on CPU and GPU) is limited and consequently there is an upper limit to the size of the input. Similarly, the number of superscalar units between CPUs and GPUs is limited and once their cores are completely saturated by the computation, the application won't benefit from any further speedup.

As mentioned earlier in Section 2.1 a supercomputer consists in a set of compute nodes working together as a single system. The outcome of this design is the possibility to run a program on multiple nodes as if they were a single node taking advantage of an extended set of computing resources. In the field of HPC the common way to implement this additional level of parallelism is represented by *multiprocessing* in the form of *Single Program Multiple Data* (SPMD); in this model, multiple instances of the same program (i.e., processes) run in parallel on different portions of the input domain. The common design of HPC applications requires that the input data is stored inside a file (or a set of files) located in the storage system and multiple processes read different portions of the input domain in a parallel fashion and store their respective data on the main memory of a compute node. In the same way as the input operations, the output ones are usually performed in parallel by each process. It is fair to assume that each process has exclusive access to a predefined set of compute resources, otherwise the overall execution time of the program wont benefit of great speedups.

Summarizing, the input data of an HPC application is split among different processes executing an instance of the same program and exploiting dedicated hardware resources located in a network of compute nodes (HPC cluster). The number of processes which can be hosted on a single compute node depends on the hardware of the node and on the application design; for example if an application

is able to exploit simultaneous multi-threading (see SMT in Section 2.4.2) it can efficiently occupy a whole CPU for its execution. Thus if the node is equipped with just one CPU then a possible resource assignment (recall the job scheduler in Section 2.1.2) can be one CPU per process. The situation can change if multiple CPUs or GPUs are located on the same node.

A crucial aspect of multiprocessing is the communication between processes. The content, the frequency of the communications and the number of processes involved, depend on the application domain; in general it is safe to assume that all HPC applications implement some form of process communication.

A powerful way to accomplish this task is to take advantage of the *Message Passing Interface* (MPI) [23], a standard defining a set of high-level library calls to let different processes communicate over a network. From the point of view of the programmer, implementing inter-process communication comes down to use a set of library procedures, without having to deal with low level network programming, thus making the HPC development much easier.

An abstraction to SPMD is represented by the *Multiple Programs Multiple Data* (MPMD) model in which MPI is able to run multiple instances of different programs (refer to [56] for an example form Intel). This thesis considers only those applications where CPUs (and co-processors) work together to solve a complex problem by executing the same program on a different portion of the input data (i.e., in a SPMD fashion).

Since a detailed description of MPI is out of the scope of this thesis, the following part of this section provides the reader with an example describing the overall structure of an MPI program. In particular, the vector sum of the example 2.2 is extended by making it scalable using MPI. The example shown inside the Listing 2.11 computes the sum of two *arbitrary long* vectors and stores the result in a file. To make the code scalable both in *space* and *time* the sum is performed

by an arbitrary number of MPI processes, each one computing the sum of two sub-vectors obtained by partitioning the global input vectors (i.e., *the global domain*).

Since the code is scalable in *space* we cannot assume that each process has enough memory to store the global input vectors on the main memory of the node, meaning that each MPI process allocates just the space to store the two input sub-vectors and the one containing the result. To make the code as simple as possible, the global input vectors are assumed to contain the first `LEN` *even* and *odd* numbers respectively, where `LEN` is an input integer. In this way each MPI process generates its own input sub-vectors without having to read them from file.

At the end of the program all processes write their output into a shared file, at a certain (computed) offset, in a parallel fashion. From line 5 to line 12 the definition of floating point number `FP` is carried out, together with its MPI counterpart `MPI_FP`, creating an abstraction on the type allowing to switch from double precision to single precision without modifying the program body. The following two lines of code contain the prototype definitions for the function `sum` and `init_vectors`.

All of these functions are executed by each MPI process. The first is defined at line 71 a serial (naive) addition of the local sub-vectors `odd` and `even`, storing result in `res`. The `step` parameter is the length of such vectors. The second procedure is defined at line 78 and its purpose is to populate `odd` and `even`. In line 19 the length of the global input vectors is set to 8 if the user does not provide an input parameter. Right after that, the MPI environment is initialized by calling `MPI_Init`. `MPI_Comm_rank` assign to each process a unique identifier of integer type and stores it into the `rank` variable. `MPI_Comm_size` stores into the variable `nProcs` the number of processes belonging to the global communicator `MPI_COMM_WORLD`. A communicator identifies a group of processes which can communicate through MPI calls, and `MPI_COMM_WORLD` identifies the group containing all MPI processes spawned at the start of the program. The following `if` statement (line 29) ensures

that the number of MPI processes does not exceed the length of the global vectors, otherwise some MPI processes would be left out from the computation.

The length of the local vector associated to each rank is stored inside `step` which initially is set to `LEN/nProcs`. Since `LEN` is arbitrary, the first `r` processes will manage vectors of one unit longer, where r is the remainder of `LEN/nProcs`. For this reason `step` is incremented in line 40 for processes having `rank` $<$ `r`, keeping in mind that `rank` spans from zero to `nProcs`-1. The `start_index` variable contains the offset inside the (hypothetical) global input vector. Between lines 49 and 52 The three local vectors (i.e., `odd`, `even` and `res`) are allocated to contain just the space they need (i.e., `bytes` bytes for each vector) to perform their local computation. Allocating the space to contain the global domain would limit the scalability of the code to the amount of memory present on the compute nodes in which the processes are hosted.

Instead, by allocating just the required portion of the input, the scalability is limited by the amount of nodes available. The procedure `init_vectors` called at line 54 and defined at line 78 populates the `odd` (resp. `even`) arrays by assigning to `odd[i]` (resp. `even[i]`) the `(start_index+i)`-th odd (resp. even) number, for $i = 0, \ldots,$`step`-1. At this point the preprocessing is over and the sum can be computed locally by each MPI process by calling the `sum` procedure at line 55. Notice that such procedure (defined at line 71) is a serial implementation and can be further optimized with SIMD 2.4.1, openMP 2.4.2 and SIMT 2.5.

The last task of the code is to store the result in a shared file which is called `mpi-sum.out`. Such file is created using a call to `MPI_File_open`, passing the global communicator as first parameter and the file name as second parameter. The third parameter is a combination (logical or) of two elements: `MPI_MODE_CREATE` and `MPI_MODE_RDWR`. The former triggers the creation of the file if does not exists and the latter opens the file in read and write mode. The fourth parameter is set `MPI_INFO_NULL` and contains no information. Usually it is used to set additional

options and their description is out of the scope of this example. Finally, the fourth parameter contain the file handle, used to identify `mpi-sum.out` in the following MPI calls.

Before writing the output on the file, `MPI_File_preallocate` is called in line 62 to tell MPI how much space is (globally) needed for the file, which is commonly placed in a shared storage (see Section 2.1) space on the HPC system in use. Even if the file is shared, each process can safely perform the writings without the risk of incurring in race conditions because each process writes in a different area of the file. Such area is computed by calling `MPI_File_seek`, which takes an offset in bytes as second argument, and sets the third parameter to `MPI_SEEK_SET` to tell MPI that the writings are performed starting from such offset. Since the data-type used in this example is defined by `FP`, the offset in bytes is set to `start_index` $\times$ `sizeof(FP)`. Finally the actual writings are performed at line 64 by `MPI_File_write`. The first argument is the file handle (`fh`), the second is the array `res` containing the sub-vectors sum, the third one is the length of `res` (i.e., `step`), the fourth one indicates the MPI data type of the vector elements (i.e., `MPI_FP`), which is can be `MPI_DOUBLE` or `MPI_FLOAT` depending on the precision adopted.

The important aspect is that `MPI_FP` must be of the same size as `FP` to ensure that MPI communications exchange the right amount of data. The last parameter contains information on the result of the call and it will be ignored. After the writings are finished `MPI_File_close` is called to close the file and `MPI_Finalize` to terminate the MPI execution environment.

Notice the difference between the MPI calls performed to open, allocate and close the file (i.e., lines 60, 62 and 66) and the ones to set the offset and perform the writings (i.e., lines 63 and 64). The former group of procedures are called *collective*, meaning that the procedure requires all processes to execute it together (at the same time). The latter group of calls are *non collective* and can be performed

separately by each MPI processes without performing any synchronization. It follows that an optimized HPC application tries to perform as less collective calls as possible, because some processes may spend idle times waiting for all other processes to reach that point in the code.

Listing 2.11

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define double_prec
#ifdef double_prec
typedef double FP;
#define MPI_FP MPI_DOUBLE
#else
typedef float FP;
#define MPI_FP MPI_FLOAT
#endif

void sum(FP * odd, FP * even, FP * c, int step);
void init_vectors(FP * odd, FP * even, int start_idx, int len);

int main(int argc, char ** argv){

  int LEN=8;
  if(argc == 2){
    LEN = atoi(argv[1]);
  }

  int rank, nProcs;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nProcs);

```

```
29    if( nProcs > LEN ){
30      if(rank==0)
31        printf("The maximum number of processes is %d.\n",LEN);
32      mpi_finalize(rank);
33      return 0;
34    }
35
36    int step = LEN/nProcs;
37    int r = LEN%nProcs;
38    int start_index;
39    if( rank < r ){
40      step++;
41      start_index = step*rank;
42    }
43    else{
44      start_index = (step+1)*r + step*(rank-r);
45    }
46    printf("[Rank %d] starting at %d, with step %d\n",
47      rank, start_index,step);
48
49    size_t bytes = step * sizeof(FP);
50    FP * odd = (FP *) malloc( bytes );
51    FP * even = (FP *) malloc( bytes );
52    FP * res = (FP *) malloc( bytes );
53
54    init_vectors(odd,even,start_index,step);
55    sum(odd,even,res,step);
56
57    MPI_File fh;
58    MPI_Status s;
59
60    MPI_File_open(MPI_COMM_WORLD, "mpi-sum.out",
61      MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
62    MPI_File_preallocate(fh, LEN*sizeof(FP));
```

```
63    MPI_File_seek(fh, start_index*sizeof(FP), MPI_SEEK_SET);
64    MPI_File_write(fh, res,step, MPI_FP, &s);
65
66    MPI_File_close( &fh);
67    MPI_Finalize();
68    return 0;
69  }
70
71  void sum(FP * odd, FP * even, FP * res, int step){
72    for(int i=0; i<step; i++){
73      res[ i ] = odd[ i ] + even[ i ];
74    }
75    return;
76  }
77
78  void init_vectors(FP * odd, FP * even, int start_idx, int len){
79    for(int i=0; i<len; i++){
80      even[i] = 2*(start_idx+i);
81      odd[i] = 2*(start_idx+i)+1;
82    }
83    return;
84  }
```

The code contained inside the Listing 2.11 was compiled and run using Open-MPI [22], an open source implementation of the MPI standard. The code is compiled with `mpicc`, which is the MPI compiler for the C language based on `gcc`. The code is run using the program `mpirun`, which spawns the number of processes specified using the option `-n` (i.e., four in the example). Each MPI process runs the executable passed as next argument which in our case is `mpisum` and takes the length of the global domain as input (i.e., 30). As the output shows, each process (or rank) computes the sum of a different sub-vector. For example, the rank 2 computes the sum of the i-th odd and even numbers, for $i = 16, \ldots, 22$. The results are written in a parallel fashion inside a file called `mpi-sum.out`. Since the

file has a binary format, we can write our own parser to read it, or we can just use a Linux tool such as `od` [57] which in this case is enough to dump our output on the screen. The `-Ad` option tells `od` to print the byte addresses of the file using a decimal format; the option `-tfD` informs `od` on the type of the elements contained in our file which is floating point (`f`) double (`D`). Each row in the output contains two double precision floating point numbers (8 bytes each) and the result meet the expectations.

```
$ mpicc -o mpisum mpi_vector_sum.c
$ mpirun -n 4 mpisum 30
[Rank 2] starting at 16, with step 7
[Rank 3] starting at 23, with step 7
[Rank 0] starting at 0, with step 8
[Rank 1] starting at 8, with step 8
$ od -Ad -tfD mpi-sum.out
0000000                       1                       5
0000016                       9                      13
0000032                      17                      21
0000048                      25                      29
0000064                      33                      37
0000080                      41                      45
0000096                      49                      53
0000112                      57                      61
0000128                      65                      69
0000144                      73                      77
0000160                      81                      85
0000176                      89                      93
0000192                      97                     101
0000208                     105                     109
0000224                     113                     117
```

3

Multi-GPU PARFLOOD

This chapter presents a multi-GPU implementation of a Finite Volume solver named PARFLOOD. The work stems from the context of fluid dynamics [17], [18], [58], [59], where the application requires to model and simulate a flooding phenomenon, in order to deliver accurate and much faster than physical time simulations. Such predictions are used to guide the adoption of real-time counter-measures and/or assess the risks in a structural planning scenario. PARFLOOD is designed for a cluster architecture composed of a set of GPU nodes interconnected by a fast network.

As outlined in Section 1.4, the motivation behind this work is to overcome the scalability limitations of the single GPU implementation of the solver [18]. To this end, a multi-GPU version of the code has been developed following the well known approaches in the field of High Performance Computing (HPC): (i) domain partitioning, (ii) border communication and (iii) dynamic load balancing.

Despite the application being domain specific, the methodology discussed can be ported to a remarkable set of applications that solve a system of PDEs on a multi-resolution grid. The problem of distributing the load across several GPU devices is a rather recent challenge, given the de-facto evolution of HPC clusters

of massively parallel devices [24]. The reader should refer to the Chapter 2 to get an insight on the HPC environment together with an overall description of the common types of parallel architectures.

Before describing the multi-GPU design of the solver, Section 3.1 summarizes some important related work in the field of GPU and multi-GPU computing and Sections 3.2, 3.3 and 3.4 cover the key elements of the single GPU implementation of PARFLOOD. After discussing the limitations of such version of the solver (Section 3.5), the following sections address the goals of this thesis by describing how to distribute the workload of a simulation on multiple GPUs. In particular, Section 3.6 addresses the the *border communication* problem, defined in Section 1.4.2. Section 3.8 addresses the *domain partitioning* problem defined in Section 1.4.1. Finally, Section 3.7 describes the multi-GPU algorithm adopted by PARFLOOD and Section 3.9 concludes the chapter addressing the *load balancing problem* defined in Section 1.4.3. The work described in this chapter and Chapter 4 has been submitted to *IEEE Transactions on Parallel and Distributed Systems* and it is currently under revision.

3.1 Related Work

Different multi-GPU implementations based on Cartesian grid exist in literature. In [60] and [61] a 3D finite difference computation is implemented on four GPUs exploiting a single compute node. In both papers the domain is partitioned along a single dimension and each partition is processed by a single GPU. While in [60] MPI implements border communication between adjacent partitions, in [61] the software is based on OpenMP. An MPI-based approach, on a larger scale, supports the work of [62]: the software implements incompressible flow computations on a 3D Cartesian grid and it scales up to 128 GPUs. It partitions the domain along one dimension and it associates each sub-domain to a single GPU (managed by

a dedicated MPI process). The cited work share a 3D Cartesian grid as domain model: this choice offers excellent performances on GPU, thanks to the regular topology and memory accesses, with the downside of being space-consuming.

A multi-resolution grid allows to selectively reduce the amount of data, while preserving adequate numerical accuracy where needed [18]. The implementation on GPU of such ideas is feasible, with an expected and small degradation of absolute performances (i.e., there is no spatial locality between logical cells in memory), while the required amount of data can be reduced by one or two orders of magnitude.

The idea of using non Cartesian data structures showed some limitations on GPU and multi-GPU architectures. In [63] a single GPU implementation of a shallow water equation (SWE) solver using a triangle mesh is proposed. As a result a CPU-optimized version of the code with eight OpenMP processes obtains a performance similar to the single GPU implementation. A multi-GPU SWE solver based on a triangle mesh is proposed in [64]. Each MPI process manages a single GPU, which is in charge of simulating a domain partition. The code exhibits a linear speedup up to four GPUs. Both [63] and [64] report the difficulties of implementing an unstructured grid on GPU. Furthermore, since [64] exploits multiple GPUs, it deals with the problem of creating additional data structures for managing the border communication between neighbor domains.

In [65] a spectral-elements-method (SEM) for seismic wave propagation is proposed. The grid exploited is a 3D hexahedra mesh. Each sub-domain is associated to a single GPU and multiple sub-domains communicate using MPI. The study presents only a weak scalability test with a constant performance. The authors also state that the overhead introduced by the multi-GPU implementation are of 11% which is in line with the results obtained in this work (see Chapter 4).

The present study differs from [63, 64, 65], mainly because it designs a grid as the optimal trade-off between Cartesian and multi resolution grid (i.e. quad-tree

like). Furthermore the dynamic load balancing problem is addressed, with the goal of equally distribute the computational load among different MPI processes.

Load balancing on multi-GPU has been investigated in [42] and [66], with applications to the field of particle physics. In the first one a multi-GPU implementation of a smoothed particle hydrodynamics (SPH) solver is presented. It dynamically adjusts the partitions size, based on two different metrics: the former balances the number of particles among partitions, the latter balances the time-step durations. In both cases the computational load is varied by particles migration from one partition to another. A different approach is taken in [66], where the transport of particles in fluids is modelled. The computational load is balanced by measuring only the idle time for processes on the same node. When an MPI process is idle, it offers its computational resources to other processes in need. Even though the application domains and the data model are different, the approach adopted in PARFLOOD relates to both [42] and [66]. Similarly to [42], PARFLOOD implements data migration among MPI processes, but, instead of migrating particles, it migrates portions of grid (i.e. blocks of computation). The metric used to evaluate the computational load is based on idle time, as in [66], but in PARFLOOD the load is globally balanced among all MPI processes.

3.2 A single GPU design

The starting point of this thesis, is a single GPU implementation of PARFLOOD [19] [17] [18] [16]; this means that a great amount of work has been previously carried out, resulting in a fully functioning *shallow water equations* solver, offloading the computational part of a simulation to a (single) *graphic processing unit* (GPU). In the last decade, scientific parallel software experienced a boost provided by the number of HPC systems equipped with NVIDIA GPUs. The programming

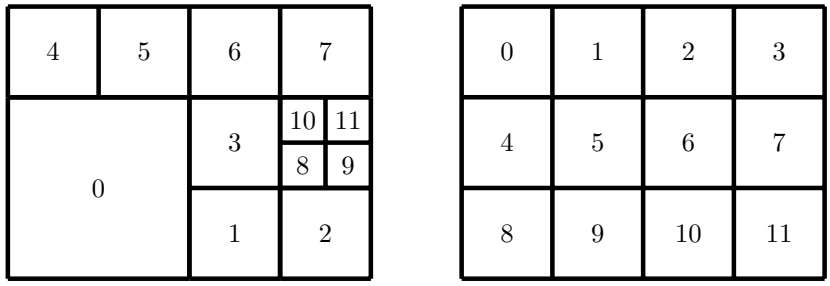
paradigm offered by CUDA architecture requires that algorithms and data structures are designed from scratch in order to fully exploit the underlying hardware capabilities. A common approach tends to port and offload parts of a sequential algorithm to a GPU device, in order to speed-up critical and compute intensive functions. This has the advantage of maintaining the design of the original software project, but it keeps part of the computation on the CPU, while hindering the potential scalability. The goal is to completely offload the computation on the GPU, while the CPU is delegated to managing tasks and network handling. The original sequential software is redesigned: data structures are adapted to exploit GPU memory access patterns, and compute kernels are optimized w.r.t. GPU warp execution.

The Listing 1 highlights the second-order accurate algorithm implemented by the single GPU in which the *Monotonic Upwind Scheme for Conservation Laws* (MUSCL) reconstruction and the *intercell numerical Fluxes* are respectively computed by two identical *half-steps*. The loop concludes with the half-step summation. Notice the analogy with the general simulation algorithm for Finite Volume solvers described in Section 1.3: there is an outer `while` loop advancing the computational time-step t of a Δ_t value. In such algorithm the inner `for` loop, which computes the fluxes, is parallelized by the GPU kernels, which are able to process a large number of cells at the same time. The mapping between CUDA threads and memory cells is described below.

Recalling the description of BUQ grids in Section 1.2.1, the memory data structure which hosts the logical grid is designed for an efficient CUDA thread mapping to cells for parallel execution. The grid is allocated with a bi-dimensional array and tiled up with GPU blocks: each block on GPU corresponds to a logical block and the physical cell size is disregarded. The geographical information is maintained in additional data structures. The cells organization within a GPU block is as in the logical blocks, i.e. the offsets of corresponding cells are equal.

Figure 3.1 depicts logical blocks (a) and the allocation in GPU memory (b). They are indexed from left to right, top to bottom. Logical blocks are *linearly indexed* and filled in GPU memory according to that ordering. The indexing, w.l.o.g., starts from the lower resolution level (i.e. spatially larger blocks).

Figure 3.1: (a) Logical block numbering of a BUQG (cells are omitted). (b) GPU memory allocation of BUQG.



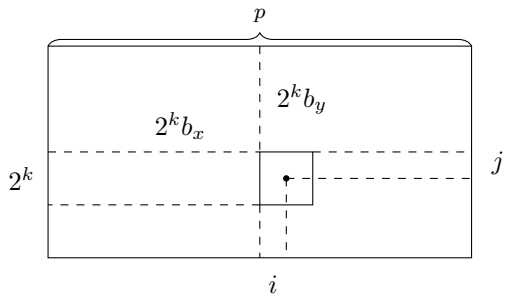
(a) (b)

For efficiency, cells are arranged on a 2D pitched matrix, with a logical schema as in Figure 3.1 (b): the data grid contains $n \times m$ blocks of size $2^k \times 2^k$ for a constant $k \in \{3, 4\}$, indexed by a mono-dimensional coordinate system.

GPU kernel execution organizes threads on a bi-dimensional CUDA grid made of $n \times m$ CUDA blocks, having $2^k \times 2^k$ cuda threads each. The correspondence between a thread and a cell is a one-to-one mapping between memory cell and CUDA threads. A thread is in charge of simulating the time-step of a single cell (compare with Equation 1.7). During the simulation, each thread frequently accesses the memory location of the cell which it is in charge of, together with its local neighborhood. Each logical thread is indexed by an address in the form (i, j), while each logical block by coordinates (b_x, b_y). The corresponding memory cells are retrieved by means of the conversion $M(i, j, b_x, b_y) = i + 2^k b_x + (j + 2^k b_y)p$ (see Figure 3.2), where p is the pitch size (e.g. the smallest number that is a multiple of

a suitable power of 2 and not smaller than $2^k m$). The pitch guarantees that rows of the matrix are aligned for efficient GPU global memory reads and it depends on the GPU card in use.

Figure 3.2: Absolute position of a cell inside memory retrieved from the block coordinates (b_x, b_y) and the cell coordinates (i, j).



Algorithm 1 Single GPU Simulation Algorithm

Require: $T_{max} \geq 0$

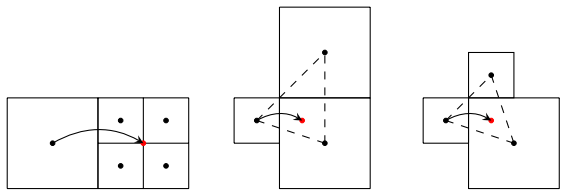
1: $t \leftarrow 0$
2: **while** $t \leq T_{max}$ **do**
3: $\Delta t_{min} \leftarrow$ deltaT(BUQGrid)
// First half-step
4: MUSCL(Δt_{min}, BUQGrid)
5: Flux(Δt_{min}, BUQGrid)
// Second half-step
6: MUSCL(Δt_{min}, BUQGrid)
7: Flux(Δt_{min}, BUQGrid)
// Half-steps summation
8: halfStepSum()
9: $t \leftarrow t + \Delta t_{min}$

3.3 Neighbor Retrieval Operation (NRO)

As the numerical stencil dictates, at each time-step $n+1$ the cell status is updated depending on the information of the cell at the time-step n, together with its surrounding cells, as described in Equation 1.8. There are two main issues that prevent optimal parallel performances: (i) border cells need to retrieve the content of their neighbor cells that belong to different blocks and (ii) the neighbor cells may have a different resolution. In particular, logically adjacent cells belonging to different blocks may be stored at distant and scattered memory locations. This violates the coalesced access pattern on GPU memory.

The Neighbor Retrieval Operation (NRO) gathers the value of one or more neighbors and interpolates them in order to model an adjacent cell of the same resolution, even in case the actual neighbors have a resolution change. Each thread executes such procedure. Figure 3.3 depicts the cases that involve a resolution change.

Figure 3.3: Different cells configurations, classified by neighbors resolutions. The red dot represents the simulated neighboring cell with the same resolution as the original one (beginning of the arrow).



The Neighbor Retrieval Operation (NRO) has a significant impact on parallel performances, if compared to the access of neighbors in Cartesian grids, which indeed preserve spatial locality in their memory representation. Inner cells of a block can find the neighborhood by regular and aligned memory accesses within the block (recall Section 3.2), while border cells need to query a graph-based data

structure in order to identify the correct neighbor blocks. The nodes of such graph are represented by block indexes, while edges by adjacency lists of neighbor block indexes. Depending on the disposition of blocks and their resolutions, a variable number of cells are processed and no aligned access is expected. For example, the block 10 in the Figure 3.1 corresponds to a graph node with six neighbors, namely $3, 6, 7, 8, 9$ and 11. Each graph node manages eight neighbors indicated by coordinates $N, S, W, E, NE, SE, NW, SW$. When a block faces two higher resolution blocks, both blocks are listed in the graph. For example node 3 has 10 and 8 as E neighbors. At each time-step, the graph is queried by threads mapped onto border cells, in order to identify correct blocks and cells in GPU memory. Notwithstanding the computational effort for dealing with border cells, the technique proved to speed up the computation, compared to uniform and high resolution, by 2 orders of magnitude [18].

3.4 Dry-Wet fronts

An interesting optimization implemented in PARFLOOD reduces the amount of calculations made by the solver by considering only the *wet* and *wettable* blocks at each time-step, discarding the *dry* ones. The wet blocks are those in which at least a cell has a water depth greater than 0 and the wettable ones are the (dry) neighbors of the wet blocks having water depth equal to 0. It follows that the wettable blocks may become wet in the following time-step and thus they need to be considered by the computational kernels (i.e., conserved variables of the dry cells might be updated). This optimization allows to further reduce the amount of blocks processed at each time-step, with a direct positive impact on the simulation time. To compute the wet blocks, each time-step the code loops over the domain cells; when a wet cell is found, the block index containing it is inserted inside an array A, which keeps track of all wet blocks. The wettable blocks are

identified from the wet blocks having at least a border wet cell. In that case the neighbor block on the adjacent side of that cell is considered wettable and can be inserted in the array A. It follows that the number of threads to spawn in total (i.e., the size of the kernels) is given by $n \cdot 2^k \cdot 2^k$ where n is the length of A and $2^k \cdot 2^k$ is the number of cells in a block. The number of CUDA blocks spawned can be considerably smaller than the total number of domain cells which would be processed otherwise.

Inside a GPU kernel, the CUDA block of coordinates b_x and b_y is linearized into an index $b_i = 0, \ldots, n-1$ and then the block threads are mapped into the memory space allocated for $A[b_i]$ (as described in Section 3.2), which is the linear index of a wet (or wettable) block and each one of its cells are computed by a thread. A further optimization performed by the computational kernels discards from the calculations the block's cells which are not wettable in the next step.

3.5 From Single GPU to Multi-GPU

Despite the significant speedups offered by the combination of GPU processing, multi-resolution discretization and the dry-wet fronts optimization, some limitations must be mentioned. Firstly, the size of the grids stored on the GPU global memory is limited as outlined in Section 1.4. In particular, the single GPU can be used to simulate floods extending over surfaces of at most $10^3 km^2$, discretized using multiresolution grids containing approximately 10^6 cells. The second limitation is related to the time scalability, provided that the code has been largely optimized and profiled over the years, the computational cost of large simulations is still remarkable. As outlined in Section 1.4 a real case simulation can take more than 20 hours of computation to simulate an event lasting 80 hours. To overcome these limitations an option can be the hardware upgrade (GPU in this case),

however, this solution pushes the solver just a little bit further in terms of performances. Instead, both memory and performance limitations can be overcome by decomposing the BUQ grid and by distributing the partitions over multiple devices as commonly done in the HPC environment.

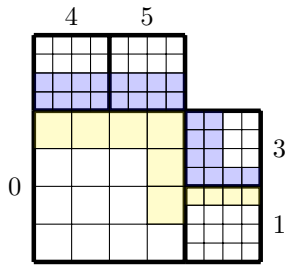
The domain decomposition is based on block granularity, meaning that whole blocks are entirely assigned to a specific GPU. As a natural extension of the Neighbor Retrieval Operation, adjacent cells belonging to blocks located on different GPUs need to communicate their content by exchanging messages through the network. The communication between partitions builds on the standard Message Passing Interface (MPI). The reader sould refer to Section 2.6 for a description of the MPI environment.

As design pattern, only one MPI process is assigned to each single CPU-GPU process. For a better control over memory transfer between CPU and GPU, the current implementation does not use *Unified Virtual Addressing* (UVA) [67], thus GPUs located on the same node, do not directly communicate with each other. Instead, to keep the implementation as general as possible, communication is performed at MPI level, between ranks either located on the same node or on different ones. The following sections describe in detail how to overcome the scalability limitations of the single-GPU, addressing one by one the goal of this thesis.

3.6 Communication data structures

When a pair of neighboring cells is located on different devices, a communication algorithm handles the exchanges between the devices in order to provide their information to the correct GPU memory. This section describes the multi-GPU NRO (MNRO), as generalization of the single GPU NRO (see Section 3.3). A partition dependent protocol is defined, establishing the order in which cells are sent from one process to another one. The protocol is driven by the adjacency graph

Figure 3.4: Two different partitions containing blocks $\{0, 1\}$ and $\{3, 4, 5\}$ respectively. The former has *horizon cells* highlighted in yellow, while the latter in blue.



of blocks and it is updated when the periodic dynamic load balancing modifies blocks assignments to ranks.

Before building the protocol, it is important to identify the cells to be exchanged between adjacent partitions. In detail, the *horizon cells* are defined as the local cells of a partition that are needed by some remote rank. For example, consider the subset of blocks $\{0, 1, 3, 4, 5\}$ from the Figure 3.1. Let us assume that a partitioning algorithm defines two partitions composed of the sets $\{0, 1\}$ and $\{3, 4, 5\}$.

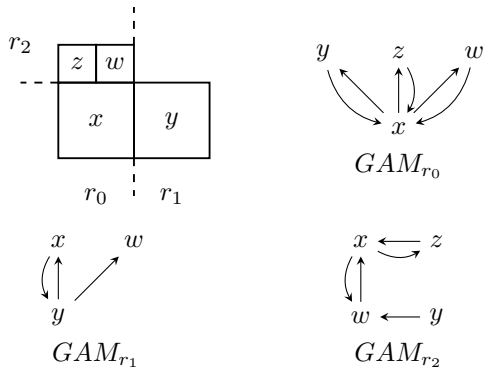
The horizon cells related to the first partition are highlighted in yellow in Figure 3.4, while the ones belonging to the second partition are highlighted in blue. Each MPI rank maps horizon cells into a linear buffer (write buffer), which is sent to neighboring partitions. Moreover, another buffer (read buffer) hosts the information received from other ranks. This section describes the preparation of write buffers and the decode of read buffers. For the sake of simplicity, from now on the cells will be omitted from the pictures, even if only horizon cells are actually communicated.

Let's make a brief description of the protocol. Consider the disposition of blocks as shown at top left of Figure 3.5. This is a typical scenario for rank data exchange: there are three MPI ranks r_0, r_1 and r_2 and each of them manages a

different partition. The block x belongs to r_0, the block y belongs to r_1 and z, w belong to r_2.

Each block is seen as an agent which evaluates its neighborhood and for each *remote neighbor* computes the cells to send and to receive from it. This information is stored inside a directed graph called *Ghost Area Map* (GAM). The Figure 3.5 shows the graphs associated to ranks r_0, r_1 and r_2, namely GAM_{r_0}, GAM_{r_1} and GAM_{r_2}. Outgoing edges from a source block indicate that some cells (i.e., horizon cells) have to be sent from the source to the target block. Due to multi-resolution dependencies, a block might need to receive remote data from another block, but at the same time does not need to send anything to it. In the figure, this happens for blocks w and y. In particular, compare the interpolation mentioned in Section 3.3: the bottom right cell of w needs to access the upper left cell of y (see the Figure 3.3).

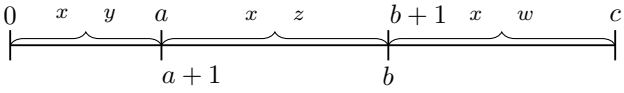
Figure 3.5: (Top left) Three MPI *ranks*, r_0, r_1 and r_2 with blocks x, y and $\{z, w\}$ respectively. (Top right and bottom) Ghost Area Maps built from processes r_0, r_1 and r_2.



GAMs alone are not informative enough for the handling of read and write buffers. For this reason, each graph is further extended by additional information attached to each edge. In particular, each rank, for each local block, sorts the order

of incoming edges from multiple remote blocks and prepares the read buffer as a concatenation of data associated to each of them. For example, let us consider the read buffer for block x, as depicted in Figure 3.6. A possible arrangement created by r_0 is: $[0, a]$ associated to (y, x), $[a+1, b]$ for (z, x) and $[b+1, c]$ for (w, x). This means that x needs to read $a + 1$ cells from y, $b - a$ cells from z and $c - b$ cells from w.

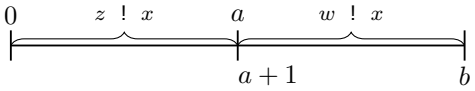
Figure 3.6: Read buffer R built by r_0: the block x reads horizon cells received from y, z and w. The resulting read buffer is the concatenation of each block's data.



Each rank precomputes the read buffer organization for each local block, and allocates the corresponding GPU memory. This memory region, referred to as *read buffer*, is denoted by R.

A similar approach is adopted to compute the write buffer, denoted by W, which stores the outgoing cells being sent. In this case, given a local block, the preprocessing algorithm computes the organization of the buffer in order to store the cells needed by each target block. For example, the process r_2 may associate the interval $[0, a]$ with the edge (z, x) and $[a + 1, b]$ with (w, x), as depicted in Figure 3.7.

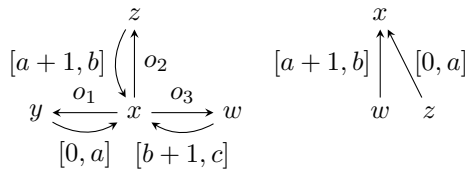
Figure 3.7: Write buffer W built from r_2: blocks z and w needs to send horizon cells to x.



Each partition r_i creates its R_i and W_i buffers independently from other partitions. Therefore, each r_i is not aware of the arrangements chosen by neighboring

partitions. For example, block x inside r_0 expects $b-a$ cells from z starting from the index $a+1$ in R_{r_0} (i.e., the R buffer of r_0). However, the block z in r_2 writes $a+1$ cells inside W_{r_2} (i.e., the W buffer of r_2) starting from 0. In order to match and retrieve correct data, adjacent partitions exchange their local GAMs. Such information allows each process to decode the W buffers as built and sent by the remote counterparts and to map them into the local R buffer. For example, r_2 sends its local graph to r_0, which in turn knows that x needs to receive cells from z and w. In this way r_0 extracts the edges (w, x) and (z, x) from r_2's graph, together with the buffer arrangement associated to them. The resulting GAMs stored by r_0 are shown in Figure 3.8. The graph on the left is used by computational kernels of r_0 to read data from R_{r_0} and to write outgoing cells into W_{r_0}. The symbols o_i are used to indicate the offsets used by r_0 for mapping the x border into W_{r_0}. The graph on the right is used by r_0 to decode incoming buffers sent from r_2 and store them into R_{r_0}, which is then uploaded on GPU.

Figure 3.8: (*left*) GAM_{r_0}, used to read and write horizon cells on GPU. (*right*) A portion of GAM_{r_2} needed by r_0 in order to read incoming buffers sent from r_2.



3.7 A general multi-GPU simulation algorithm

This section presents the general simulation algorithm and summarizes the high level operations carried out during each time-step computation. The presentation focuses on the multi GPU communication strategies and implementation. The details of the adopted shallow Water Equations (SWE) solver have already been

discussed in [17, 18]. As already pointed out, the ideas guiding the general handling of distributed multi-resolution domain can be adapted to a vast range of 2D finite volume solvers and they represent the main contribution of the thesis.

Algorithm 2 contains the main while loop, which terminates when the time variable t reaches the maximal physical time T_{max} to be simulated. At line 1, the current time t is set to 0 and an iteration counter is introduced to initiate the repartitioning algorithm every b iterations. The parameter b is referred to as the *repartitioning rate*: it controls how often the grids are rebalanced. Line 3 verifies whether b iterations are elapsed from last repartition and calls the dynamic load balancing procedure. Such procedure, detailed in Section 3.9, aims at minimizing the loop time duration differences among partitions, by redistributing portions of grid from longer (in term of loop-time) MPI processes to shorter ones. At line 4, the limiting minimum Δt is locally computed for each GPU and provided to the `mpiDtReduction` procedure, which computes the global minimum Δt value. Such call represents the only unavoidable synchronization barrier for this algorithm, because the Δt must be agreed on by all processes at each time step. It is implemented by a collective (blocking) call executed by all MPI ranks (i.e., `MPI_COMM_WORLD`, see Section 2.6). When reaching line 5, almost all processes spend some idle time and wait for other ranks to terminate their previous tasks. Lines 7 and 10 invoke a computational kernel that computes the Monotonic Upwind Scheme for Conservation Laws (MUSCL) reconstruction to achieve 2-nd order accuracy in space. A common CUDA design pattern overlaps the execution of GPU kernels and CPU tasks, with the goal of hiding the CPU operations while the GPU kernels are running. This can be done only if the two computations are independent. This feature is applied to two kernels `MUSCL` and `Flux`. They are divided into two parts each. The first part of each kernel (i.e., lines 7 and 12) computes the blocks that do not contain *Horizon Cells* (see Section 3.6). This computation is independent from other GPUs blocks and, concurrently, the CPU can perform the

MPI communication (lines 8, 13), implemented by the `MPI-BORDERS` procedure defined at line 19. Such procedure takes as input the W buffer along with the *Horizon Cells* to be sent to remote adjacent processes and it returns the R buffer, containing remote cells. In particular, for each adjacent partition (line 20) with rank r, a pair of non-blocking point-to-point (i.e., `MPI_Isend` and `MPI_Ireceive`) MPI calls is performed, one in charge of sending W (line 21) and the other one for receiving W_r (line 22), which contains remote cells received from r. In line 23 the `mpiWaitAll` procedure waits for all the communication to be completed. Even though `mpiWaitAll` is a blocking call, its overhead can be completely masked by the GPU kernels at lines 7 and 12, provided that those kernels are of a sufficient minimal duration. The procedure `processBorders` at line 25 decodes the received buffers and fills the R buffer according to Section 3.6. The R buffer is copied to the GPU (lines 9 and 14). Now the GPU can simulate the border blocks (lines 10 and 15) depending on remote cells stored in the R buffer. After line 16, a second half-step computation is performed (because of the second order time accuracy): it contains the same sequence of operations between lines 7 and 15. The cycle ends by increasing the loop counter and updating t.

3.8 Partitioning algorithms

In this work two partitioning algorithms have been tested. The simpler one sorts all blocks according to a specified topological order and splits this set in n parts, where n is the number of partitions. This type of partitioning will be referred to as 1D partitioning, because the resulting partitions can be seen as contiguous slices along the chosen axis. Figure 3.9 shows an example of a partition in four parts, based on the following topological order of points in space: $(x_0, y_0) < (x_1, y_1)$ iff $x_0 < x_1$ or ($x_0 = x_1$ and $y_0 < y_1$), with (x_i, y_i) being the block barycenter. This procedure produces an ordered sequence of block indexes, which is then split into n

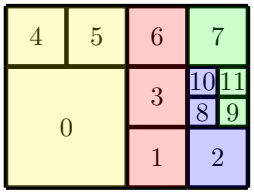
Algorithm 2 General Simulation Algorithm

Require: $b > 0, T_{max} \geq 0$

1: $loopCt \leftarrow 1$, $t \leftarrow 0$
2: **while** $t \leq T_{max}$ **do**
3: **if** $(loopCt$ **mod** $b) = 0$ **then** dynLoadBalancing( )
4: $\Delta t \leftarrow$ deltaTCalculation( )
5: $\Delta t_{min} \leftarrow$ mpiDtReduction(Δ_t)
6: $W \leftarrow$cudaMemcpyDeviceToHost( )
7: MUSCL(Δt_{min}, InternalBlocks)
8: $R \leftarrow$MPIBORDERS(W)
9: gpuMemcpyHostToDevice(R)
10: MUSCL(Δt_{min}, BorderBlocks, R)
11: $W \leftarrow$cudaMemcpyDeviceToHost( )
12: Flux(Δt_{min}, InternalBlocks)
13: $R \leftarrow$MPIBORDERS(W)
14: gpuMemcpyHostToDevice(R)
15: Flux(Δt_{min}, borderBlocks, R)
16: $W \leftarrow$cudaMemcpyDeviceToHost( )
... *Second half step*
17: $loopCt \leftarrow loopCt + 1$
18: $t \leftarrow t + \Delta t_{min}$

19: **procedure** MPIBORDERS(W)
20: **for all** $r \in AdjRanks$ **do**
21: mpiIsend(r,W)
22: mpiIrecv(r,W_r)
23: mpiWaitAll()
24: **for all** $r \in AdjRanks$ **do**
25: $processBorders(R, GAM_r, W_r)$
26: **return** R

Figure 3.9: A simple domain decomposition in four partitions, produced with the 1D algorithm.



equal sized subsequences (with one unit tolerance), where n is the number of MPI processes involved in the simulation. For example, in Figure 3.9, the topological ordering produces the sequence of twelve blocks `[4,0,5,1,3,6,8,10,2,7,9,11]`, which is then split into four equal parts.

Besides the simplicity of its implementation, this algorithm is a first attempt to balance the computational load by splitting partitions in equal parts and it is only implemented in a static form (i.e., without dynamic partitioning). Section 4.2 discusses the drawbacks and limits of such algorithm.

The second algorithm is based on Hilbert Space Filling Curves (HSFC) and it is implemented inside the Zoltan Library (see ug_alg_hsfc at [68]). The HSFC defines a curve along a self avoiding walk through a set of (barycenter of) blocks in the physical space, while preserving the proximity property between the traversed points. The key property is that if two points are close to each other on the curve (i.e. they have a similar traveled distance from the beginning of the curve), they are also close to each other in the physical space. The algorithm divides the curve into n pieces of equal length, which correspond to the set of blocks that form each partition. Such partitions are expected to have a relatively large area/perimeter ratio and the partition's border blocks are expected to be compact, with a limited impact on network communication of borders. As described in Section 3.7, MPI border communication is masked by the kernel working on internal blocks (if long

enough). Uncontrolled partitioning can increase the number of border blocks with worse masking.

The current implementation transfers to the Zoltan library a set of points in the 2D space (defined by the blocks barycentres). Based on the differences between previous and current divisions on the curve, each MPI process is given the set of blocks to be imported and the set of blocks to be exported. The analysis of such local migration pattern, allows to build a global table that contains, for each block, the new destination rank. The table defines the actual *data migration*. The domain partitioning relies on *Zoltan_LB_Partition* procedure, whilst the update of the partition weights is performed by *Zoltan_LB_Set_Part_Sizes*(see ug_interface_lb at [68]).

The grid allocation is static, in order to avoid expensive re-allocations at runtime, and it reserve some additional space for future blocks incoming from other partitions. The number of blocks of a partition is variable at runtime, depending on the spatial blocks exchanged during the load balancing procedure.

The approach adopted is to define two thresholds for the maximum (δ_M) and the minimum (δ_m) number of blocks which can be contained inside a partition. The partition size, in terms of blocks, belongs to the interval $[\delta_m, \delta_M]$ and this bounds the weights input to the Zoltan library. In particular $\delta_m > 0$ prevents partitions from emptying, which may happen when using the Zoltan Library, and δ_M prevents local grids from exceed a certain size. If a partition ever reaches one of the bounds, the dynamic load balancing is not performed.

3.9 Dynamic Load Balancing

This section discusses the challenges posed by an effective dynamic load balancing, in the context of a multi-GPU and multi-resolution framework. The synchronization barrier enforced at each iteration (line 5 of algorithm 2) performs a collective

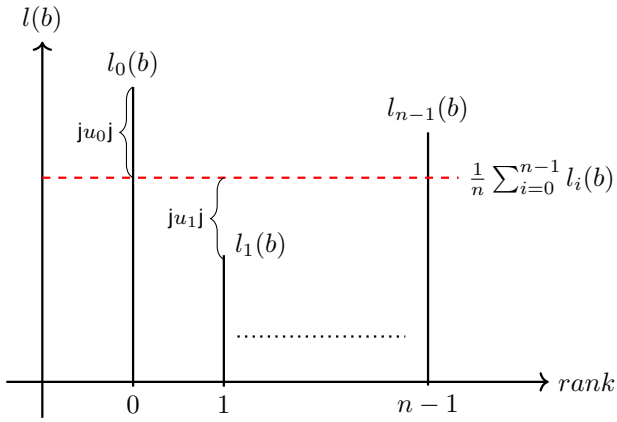
reduction call. Unavoidably, faster processes spend some idle time while waiting at the barrier (called *synchronization time*).

The key idea behind *dynamic load balancing* is to minimize the idle times by actively modifying the computational load depending on runtime performances. In PARFLOOD, the computational load of a rank can not be directly related to the number of cells/blocks to be computed. The main reason is caused by the presence of *dry* blocks, namely blocks where every cell contains no water. As mentioned in Section 3.4, for optimization purposes, such blocks can be ignored during the iteration, since no modifications are expected. During the simulation, a block may become partially wet and/or completely dry, depending on the simulation evolution. Let us also mention network delays, which are unpredictable and independent from computation. Nevertheless they can affect some of the ranks, depending on cluster topology and load.

Building a complete model for measuring the computational cost, given a specific partition is a complex task. We also believe that it is unnecessary, since a simpler algorithm can provide excellent results. A dynamic and efficient algorithm that acts *a posteriori* has been developed, depending on observations of recent unbalances in performances. Some blocks are migrated among ranks, since they represent the minimal processing structure for GPUs. The actual computational cost of a block is not known in advance, since it can vary during the simulation. If the migration has little impact, it will be periodically repeated, until the set of migrated blocks is able to rebalance the idle times.

In [44] it is presented a viable approach: the Dynamic Load Balancing (DLB) library [66] lets idle MPI processes lend their resources to other MPI processes (in the need of them) on the same node. In PARFLOOD, each grid is allocated on a specific GPU and the computational resources of a device are strictly associated to an MPI process. Slower processes are encouraged to send a portion of their grid to faster processes on different location over the network. The partition update

Figure 3.10: Representation of the heuristics used to balance the computational load among all MPI processes.



is periodically performed, with the aim to converge to a balanced situation. To this end, a weight $w \in [0, 1]$ is associated to each process, such that $\sum_{i=0}^{n-1} w_i = 1$, where n is the number of MPI processes. Initially, each process has weight $\frac{1}{n}$ and the grid is equally (block-wise) distributed among all partitions. At runtime, the weights are dynamically adjusted using the following heuristic: each process communicates its average synchronization time over the last b iterations and it computes its deviance from the average among all ranks. This measure suggests whether the process waits more than the others or it causes a slow down for the others. Ideally, a perfectly balanced iteration would vanish such times for every rank. Formally, each process p computes the difference between its synchronization time and the average one as:

$$u_p := l_p(b) - \frac{1}{n}\sum_{i=0}^{n-1} l_i(b) \tag{3.1}$$

where $l_i(b)$ is the average synchronization time of the MPI process i over the last b time-integration cycles. Averaged time filters out fluctuations due to temporarily

network slowdowns and measure errors. Figure 3.10 depicts the main idea: the red dashed line represents the average synchronization time among all processes. The ultimate goal of the heuristic is to minimize $|u_i|$. For example, the rank 0 is considered *lighter* than rank 1, because $l_0(b) > l_1(b)$, meaning that process 0 spends more time in an idle status. Furthermore, since the process 0 has an idle time above the average (red line), the value u_0 is positive, meaning that rank 0 could accept extra blocks in its grid from some other process, with the probable effect of increasing the kernel processing and symmetrically reducing the synchronization time. Vice versa, the process 1 has $u_1 < 0$, meaning that it should export some of its blocks. Once u_p has been computed, process p maps u_p into the interval $[-1, 1]$ by dividing it for the *maximum average loop time* over b cycles; formally

$$z_p := \frac{u_p}{max_{0 \leq i \leq n-1}\{L(b)\}} \tag{3.2}$$

The loop-time L has been chosen instead of the synchronization time l, because L is orders of magnitude greater than l and in case of no network overheads (i.e. GPUs are located on the same node and a well balanced computational load) $l \sim 0$. Finally, a new weight w'_p is computed as $w_p + z_p\varepsilon$, where $0 \leq \varepsilon < 1$ is an arbitrary constant called *sensitivity* used to control the weight variation z_i. Note that $\sum_{i=0}^{n-1} w'_i = 1$. The weights w' are used to control the partitioning algorithm, which is in charge of determining a proportional new block assignment to ranks. In an incremental framework, this produces a minimal set of blocks that are sent to a remote partition, as detailed in Section 3.8.

4

Numerical Tests

This section assesses the performance of multi-GPU PARFLOOD implementation by means of a comprehensive set of tests, aimed at supporting the impact of presented ideas. All tests have been carried out on the Piz-Daint cluster located at CSCS Swiss National Supercomputing Centre. The system nodes are equipped with a Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100. In particular, each node is equipped with a single GPU and the communication between GPUs is performed across distinct physical nodes. Furthermore, the type of processors has limited impact, since the computation is completely offloaded to GPUs. Section 4.1 presents the results of *weak scalability* tests using both 1D and 2D-HSFC partitionings algorithms. Section 4.2 addresses the improvements obtained by overlapping the MPI communication with kernel execution. The same section also establishes the minimal GPU load to achieve the optimal node speedup and network masking in the *strong scalability* test. Section 4.3 compares the performance of 1D against the 2D-HSFC partitioning (without dynamic load balancing) by performing a strong scalability test. Finally, the Section 4.4 highlights the performances of dynamic load balancing, implemented as described in Section 3.9.

4.1 Weak Scalability

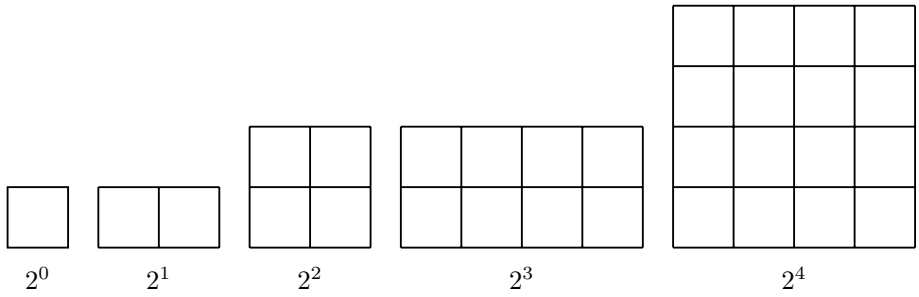
The weak scalability test quantifies the overheads introduced by the multi GPU code and also compares the efficiencies obtained by 1D and HSFC partitionings. Recalling the Section 2.2, the *weak scaling efficiency*, computed as a percentage, is defined as $100 \cdot T_1/T_n$, where T_i is the simulation time obtained using i GPUs. The weak scaling test is parametric in the number of GPU in use and it defines a constant partition size to be assigned to all processes involved. The test increases the number of partitions (or processes) involved in the simulation and evaluates the efficiency.

A crucial aspect, in order to produce meaningful results and quantify the overheads introduced by the multi-GPU code, is that the computational time (network and overheads excluded) assigned to each partition should be balanced because it is difficult to directly control such metric, given the multi resolution framework. For this reason, a test with fixed shape partitions (constant border size) and with uniform resolution blocks is created. The latter constraint helps in the generation of parametric models.

The tests define a constant partition size of $2,5 \times 10^6$ cells. This size is sufficient to guarantee a GPU proper load and stable and comparable kernel performances. The only difference between 1D and 2D partitioning algorithms is the shape of the input model. In 1D setting, grids have a fixed height and they grow along the x axis, thus generating a $1 \times n$ ratio grid for n processes. The 2D setting uses a square grid that grows in both directions, in order to allow the HSFC partitioning to fill the 2D space and produce partitions with equal size (see Figure 4.1).

The tests have been performed on 2^i GPUs, for $i = 0, \ldots, 6$ (i.e, up to 64 GPUs). Since each partition has exactly the same number of blocks, the expected running time for each test is the same. In both cases, the MPI communication masking is active, meaning that border communication is overlapped with CUDA

Figure 4.1: Generation of 2D input models with 2^i partitions.



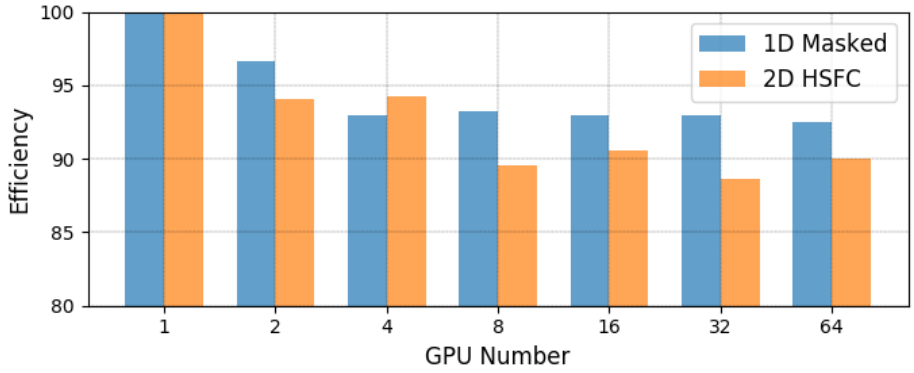
kernels dedicated to internal blocks (i.e. the ones that do not requires information from other MPI processes). Figure 4.2 shows the computational efficiency obtained for simulations with an increasing number of GPUs. The 1D partitioning converges to an efficiency of 93% and the 2D version performs with an efficiency around 90%. In both cases the loss of efficiency is caused by the Δ_t reduction (i.e., line 5 of Algorithm 2). Such instruction acts as a blocking MPI barrier that cannot be masked. The time step reduction takes slightly more time in the HSFC version, because each MPI rank has more neighbors than in the 1D partitioning. Finally, in both $1D$ and $HSFC$ partitioning, the network communication overheads are completely masked with GPU kernel execution for internal blocks.

4.2 1D Strong Scalability Test

The goal of this test is to compare an algorithm version without the masking of the MPI communication against the optimized version which implements the masking.

As already mentioned in Section 2.2, the *strong scaling efficiency*, computed as a percentage, is defined as $100 \cdot T_1/(nT_n)$, where T_i is the simulation time obtained using i GPUs. The input model consists of a circular dam break discretized by means of a multiresolution grid having 20 millions cells. The Figure 4.3 depicts the resolution levels of the input model: the darker the color, the higher the

Figure 4.2: Efficiency graph of the Weak Scalability Test comparing $1D$ partitioning with the $2D$-HSFC one.

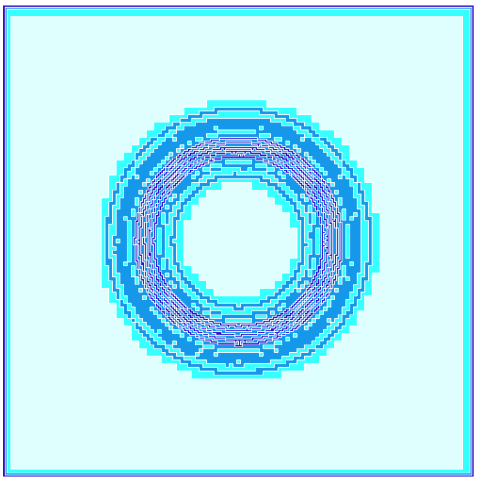


resolution level. The domain is partitioned along one dimension (1D) up to 32 GPUs. The Figure 4.5 (left) shows the partitioning produced by the 1D algorithm (Section 3.8) with eight processes (the two partitions at the far left and right are very narrow and almost invisible due to the high resolution blocks imposed by the multiresolution partitioning). The graph in Figure 4.4 shows that the optimized version scales significantly better than the other one.

For 4 GPUs the efficiency drops to 93% for the masked version and to 90% for the non masked one. The gap grows for 8, 16 and 32 GPUs with a difference of 10% for 32 GPUs. The optimized version is able to mask the communication even for a rather small partition size (i.e. simulation with 32 GPUs, in which each GPU contains about 0.5M cells).

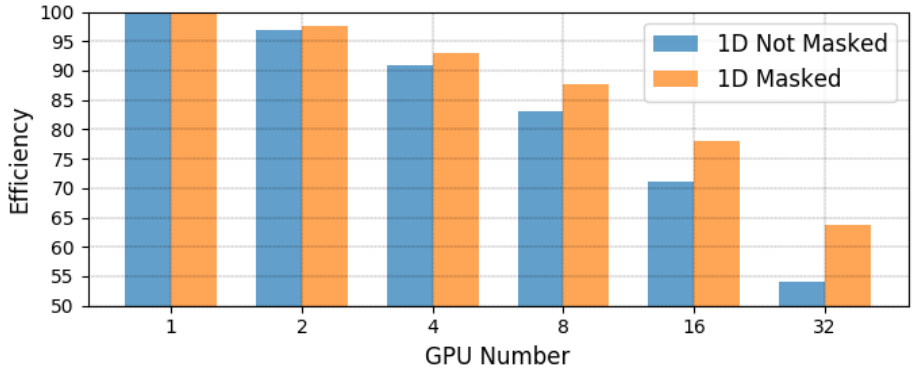
Ruling out the overheads caused by border communication, the reasons behind the drop of efficiency that also affects the optimized version have been investigated. by using the NVIDIA profiler tool `nvprof`. The profiler keeps track of the GPU kernels execution times. Since `nvprof` does not support natively the CPU profiling, the NVIDIA *tools extension* (in short `nvtx`) API has been used to pinpoint

Figure 4.3: Circular Dam Break discretized by means of a BUQ grid having multiple resolution levels. The darker the color, the higher the resolution level.



all the tasks performed on CPU (e.g. MPI communication, border processing and time step reduction) and visualize them on the NVIDIA visual profiler (i.e., `nvvp`) together with all the kernels data generated by `nvprof`. As result of the investigation, the major reason behind the efficiency loss is caused by the border kernels, which deal with basically constant number of blocks, independently on the number of GPUs in use. This directly follows from the 1D partitioning schema: when partitioning the domain along one dimension, the internal kernels scale (i.e., lines 7 and 12 of Algorithm 2), since the number of internal blocks of a partition gets divided proportionally. Unfortunately, the border kernels (i.e., lines 10 and 15) show an almost constant behavior (small variations can be present if the grid contains multi resolution blocks), because the border size is basically constant. Referring to the Figure 4.5 (left), increasing the number of partitions results in increasing the number of slices without decreasing the border size. As an example consider the Figure 4.5 (right) featuring a domain decomposition over 16 MPI ranks using HSFC. The borders are considerably smaller compared to the

Figure 4.4: Efficiency of the strong scalability test of the non-masked $1D$ version and the masked one.



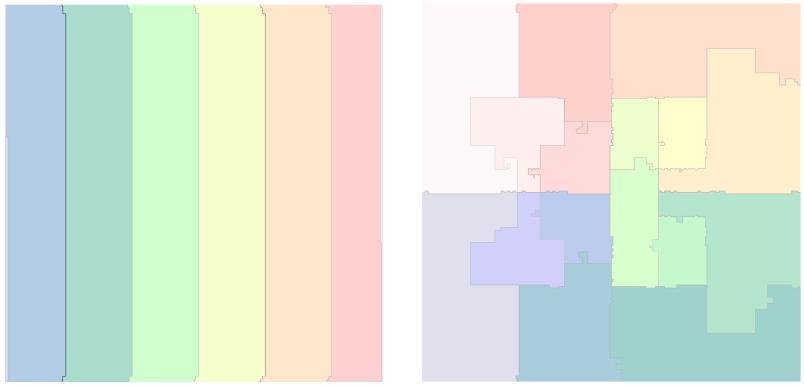
$1D$ case. Section 4.3 compares these two partitionings. In the 1D partitioning, the efficiency is hindered by the synchronization barrier at Δt reduction (line 5 of Algorithm 2). The waiting time at the barrier increases with the number of GPUs and it is caused by the different computational load among GPUs. The multi-resolution grid triggers branches in the code and causes small variations in the time-step duration. Concluding, the drop-down of efficiency is mainly due to the non-scalable border size and secondarily to the unbalanced computational load caused by the multi-resolution grids.

4.3 Strong Scalability HSFC vs 1D

This section investigates the performance enhancements introduced by the 2D partitioning based on Hilbert Space Filling Curves. Furthermore, it determines the lower bound for perfect scaling of CUDA kernels processing internal blocks (lines 7 and 12 of Algorithm 2).

The ideal setting for capturing such aspects is a BUQ grid with uniform res-

Figure 4.5: Partitioning: 1D in eight parts (left) and HSFC in 16 parts (right) of the grid in Figure 4.3.



olution and rectangular arrangement of blocks. The grid is internally handled as a BUQG (i.e., using the block numbering as mentioned in Section 3.2). This setting allows to validate the BUQG infrastructure, without incurring in the multi resolution related overheads, that may introduce unbalanced kernel executions.

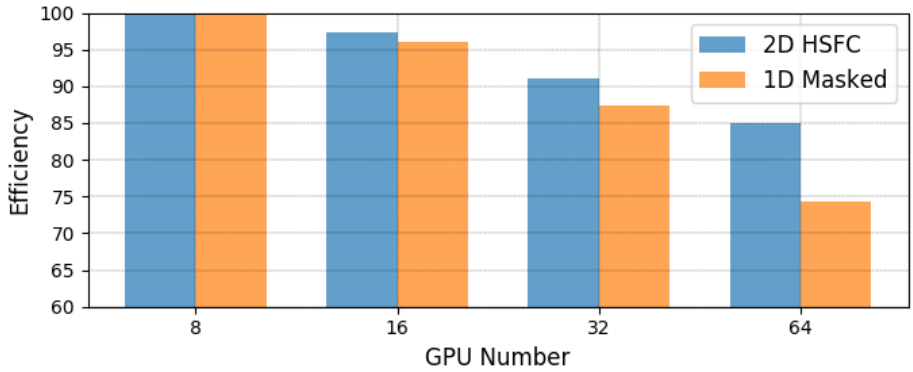
The input model for the strong scalability test is a circular dam break discretized by means of a uniform BUQ grid containing approximately 82 millions cells. This model is designed to guarantee that the HSFC algorithm creates rectangular shaped partitions that minimize the border size.

The test is performed ranging from 8 to 64 GPUs. In order to have a fair comparison, each GPU should run kernels with nearly 100% efficiency and a minimal grid size is guaranteed for each GPU. This requires an initial grid size that can not be stored in GPU memory for less than 8 GPUs.

Figure 4.6 shows the improvements introduced by HSFC. As stated in Section 4.2, border kernels of $1D$ partitioning do not scale and tests show that the efficiency drops almost linearly with the number of GPUs, reaching 73% for 64 GPUs. This is not the case for HSFC, which reaches 85% of efficiency with 64 GPUs and a partition size of approximately 1.3 million cells. Tests with smaller

input grids show a loss of efficiency in the inner kernel on the single GPU. This is the empirical measure of the suggested GPU load for best occupancy. The HSFC algorithm shows visible improvements, compared to $1D$, starting from 16 partitions.

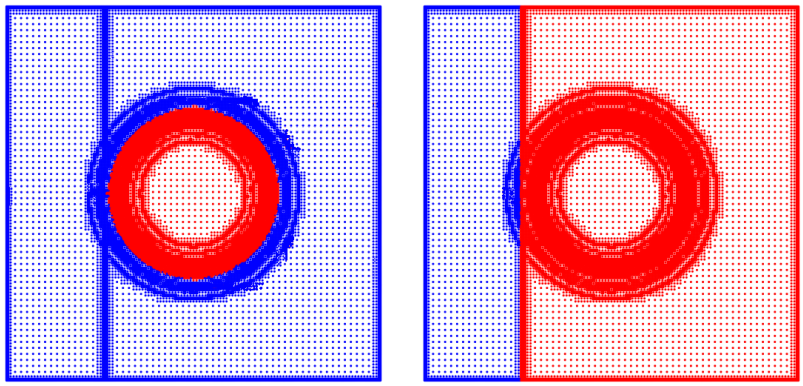
Figure 4.6: $2D$-HSFC partitioning and $1D$ partitioning strong scalability tests.



4.4 Dynamic Load Balancing on a realistic scenario

This Section assesses the DLB heuristic on a realistic test scenario. The input model for this test is a circular dam-break located at the centre of a domain of size $50m \times 50m$ and discretized by means of a multi-resolution grid containing $\sim 6.5 \times 10^6$ cells. The grid is shown in Figure 4.7 (on the left), where the wet and dry portions of the grid are shown in red and blue, respectively. A vertical wall is introduced which confines the water on the right side of the domain while the left region is *non-wettable*. The wall is covered by high resolution blocks and it is visible as a denser vertical line. Figure 4.7 (right) shows the end of the simulation, where water completely covers the right region.

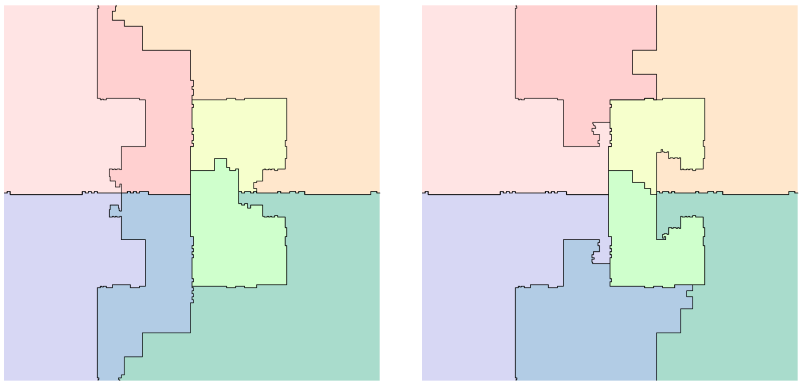
Figure 4.7: Multi-resolution circular dam-break with dry (blue) and wet (red) blocks at the start of the simulation (left) and at end (right). A wall prevents the water from inundating the left portion of the domain.



Initially, the domain is partitioned into eight equal parts (i.e., with the same number of cells) as shown in Figure 4.8 on the left. The solver processes only the wet cells during each time-step (i.e., the red ones). Therefore the cells in the non-wettable zone will never be processed, making the partitions containing such cells computationally less expensive in the long term. The heuristic defined in Section 3.9 is able to capture such unbalances and dynamically adapt the partitions size, as shown in Figure 4.8 on the right, converging to a balanced situation. Note that large partitions on the left side of the domain contain a significant number of dry cells. Such partitions have the same computational weight as wet and smaller partitions.

Figure 4.9 compares the time-step duration (e.g. the duration of each `while` cycle in Algorithm 2) in static (depicted in dashed blue) and dynamic (in red) versions of the solver. The loop-times of different ranks overlap since they are synchronized by the barrier at each iteration. The re-partitioning rate (i.e., b in Algorithm 2) is set to 500: the dynamic version executes the repartitioning each 500 cycles whilst the static one simply gathers timing information for statistics. As

Figure 4.8: (Left) Initial partitioning of the circular dam-break in eight sub-grids of the same size. (Right) Partitions at the end of the simulation after the dynamic load balancing.

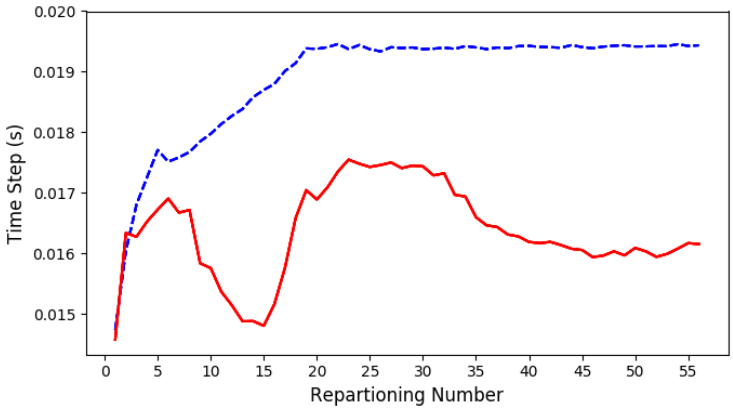


expected, at beginning both versions perform with the same time step duration. During the simulation the wet area reaches more cells and correspondingly the time step duration increases. In the static version the increase reaches $\sim 19ms$, while the dynamic one is able to reduce the time to $\sim 16ms$, thanks to the load balancing.

As mentioned in Section 3.9, "lighter" partitions spend some idle time while executing the global Δt reduction (i.e., line 5 of Algorithm 2). Such idle times are expected to be (almost) constant in the static version from a certain point on, and to decrease in the dynamic one. Figure 4.10 depicts synchronization times for each of the 8 ranks (dashed blue lines for static case and solid red lines for dynamic one). In both cases the "heaviest" rank has a synchronization time equal to zero at the beginning. In the dynamic case, all ranks, as expected, converge to the same value (in the order of microseconds), while the static case stabilizes once wettable part of the domain is completely reached. (see Figure 4.7 on the right).

Figure 4.11 reports, for each rank, the partition weight w (in dashed blue) together with the weight correction z (in red), both defined in Section 3.9 and

Figure 4.9: Static and dynamic computational loop times.



referred to the dynamic case. As the water inundates the domain, the heuristic described in Section 3.9 adapts the weights of each partition at runtime by moving blocks from the heavier partition to the lighter ones, especially those containing the non-wettable zone. Similarly, the z values converge to 0 as the load is gradually balanced.

In conclusion, the introduction of the dynamic load balancing shows that it is possible to remarkably reduce the time step duration and make the dynamic version faster. In this particular test-case the dynamic version performed 16% faster compared to the static one. The cost of the repartitioning procedure is only 4.5% of a single time-step. Such cost is distributed over the $b = 500$ iteration (the repartitioning rate) and it is possible to further reduce it, by increasing the b value and/or by moving a greater proportion of blocks each time (for faster convergence).

Figure 4.10: Synchronization times for static (dashed blue) and dynamic (red) versions.

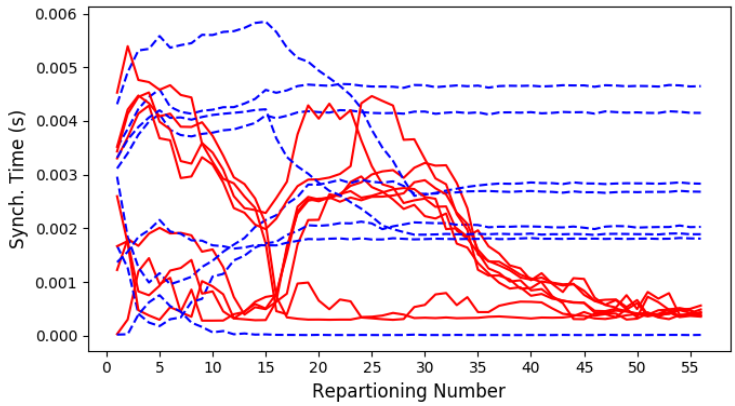
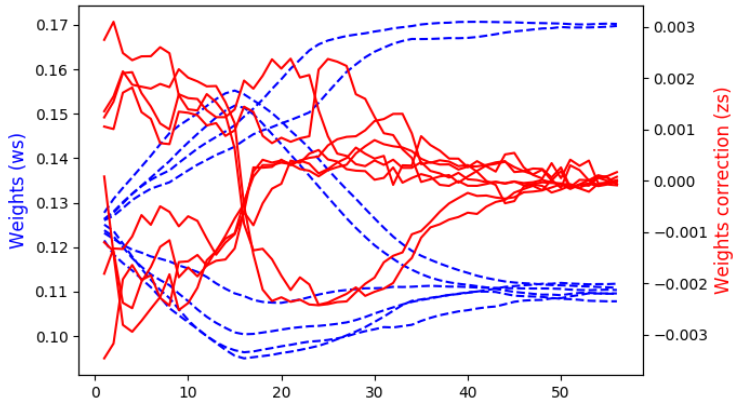


Figure 4.11: Dynamic load balancing: Weights corrections (in red) and variations (dashed blue) for each partition.



5

Conclusions and Future Works

The work carried out in this thesis is part of a research project investigating the possible ways of dealing with flooding events. Such inevitable and life-threatening calamities are likely to increase in frequency and intensity in the near future as a consequence of the global warming [8] [9]. The only way we have to deal with these phenomena is to increase the resilience of the territory by making fast and accurate predictions in order to limit the damages to people and infrastructures.

In practice, the evolution in time of flooding phenomena can be described through a mathematical model and simulated using a computer program, also called *solver*, that can be used to (i) simulate the evolution of a flood in progress and (ii) model a hypothetical scenario of particular likelihood and observe its behavior.

The former case is helpful in those situations where a fast reaction is needed in order to protect people and infrastructures, for example by making fast evacuation plans. A fast reaction can be achieved only if the ratio of simulation time to physical time is very low. In other words the first case needs *realtime* (or quasi-realtime)

simulations. The latter scenario can be used to prevent a flood which is likely to happen, for example by working on the mitigation infrastructures. This thesis addresses both problems by presenting a general GPU-MPI parallelization of a finite volume solver named PARFLOOD, which simulates flooding events by solving the $2D$ Shallow Water Equations (SWE). The SWE proved to be the right choice for describing events like river inundations where the water can propagate for thousands of square kilometers. The alternatives to the SWE are $1D$ or $3D$ models. While the former ones lack in generality the latter ones are computationally too expensive for this kind of problems [15].

PARFLOOD adopts a multi-resolution grid named *Block Uniform Quad-Tree Grid* designed to be efficiently exploited by NVIDIA GPUs. Such grid represents a good compromise between Cartesian and unstructured grids (e.g., triangle meshes) in that allows to reduce the memory occupancy while preserving high levels of accuracy [18]. Notwithstanding the great results obtained by the single GPU implementation of the solver, the lack of scalability is the main factor which led to the multi-GPU implementation.

To this end, the commonly known approaches in the field *High Performance Computing* (HPC) have been adopted in order parallelize the solver. In particular (i) two partitioning algorithms have been implemented together with (ii) the data structures for an efficient border communication and (iii) a load balancing heuristic to guarantee the same computational load among differed partitions. The first partitioning algorithm considered, divides the input grid along one direction whilst the second algorithm, based on Hilbert Space Filling Curves (HSFC), is able to produce $2D$ partitions. Both algorithms distribute the same amount of cells to each partition and exhibit a constant trend in the Weak Scalability Test, maintaining a $\sim 90\%$ efficiency up to 64 GPUs. In both cases the $\sim 10\%$ of efficiency loss is largely due to the time-step reduction.

The Strong Scalability Test is where HSFC outperforms the $1D$ algorithm. The increasingly large number of partitions having smaller size highlight the main problem with the $1D$ algorithm: the border size remains constant together with the amount of time needed to process the border cells. For this reason, the strong scaling efficiency drops to 74% with 64 GPUs while HSFC still scales at 85% on a grid containing ~ 85 millions cells.

The border communication between adjacent partitions is implemented using the MPI standard and takes advantage of specific data structures designed to deal with a variable number of neighbor blocks, depending on the multiresolution. To quantify the overheads introduced by the border communication we performed a strong scalability test comparing two versions of the solver obtained by turning on and off the overlapping between GPU kernels and border communication. The results showed a significant increase in the efficiency, up to 10% higher on 32 GPUs, when the MPI communications are masked by GPU kernels.

In order to model realistic case scenarios, PARFLOOD manages dry-wet fronts by excluding from the time-step computation all the dry cells which are not adjacent to any wet cell. This functionality has the side effect of creating considerable load imbalances among partitions, due to the ever changing number of wet cells to be processed. For this reason a load balancing heuristic has been implemented, in order to guarantee an almost perfect balance among partitions during the simulations. The heuristic aims at minimizing the synchronization times arising from the unavoidable collective time-step reduction and proved to be effective in a real test case scenario. In particular the results showed that the synchronization times can be globally reduced to the order of microseconds with 8 GPUs with a consequent decrease in the time-step duration.

In conclusion, thanks to the work carried out in this thesis, it is now possible to perform simulations on multiple GPUs, overcoming the memory limitations of a single GPU. Empirical tests suggest that PARFLOOD is able to maintain an

efficieny of 85%, and therefore a linear speedup on the computation, by keeping the partition size at least $1,3 \cdot 10^6$ cells. In this way, a simulation lasting 20 hours on one GPU, can be completed in ~ 3 hours using 8 GPUs.

In the future, in order model real flooding events, the further capability of dealing with *open boundary conditions* must be added to the multi-GPU implementation of the solver. In short, the challenge is to treat cells having boundary conditions as a single unit which is forced to stay on the same partition. In PARFLOOD, those cells having the same boundary condition are modeled through a segment in the $2D$ space which is rasterized into a set of cells during the preprocessing stage of the solver, when the multi-resolution grid is firstly created. Such cells may belong to multiple blocks and consequently, if the partitioning algorithm decides to migrate a block having a boundary condition, then all blocks containing cells with the same condition must be migrated together. In the future we also plan to compare different graph based 2D partitioning algorithms, to perform a systematic convergence study for the dynamic partitioning algorithm and to test large test cases having at least 100M cells.

References

[1] Valerie Masson-Delmotte, P Zhai, HO Pörtner, D Roberts, J Skea, PR Shukla, Anna Pirani, W Moufouma-Okia, C Péan, R Pidcock, et al. **IPCC, 2018: Summary for Policymakers**. *Global warming of*, **1**, 2018. 1

[2] IPCC. **Climate Change and Land**. https://www.ipcc.ch/report/srccl/. Accessed: 17-09-2019. 1

[3] Nigel W Arnell and Simon N Gosling. **The impacts of climate change on river flood risk at the global scale**. *Climatic Change*, **134**(3):387–401, 2016. 2

[4] Lorenzo Alfieri, Luc Feyen, Francesco Dottori, and Alessandra Bianchi. **Ensemble flood risk assessment in Europe under high end climate scenarios**. *Global Environmental Change*, **35**:199–212, 2015. 2

[5] Günter Blöschl, Julia Hall, Juraj Parajka, Rui AP Perdigão, Bruno Merz, Berit Arheimer, Giuseppe T Aronica, Ardian Bilibashi, Ognjen Bonacci, Marco Borga, et al. **Changing climate shifts timing of European floods**. *Science*, **357**(6351):588–590, 2017. 2

[6] Hessel C Winsemius, Jeroen CJH Aerts, Ludovicus PH van Beek, Marc FP Bierkens, Arno Bouwman, Brenden Jongman, Jaap CJ Kwadijk, Willem Ligtvoet, Paul L Lucas, Detlef P Van Vuuren, et al. **Global drivers of future river flood risk**. *Nature Climate Change*, **6**(4):381, 2016. 2

[7] Liang Yang, Jürgen Scheffran, Huapeng Qin, and Qinglong You. **Climate-related flood risks and urban responses in the Pearl River Delta, China**. *Regional Environmental Change*, **15**(2):379–391, 2015. 2

[8] L Alfieri, P Burek, L Feyen, and G Forzieri. **Global warming increases the frequency of river floods in Europe**. *Hydrology and Earth System Sciences*, **19**(5):2247–2260, 2015. 2, 101

[9] Lorenzo Alfieri, Berny Bisselink, Francesco Dottori, Gustavo Naumann, Ad de Roo, Peter Salamon, Klaus Wyser, and Luc Feyen. **Global projections of river flood risk in a warmer world**. *Earth's Future*, **5**(2):171–182, 2017. 2, 101

[10] UNFCCC. **The Paris Agreement**. https://unfccc.int/process-and-meetings/the-paris-agreement/the-paris-agreement. Accessed: 17-09-2019. 2

[11] Lorenzo Alfieri, Luc Feyen, and Giuliano Di Baldassarre. **Increasing flood risk under climate change: a pan-European assessment of the benefits of four adaptation strategies**. *Climatic Change*, **136**(3-4):507–521, 2016. 2

[12] Euleterio F. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. Wiley, February 2001. Out of Production. 3

[13] Euleterio F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer-Verlag Berlin Heidelberg, 3 edition, 2009. 3

[14] Eleuterio F Toro. **Lectures on Hyperbolic Equations and Their Numerical Approximation**. In *Non-Newtonian Fluid Mechanics and Complex Flows*, pages 91–169. Springer, 2018. 3

[15] Alessia Ferrari. *2D Shallow Water modelling of flood propagation: GPU parallelization, non-uniform grids, porosity, reverse flow routing*. PhD thesis, University of Parma, Parco Area delle Scienze, 181/A, 43124 Parma PR, 3 2018. 3, 102

[16] Alessia Ferrari, Renato Vacondio, Susanna Dazzi, and Paolo Mignosa. **A 1D–2D Shallow Water Equations solver for discontinuous porosity field based on a Generalized Riemann Problem**. *Advances in water resources*, **107**:233–249, 2017. 4, 68

[17] R. Vacondio, A. Dal Palù, and P. Mignosa. **GPU-enhanced Finite Volume Shallow Water solver for fast flood simulations**. *Environmental Modelling & Software*, **57**:60 – 75, 2014. 4, 65, 68, 80

[18] R. Vacondio et al. **A non-uniform efficient grid type for GPU-parallel Shallow Water Equations models**. *Environmental Modelling & Software*, **88**:119 – 137, 2017. 4, 11, 14, 65, 67, 68, 73, 80, 102

[19] Renato Vacondio, Francesca Aureli, Alessia Ferrari, Paolo Mignosa, and Alessandro Dal Palù. **Simulation of the January 2014 flood on the Secchia River using a fast and high-resolution 2D parallel shallow-water numerical scheme**. *Natural Hazards*, **80**(1):103–125, 2016. 4, 68

[20] Francisco Alcrudo and Pilar Garcia-Navarro. **A high-resolution Godunov-type scheme in finite volumes for the 2D shallow-water equations**. *International Journal for Numerical Methods in Fluids*, **16**(6):489–505, 1993. 4

[21] Vincenzo Casulli and Roy A Walters. **An unstructured grid, three-dimensional model based on the shallow water equations**. *International journal for numerical methods in fluids*, **32**(3):331–348, 2000. 4

[22] **Open MPI: Open Source High Performance Computing**. https://www.open-mpi.org/, 2019. Accessed: 17-09-2019. 17, 63

[23] MPI Forum. **Message Passing Interface (MPI) Forum Home Page**. http://www.mpi-forum.org/. Accessed: 17-09-2019. 17, 57

REFERENCES

[24] **Top 500: statistical list of supercomputers**. https://www.top500.org/. Accessed: 17-09-2019. 19, 66

[25] Kenta Yamamoto, Vikas Dubey, Katsumasa Irie, Hanayo Nakanishi, Himanshu Khandelia, Yoshinori Fujiyoshi, and Kazuhiro Abe. **A single K+-binding site in the crystal structure of the gastric proton pump**. *bioRxiv*, page 608851, 2019. 20

[26] Gabriel Wlazłowski, Kazuyuki Sekizawa, Maciej Marchwiany, and Piotr Magierski. **Suppressed solitonic cascade in spin-imbalanced superfluid Fermi gas**. *Physical review letters*, **120**(25):253002, 2018. 20

[27] Ebru Bozdağ, Youyi Ruan, Nathan Metthez, Amir Khan, Kuangdai Leng, Martin Van Driel, Mark Wieczorek, Attilio Rivoldini, Carène S Larmat, Domenico Giardini, et al. **Simulations of seismic wave propagation on Mars**. *Space Science Reviews*, **211**(1-4):571–594, 2017. 20

[28] Julien Seguinot, Susan Ivy-Ochs, Guillaume Jouvet, Matthias Huss, Martin Funk, and Frank Preusser. **Modelling last glacial cycle ice dynamics in the Alps**. *The Cryosphere*, **12**(10):3265–3285, 2018. 20

[29] **CUDA Toolkit Documentation Architecture and programming model**. https://docs.nvidia.com/cuda/. Accessed: 17-09-2019. 20, 47

[30] The Khronos Group. **OpenAcc**. https://www.khronos.org/opencl/. Accessed: 17-09-2019. 20

[31] The OpenMP Architecture Review Board. **OpenMP**. http://openmp.org//. Accessed: 17-09-2019. 20

[32] OpenACC Organization. **OpenAcc**. http://www.openacc-standard.org/. Accessed: 17-09-2019. 20

[33] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. **Kokkos: Enabling manycore performance portability through polymorphic memory access patterns**. *Journal of Parallel and Distributed Computing*, **74**(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. 20

[34] Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E Nagel, and Michael Bussmann. **Alpaka–An Abstraction Library for Parallel Kernel Acceleration**. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 631–640. IEEE, 2016. 20

[35] The Khronos Group. **SYSCL**. https://www.khronos.org/sycl/. Accessed: 17-09-2019. 20

[36] IEEE. **Standards for ethernet**. https://standards.ieee.org/standard/802_3-2018.html. Accessed: 17-09-2019. 21

[37] Gregory F Pfister. **An introduction to the infiniband architecture**. *High Performance Mass Storage and Parallel I/O*, **42**:617–632, 2001. 21

[38] OpenSFS and EOFS. **Lustre File System**. http://lustre.org/. Accessed: 17-09-2019. 22

[39] **IBM General Parallel File System (GPFS)**. https://www.ibm.com/support/knowledgecenter/en/SSPT3X_3.0.0/com.ibm.swg.im.infosphere.biginsights.product.doc/doc/bi_gpfs_overview.html, 2019. Accessed: 17-09-2019. 22

[40] Adela Ionescu. *Computational Fluid Dynamics: Basic Instruments and Applications in Science*. BoD–Books on Demand, 2018. Accessed: 17-09-2019. 23, 30

[41] **Slurm Workload Manager**. https://slurm.schedmd.com/documentation.html. Accessed: 17-09-2019. 24

[42] J. M. Domínguez et al. **New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters**. *Computer Physics Comm.*, **184**(8):1848–1860, 2013. 28, 68

[43] Sparsh Mittal and Jeffrey S Vetter. **A survey of CPU-GPU heterogeneous computing techniques**. *ACM Computing Surveys (CSUR)*, **47**(4):69, 2015. 29

[44] G. Houzeaux et al. **Dynamic load balance applied to particle transport in fluids**. *International Journal of Computational Fluid Dynamics*, **30**(6):408–418, 2016. 29, 85

[45] Intel. **Memory Performance in a Nutshell**. https://software.intel.com/en-us/articles/memory-performance-in-a-nutshell. Accessed: 26-09-2019. 30, 31

[46] University of New Mexico: Electrical and Computer Engineering Department. **Reducing Cache Miss Rate**. http://ece-research.unm.edu/jimp/611/slides/chap5_3.html. Accessed: 17-09-2019. 31

[47] Intel. **How Intel AVX2 Improves Performance on Server Applications**. https://software.intel.com/en-us/articles/how-intel-avx2-improves-performance-on-server-applications. Accessed: 17-09-2019. 40

[48] Intel. **The Intel Advanced Vector Extensions 512 (Intel AVX-512) Vector Length Extensions Feature on Intel Xeon Scalable Processors**. https://software.intel.com/en-us/articles/the-intel-advanced-vector-extensions-512-feature-on-intel-xeon-scalable. Accessed: 17-09-2019. 40

[49] **Intel CPU Xeon E5-2695v4**. https://ark.intel.com/content/www/it/it/ark/products/91316/intel-xeon-processor-e5-2695-v4-45m-cache-2-10-ghz.html, 2019. Accessed: 17-09-2019. 40, 54

[50] Intel. **Race Conditions**. https://software.intel.com/en-us/blogs/2013/01/06/benign-data-races-what-could-possibly-go-wrong. Accessed: 17-09-2019. 45

[51] NVIDIA. **Branching and Divergence**. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#branching-and-divergence. Accessed: 17-09-2019. 48

[52] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. **Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU**. *ACM SIGARCH computer architecture news*, **38**(3):451–460, 2010. 54

[53] **Intel Vtune Amplifier**. https://software.intel.com/en-us/vtune-amplifier-help-ipc, 2019. Accessed: 17-09-2019. 54

[54] **Nvidia Tesla P100**. https://www.nvidia.com/en-us/data-center/tesla-p100/, 2019. Accessed: 17-09-2019. 55

[55] **CUDA PTX-ISA**. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, 2019. Accessed: 17-09-2019. 55

[56] Intel. **MPMD Launch Mode**. https://software.intel.com/en-us/mpi-developer-guide-linux-mpmd-launch-mode. Accessed: 17-09-2019. 57

[57] **od - Linux manual page**. http://man7.org/linux/man-pages/man1/od.1.html, 2019. Accessed: 17-09-2019. 64

[58] R. Lamb, M. Crossley, and S. Waller. **A fast two-dimensional floodplain inundation model**. In *Proceedings of the Institution of Civil Engineers-Water Management*, **162**, pages 363–370, 2009. 65

[59] B. F. Sanders, J. E. Schubert, and R. L. Detwiler. **ParBreZo: A parallel, unstructured grid, Godunov-type, shallow-water code for high-resolution flood inundation modeling at the regional scale**. *Advances in Water Resources*, **33**(12):1456–1467, 2010. 65

[60] P. Micikevicius. **3D finite difference computation on GPUs using CUDA**. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pages 79–84. ACM, 2009. 66

[61] T. Sourouri, M.and Gillberg, S. Baden, and X. Cai. **Effective multi-GPU communication using multiple CUDA streams and threads**. In *ICPADS*, pages 981–986. IEEE, 2014. 66

[62] D. Jacobsen, J. Thibault, and I. Senocak. **An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters**. In *AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, page 522, 2010. 66

[63] G. Petaccia, F. Leporati, and E. Torti. **OpenMP and CUDA simulations of Sella Zerbino Dam break on unstructured grids**. *Computational Geosciences*, **20**(5):1123–1132, 2016. 67

[64] M. De La Asunción et al. **An MPI-CUDA implementation of an improved Roe method for two-layer shallow water systems**. *Journal of Parallel and Distributed Computing*, **72**(9):1065–1072, 2012. 67

[65] Dimitri Komatitsch et al. **High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster**. *Journal of computational physics*, **229**(20):7692–7714, 2010. 67

[66] M. Garcia-Gasulla et al. **Load balancing of an mpi parallel unstructured cfd code using dlb and openmp**. *Submitted to International Journal of HPC Applications*, 2017. 68, 85

[67] nvidia. **Unified Memory in CUDA 6**. https://devblogs.nvidia.com/unified-memory-in-cuda-6/. Accessed: 17-09-2019. 75

[68] **The Zoltan Library Domain Decomposition and Load Balancing Algorithms**. http://www.cs.sandia.gov/Zoltan/. Accessed: 17-09-2019. 83, 84