Master in High Performance Computing

Analysis of OpenFOAM performance obtained using modern C++ parallelization techniques

Supervisor(s):
Giovanni Stabile,
Filippo Spiga,
Matthew Martineau

Candidate:
Giulio Malenza

8th edition
2021–2022

To my family.

Abstract

Current trends in high performance computing (HPC) include the use of Graphics Processing Units (GPUs) as massively parallel co-processors coupled with CPUs to accelerate the solution of complex physics and engineering problems like computational fluid dynamics (CFD).

OpenFOAM is a popular open-source CFD parallel software used by scientists and engineers worldwide. Several attempts have been performed to adapt the codebase to run on GPUs, with variable successes. Recently NVIDIA, CINECA and ESI-OpenCFD have collaborated to accelerate solver computation in OpenFOAM using the NVIDIA AmgX linear solver library. Other sections of the code still bounded by CPU can quickly become a limiting factor in achieving good end-to-end performance.

The main objective of this project will be to explore the use of modern ISO C++ parallel constructs to accelerate well-defined kernels extracted from the main application, in particular routines associated to operators evaluation.

Contents

Abstract **v**

1 Introduction **1**

1.1 Background . . . 1

1.2 Project overview . . . 2

2 Standard ISO C++ **5**

2.1 Concurrency in C++ . . . 5

2.2 STL Algorithms . . . 6

2.2.1 Class ”Iteration & Transform” . . . 7

2.2.2 Class ”Reductions” . . . 8

2.2.3 Class ”Search” . . . 9

2.2.4 Class ”Memory Movement & Initialization” . . . 9

2.2.5 Class ”Reorder” . . . 10

2.3 Parallel STL Algorithms . . . 10

2.4 Support in NVIDIA HPC SDK . . . 11

2.4.1 NVIDIA Unified Memory . . . 12

3 OpenFOAM **13**

3.1 Domain Discretization . . . 14

3.2 Gauss-Green Gradient Computation . . . 17

4 Implementation **21**

4.1 Previous works . . . 21

4.2 Identification of `simpleFOAM` hot-spots . . . 21

4.3 Code Porting and Refactoring . . . 23

4.3.1 Routine `SurfaceIntepolationScheme` . . . 23

4.3.2 Routine `gradf` . . . 24

4.3.3 Routine `CorrectBoundaryConditions` . . . 33

4.4 OpenFOAM gradient mini-app . . . 35

4.5 Profiling a GPU application . . . 36

4.6 Tuning data movement with explicit pre-fetching . . . 36

5 Results **39**
5.1 Mini-app input cases 39
5.2 Computing platform 39
5.3 Results obtained 40
5.3.1 Performance running `Mesh 16M` input case 41
5.3.2 Performance running 32M mesh input case 44
5.3.3 Performance running 64M mesh input case 45
5.4 General considerations 46

6 Conclusions **49**

A Execution Times for **51**
A.1 Global Execution Times for 51
A.2 Execution Times for `Mesh 16M` 52
A.3 Execution Times for `Mesh 32M` 52
A.4 Execution Times for `Mesh 64M` 53

Bibliography **53**

List of Figures

2.1 Simplified timeline of C++ standard development from 1980 to the present day. 5

3.1 Example of structured and unstructured meshes 15
3.2 Owner and Neighbour 16
3.3 *diag*, *lower* and *upper* lists of a OpenFOAM matrix 17

4.1 Partial Kcachegrind output of the gradient computation 22
4.2 List of List 26
4.3 Optimized list of list 27
4.4 Sorting list 27
4.5 Start owner list 28
4.6 Start list creation 31
4.7 `Weight first cycle` without pre-fetching tools 37
4.8 `Weight first cycle` with pre-fetching tools 38
4.9 Pre-fetching section 38

5.1 The NVIDIA Arm HPC Developer Kit platform. 40

List of Tables

5.1 Mesh specifications 39
5.2 Explanations of various NVTX tags acronyms used. 41
5.3 speed-up first iteration `Mesh 16M` (HYBRID = 10 MPI + × 8 CPU threads) . 42
5.4 speed-up second iteration `Mesh 16M` (HYBRID = 10 MPI × 8 threads) 43
5.5 speed-up first iteration `Mesh 32M` (HYBRID = 10 MPI × 8 threads) 44
5.6 speed-up second iteration `Mesh 32M` (HYBRID = 10 MPI × 8 threads) 45
5.7 speed-up first iteration `Mesh 64M` (HYBRID = 10 MPI × 8 threads) 45
5.8 speed-up second iteration `Mesh 64M` (HYBRID = 10 MPI × 8 threads) 46
5.9 Full speed-ups of `GRAD` during the First Iteration (HYBRID = 10 MPI × 8 threads) 47
5.10 Full speed-ups of `GRAD` during the Second Iteration (HYBRID = 10 MPI × 8 threads) 47

A.1 Global Execution Times for first iteration (HYBRID = 10 MPI × 8 threads) . 51
A.2 Global Execution Times for second iteration (HYBRID = 10 MPI × 8 threads) 51
A.3 Times first iteration `Mesh 16M` (HYBRID = 10 MPI × 8 threads) 52
A.4 Times second iteration `Mesh 16M` (HYBRID = 10 MPI × 8 threads) 52
A.5 Time first iteration `Mesh 32M` (HYBRID = 10 MPI × 8 threads) 52
A.6 Times second iteration `Mesh 32M` (HYBRID = 10 MPI × 8 threads) 53
A.7 Times first iteration `Mesh 64M` (HYBRID = 10 MPI × 8 threads) 53
A.8 Times second iteration `Mesh 64M` (HYBRID = 10 MPI × 8 threads) 53

Listings

2.1 for_each algorithm . . . 7
2.2 transform algorithm(unary operator) . . . 7
2.3 transform algorithm(binary operator) . . . 8
2.4 reduce algorithm . . . 8
2.5 count algorithm . . . 8
2.6 search algorithm . . . 9
2.7 find algorithm . . . 9
2.8 copy algorithm . . . 9
2.9 generate algorithm . . . 10
4.1 dotInterpolate cycle . . . 23
4.2 dotInterpolate cycle in parallel . . . 23
4.3 weights cycle . . . 24
4.4 forAll macro syntax . . . 24
4.5 weights cycle in parallel . . . 24
4.6 Gradient field constructor . . . 25
4.7 Field constructor . . . 25
4.8 Field constructor in parallel . . . 25
4.9 First cycle in the gradf . . . 26
4.10 Sorting lists . . . 27
4.11 Start list creation . . . 28
4.12 OwnerList creation . . . 29
4.13 OwnerList and OwnerStart allocation . . . 29
4.14 first cycle gradient in parallel . . . 30
4.15 Second cycle gradient . . . 30
4.16 List creation . . . 31
4.17 Second cycle in parallel . . . 32
4.18 Third cycle . . . 32
4.19 Third cycle in parallel . . . 33
4.20 Correction boundary condition . . . 33
4.21 mag function . . . 33
4.22 Parallel mag function . . . 33

4.23 PatchInternalField function 34
4.24 PatchInternalField function in parallel 34
4.25 Deltacoeff function 34
4.26 Deltacoeff function in parallel 34
4.27 Test Application 35

Chapter 1

Introduction

1.1 Background

Current trends in High Performance Computing (HPC) include the use of accelerators, such as graphics processing units (GPUs), as co-processors coupled with Central Processing Units (CPUs) in order to accelerate numerical operations common to different applications. From the hardware point of view, technology providers such as NVIDIA, Intel and AMD are focused on developing and improving these new technologies adding in each generation new functionalities such mixed precision, better memory hierarchies and tensor core for high throughput. From the software point of view, many efforts have been made to make the best use of these new technologies and make them easy for users to handle. GPUs were originally developed to accelerate the rendering of 3D graphics. Graphical rendering consists mainly of simple mathematical operations performed on large amounts of data in SIMD-like fashion. Over the years GPUs became more flexible and programmable. In particular, thanks to the introduction of programming models such as CUDA, developers were able to write code and execute it on the GPUs by using a very pragmatic and intuitive programmable API.

OpenFOAM is one of the most popular open-source Computational Fluid Dynamics (CFD) software due to its flexibility and other optimized features. It is manly written in C++ and it is based on the finite volume method [23]. Any OpenFOAM simulation consists manly in two parts:

- In the ***assembly*** part, linear systems of algebraic equations are derived from the discretization of a set of Partial Differential Equations (PDE).
- In the ***solver*** part, the solution of these systems is obtained using numerical methods.

Discretizing and solving these equations is not only difficult from the theoretical point of view but also from a computational one. Previous works [1] show that the main bottleneck on the OpenFOAM simulation resides in the solver part for a broader set of use-cases. Many efforts have been devoted to improve the performance of the solver part [19, 20] as well as non-solver

parts [25]. However, also the assembly part requires time and resources in a general end-to-end OpenFOAM simulation. It has been observed that assembly can too become a dominant aspect of a simulation.

One of the main features that makes OpenFOAM easily portable on different CPU architectures is the portable nature of the programming language (C++). Therefore, the challenge is to modify the code-base to run on GPUs without limiting its portability and without introducing many duplicated specialized code for different engines. Despite all the recent developments, porting a code to the GPUs is rarely a quick and straightforward process. Many GPU vendors develop libraries, programming languages and frameworks to make it easier for developers to perform such task with ease of mind. However, many of these technologies and paradigms limit code portability to specific architectures. It is not uncommon to have to rewrite or modify certain parts of the code. Starting from the C++11, first set of parallel construct and concurrency topics have been introduced as part of the standard C++ language. Further iterations of the C++ standards (C++14, C++17, C++20) have progressively introduced new concepts and refined the specification. From the Standard C++17 the algorithms of the Standard Template Library (STL) were extended in order to express and exploit parallel computation, for example by introducing the so called *execution policies* [6].

1.2 Project overview

The purpose of this work is to explore the use of modern ISO C++ parallel constructs to accelerate OpenFOAM computation. By continuing to use standard C++, we are allowed to unlock extra parallelization opportunities (multi-core CPUs and GPU offload) without limiting its portability. The novelty of the developed Proof-of-Concept consists in unlocking multi-core and GPU execution with just a compiler flag switch, avoiding adding extra parallel programming models (such as OpenMP, Thrust or SYCL) and avoiding restrict the execution of OpenFOAM simulations to specific accelerated platforms. The code is *just* modern C++, it runs on any CPU using any standard-compliant open-source or commercial C++ compiler. Offload capabilities are provided by the compiler without requiring developer/user special technical knowledge.

Chapter 2 introduces the theoretical concepts of OpenFOAM relevant to this work. In particular the domain discretization and the Gauss-Green gradient evaluation are explained.

Chapter 3 is devoted to describing how the implementation was made. Starting with a brief description of the work previously done to optimise OpenFOAM, the chapter dives into the selected kernels. Both in the original and in the modified version (where possible) in order to clearly display the changes needed to unlock opportunities for parallel execution.

Chapter 4 introduces first the platform used to run the simulations and then analyzes the results obtained running several kind of experiments: pure MPI (original) execution, hybrid MPI + multi-thread, MPI + GPU offload

Chapter 5 provides a summary of the achieved results, the merits and shortcomings of the Proof-of-Concept developed and discuss few extra ideas to expand this research.

Chapter 2

Standard ISO C++

The history of the standard library C++ starts in the 1980s and continues to the present day. Figure 2.1 summarizes the main developments over the years. The first version of the C++ standard (C++98) consisted of three components: a library to handle the I/O data, a string library to manipulate strings and the Standard Template Library (or STL). The Standard Template Library is a collection of containers and algorithms that communicate with each other via iterators.

In 2005 a technical report (TR1) introduced many new features which were ratified and made official in the C++11 standard. In the standard C++11 the *memory consistency model* was introduced describing the allowed behavior of multi-threaded programs executing with shared memory [6]. The C++14 was an update of the C++11, the C++17 introduced new features such as *parallel STL algorithms*.

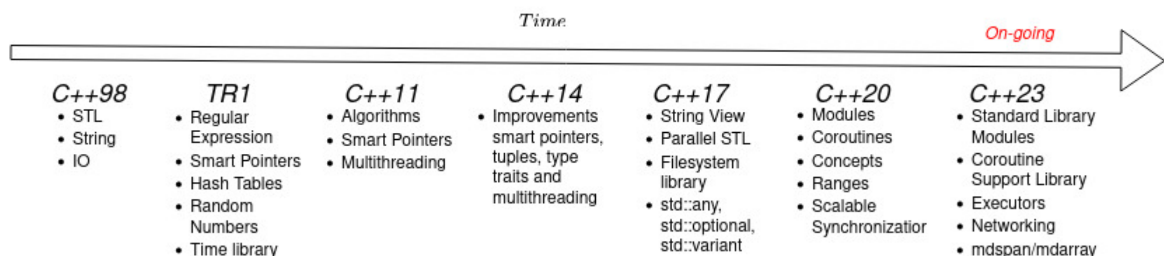


Figure 2.1: Simplified timeline of C++ standard development from 1980 to the present day.

2.1 Concurrency in C++

A task can be defined as an activity that could be performed concurrently with other activities. In this contest the word concurrency can be identified as the execution of several tasks simultaneously. Traditionally the difference between *parallelism* and *concurrency* is a matter of intent. Both in practical terms mean taking advantages from the hardware capabilities

performing multiple tasks simultaneously. However, the former is more related to increasing performances during the processing of large amounts of data, while the latter refers to increasing performances by separating a large job into smaller tasks that can be executed simultaneously. In this work the words *concurrency* and *parallelism* will have the same meaning, i.e. performing a calculation simultaneously using multiple threads.

Before the standard C++11 the system only knew about one control flow which guaranteed the programmer that the observed behaviour of the program was the exact consequence of the instructions executed in the source code [7]. The memory model introduced in C++11 gave the user the possibility of executing code in parallel using threads. To do this, however, developers had to establish rules to avoid corrupted operations.

2.2 STL Algorithms

Originally the Standard Template Library(STL) was mainly composed of three components: the **containers**, objects that store a collection of other objects[1], **algorithms** that run on the containers and **iterators** that allow algorithms to run over the container elements.

Containers can be classified as sequential or associative. Sequential containers are: `vector`, `array`, `deque`, `forward_list` and `list`. Associative containers are containers based on `key-value pairs`. They can be ordered or unordered. `Set`, `map`, `multiset` and `multimap` are ordered containers while `unordered_set`, `unordered_map`, `unordered_multiset` and `unordered_multimap` are unordered. Finally, there are container adapters which provide a different interface for sequential containers, the most important are: `stack`, `queue`, `priority_queue`.

Iterators are the glue between containers and algorithms. Specifically they are objects that represent a pointer which is able to point some element in a range of elements. It has the ability to iterate over the container using operators like increment(++), decrement($--$) and dereference($*$)[2]. There are different iterator categories, the classic ones are: *input iterator*, *output iterator*, *forward iterator*, *bidirectional iterator*, *random-access iterator.*

Input and Output iterators are the simplest and limited iterators, they can move forward with the increment operator and they are used for input and output operations. The forward iterators are both input and output iterators, so they are able to do input and output operations. They can iterate over a container with the increment operator(++) and, if they point to a class object, they can refer to one of its members with $\rightarrow$ operator. Bidirectional iterators are similar to the forward with the ability to go backward using the decrement operator($--$). As the forward iterators they can be compared using operators($==$ or $!=$).

[1] `https://cplusplus.com/reference/stl/`
[2] `https://en.cppreference.com/w/cpp/iterator`

Finally, Random-access iterators implement all the functionality of the bidirectional iterators and, moreover, they can access ranges non-sequentially using the operator([]). They can be compared with the following operators: $==, !=, <, <=, >, >=$.

Originally, C++11 introduces around 80 algorithms into the Standard Template Library[3]. With the evolution of the standard, other algorithms were added over time and now there are more than 100 algorithms implemented.

2.2.1 Class "Iteration & Transform"

This class of algorithms allows to *iterate* or *transform* container elements. A key and popular algorithm in this class is `for_each`[4]. This algorithm applies a function pointer to each element in the container over which it iterates. Its behaviour is similar to the following one 2.1:

```
1 template<class InputIt, class Function>
2 constexpr Function for_each(InputIt first, InputIt last, Function f)
3 {
4     for(; first!=last; ++fist){
5         f(*first);
6     }
7     return f;
8 }
```

Listing 2.1: for_each algorithm

Another important algorithm, used many times in this work is the `transform`[5]. This algorithm applies a given operation to a range and returns the result in another range, keeping the original container elements ordered. There are two variants of this algorithm depending on the type of operations used by the user. Indeed the operator could be unary, taking only argument, or binary, taking two arguments. In the first case the behaviour of the algorithm is similar to the following code (2.2):

```
1 template<class InputIt, class OutputIt, class UnaryOpn>
2 OutputIt transform(InputIt first1, InputIt last1,
3                     OutputIt d_first, UnaryOpn unary_op)
4 {
5     while (first1 != last1)
6         *d_first++ = unary_op(*first1++);
7
8     return d_first;
9 }
```

Listing 2.2: transform algorithm(unary operator)

In the second case, the algorithm behaviour is like the following 2.3:

[3] `https://en.cppreference.com/w/cpp/algorithm`
[4] `https://en.cppreference.com/w/cpp/algorithm/for_each`
[5] `https://en.cppreference.com/w/cpp/algorithm/transform`

```
template<class InputIt, class InputIt2,
        class OutputIt, class BinaryOp>
OutputIt transform(InputIt first1, InputIt last1,
                    InputIt2 first2, OutputIt d_first,
                    BinaryOp binary_op)
{
    while (first1 != last1)
        *d_first++ = binary_op(*first1++,*first2++);

    return d_first;
}
```

Listing 2.3: transform algorithm(binary operator)

The algorithms `transform_reduce`, `transform_inclusive_scan` and `transform_exclusive_scan` apply respectively a transform followed by a reduce operation in the first case, an inclusive scan in the second case, and an exclusive scan in the last case. In particular inclusive and exclusive scans are important for computing in parallel common algorithm such as prefix sum.

2.2.2 Class "Reductions"

These algorithms usually are useful to perform some type of summary operations. For example the algorithm `count`[6] can be used to count the number of elements that satisfy some condition in a given range.

One of the most important algorithms in this class is the `reduce`. A possible sketch of the implementation of this algorithm is the following 2.4:

```
  template<class InputIt, class Tp, class BinaryOp>
    Tp reduce(InputIt first, InputIt last, Tp init, BinaryOp binary_op)
    {
      for (; first != last; ++first)
      init = binary_op(init, *first);
      return init;
    }
```

Listing 2.4: reduce algorithm

Another example of algorithm in this class is the `count`:

```
template<class InputIt, class T>
typename iterator_traits<InputIt>::difference_type
    count(InputIt first, InputIt last, const T& value)
{
    typename iterator_traits<InputIt>::difference_type ret = 0;
    for (; first != last; ++first)
        if (*first == value)
```

[6]https://en.cppreference.com/w/cpp/algorithm/count

```
8             ++ret;
9     return ret;
10 }
```

Listing 2.5: count algorithm

2.2.3 Class "Search"

In this class the most relevant algorithm for this work are the `search` and the `find` algorithms. 2.6 look for the first occurrence of a range in another range. Instead 2.7 returns an iterator to the first element in a range of elements that satisfies the condition equal to a specific value.

```
1 template<class ForwardIt1, class ForwardIt2>
2 constexpr ForwardIt1 search(ForwardIt1 first, ForwardIt1 last,
3                             ForwardIt2 s_first, ForwardIt2 s_last)
4 {
5     while (1) {
6         ForwardIt1 it = first;
7         for (ForwardIt2 s_it = s_first; ; ++it, ++s_it) {
8             if (s_it == s_last) return first;
9             if (it == last)  return last;
10            if (!(*it == *s_it)) break;
11        }
12        ++first;
13    }
14 }
```

Listing 2.6: search algorithm

```
1 template<class InputIt, class T>
2 constexpr InputIt find(InputIt first, InputIt last, const T& value)
3 {
4     for (; first != last; ++first) {
5         if (*first == value) {
6             return first;
7         }
8     }
9     return last;
10 }
```

Listing 2.7: find algorithm

2.2.4 Class "Memory Movement & Initialization"

In this class the `copy` algorithm can be found. This algorithm simply copies all the elements in a range into another range. Its structure is the following 2.8.

```
1 template<class InputIt, class OutputIt>
2 OutputIt copy(InputIt first, InputIt last,
```

```
3                OutputIt d_first)
4 {
5     for (; first != last; (void)++first, (void)++d_first) {
6         *d_first = *first;
7     }
8     return d_first;
9 }
```

Listing 2.8: copy algorithm

Another particularly interesting algorithm in this class is the `generate`[7]. It assigns each element in a range a value generated by a given function object 2.9.

```
1 template <class ForwardIterator, class Generator>
2   void generate ( ForwardIterator first, ForwardIterator last, Generator gen )
3 {
4   while (first != last) {
5     *first = gen();
6     ++first;
7   }
8 }
```

Listing 2.9: generate algorithm

Finally in this class of algorithms there is the `fill`, which assigns a specific value to each element in a range. It is useful for example during the container initialization.

2.2.5 Class "Reorder"

In this class there are algorithms like `sort`, which sorts elements in a container, `rotate`, which produces a left rotation on a range of elements and other such algorithms.

2.3 Parallel STL Algorithms

When the standard C++17 was introduced, many algorithms of the Standard Template Library were extended in order to be able to invoke the `execution policy`[8]. By specifying an execution policy, the target algorithm can be executed in multiple ways: sequentially, sequentially with the vectorization, parallel and parallel with the vectorization.

Originally, in C++17, the execution policies were three: sequenced, parallel and parallel unsequenced. Then in the standard C++20 the policy unsequenced was introduced. These policies are defined in the header <execution>:

- `std::execution::seq` → The sequenced policy (since C++17). This policy forces the execution of an algorithm to run sequentially on the CPU.

[7] https://en.cppreference.com/w/cpp/algorithm/generate
[8] https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t

- `std::execution::unseq` → The unsequenced policy (since C++20). With this policy the calling algorithm is executed using vectorization on the calling thread.
- `std::execution::par` → The parallel policy (since C++17). This policy tells the compiler that the algorithm could be run in parallel.
- `std::execution::par_unseq` → The parallel unsequenced policy (since C++20). This policy allows the algorithm to be run in parallel on multiple threads each able to vectorize the calculus.

The first important thing to note is that parallel execution policies allow the system to perform the calculation with multiple threads, but this permission is not a requirement. The standard C++ leaves the compiler great freedom in deciding if, when and how to run the algorithms in parallel.

Another important thing is that the parallel algorithm does not automatically protect the user from data races and deadlocks. The user must be sure that by changing the execution policy, typically from sequential to parallel, no deadlocks or data races are introduced. As seen in the previous sections, atomic operations can be useful to manage these situations.

2.4 Support in NVIDIA HPC SDK

The NVIDIA HPC SDK is a comprehensive toolbox for GPU accelerating HPC modeling and simulation applications. It includes the C, C++, and Fortran compilers, libraries, and analysis tools necessary for developing HPC applications on the NVIDIA platform.

The NVIDIA C++ `nvc++` compiler is able to compile and generate an executable that can run on GPUs just by adding in the command-line the option `-stdpar`. If `-stdpar=multicore` is used then the generated code than execute multi-threading on CPU. Extra compiler flags can tune CUDA runtime version and target compute capability. The NVIDIA HPC SDK allows to mix various programming languages (e.g. standard ISO C++ for GPU offloading with CUDA API calls or CUDA kernels).

NVIDIA has demonstrated in recent studies [14, 15] that cleaner code can be written and performance on GPU can be improved on par with other more explicit programming models such as OpenACC or CUDA. An example of this is *Lulesh*, a hydrodynamics mini-app from Lawrence Livermore National Laboratory (LLNL) written in C++ and then modified to use standard parallelism with NVC++ [13]. Analyses show that performance obtained using standard C++ parallelism with offload on an NVIDIA A100 GPU are around 13X faster than using the OpenMP offload on an AMD EPYC 7742.

2.4.1 NVIDIA Unified Memory

One of the key points in developing code for GPUs is how data movement between CPU and GPU takes place. CUDA and OpenACC have specific directives that allow the user to manage this data movement. The ISO C++ does not include any construct or concept or operation to manage explicitly data movement between two distinct computing devices. Allowing so would break the C++ memory model and the C++ abstract machine.

In order to allow programming GPU using standard C++ parallel constructs, the data movement management has to be entirely *transparent* and self-managed. To do so, `nvc++` assume CUDA Unified Memory [17] enabled by default.

First introduced in CUDA 6, Unified Memory marks a turning point on how program NVIDIA GPUs. Unified memory creates a single *Unified Virtual Address* space accessible from any processor in a system [12] and allow to allocate a pool of managed memory that is shared between CPU and GPU. Data allocated with this approach can be read and written from code running on CPUs or GPUs. The NVIDIA CUDA driver and NVIDIA GPU hardware are responsible, via a page-fault mechanism, to automatically trigger data migration based on its use. The way it works is simple: if a kernel running on the GPU accesses a page that is not resident in its memory then it faults allowing the page to be automatically migrated to the GPU memory on-demand.

Only data dynamically allocated on CPU can be marked *managed*. Hence data allocated on the CPU or GPU stack cannot be automatically moved. This means that any de-referenced pointer and any referenced object in a parallel C++ algorithm must refer to the CPU heap. In the other cases the result is a memory violation in GPU code. For example:

- use `std::vector` since it is allocated dynamically.
- do not use `std::array` since it is allocated on the CPU stack.

Chapter 3

OpenFOAM

OpenFOAM [18] is one of the most popular computational fluid dynamics (CFD) software due to its flexibility and other optimization features. It is written mainly in C++ and it is based on the finite volume method (FVM). This method is used to representing and evaluate general partial differential equations (PDE) in the form of algebraic equations.

The Navier-Stokes equations are among the most important partial differential equations of fluid dynamics. In the case of incompressible flow these equations can be written as follows 3.1:

$$\begin{cases} \nabla \cdot \vec{v} = 0 \\ \frac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \nabla)\vec{v} = \nu \Delta \vec{v} - \frac{1}{\rho}\nabla p + \vec{g} \end{cases} \tag{3.1}$$

The two equations of the previous system represent the conservation of mass and momentum respectively. In this equations: $\vec{v}$ is the velocity vector, p is the pressure field, $\vec{g}$ represents body accelerations acting on the continuum, for example gravity, inertial accelerations, electrostatic accelerations, and so on, ρ is the fluid density, ν is the kinematic viscosity.

Considering the second equation, the first term on the left hand side is the transient term and represents the time variation. The second term is the convective term. The first term on the right hand side is the diffusion term. The second term is the pressure gradient term and the last term is an external source term.

To solve the previous equations OpenFOAM first discretizes them in a set of algebraic equations and then solves them using a numerical method.
Since the aim of this work is to try to understand how to apply the parallelization techniques of the new standard C++, seen in chapter 2, to the assembly phase of OpenFOAM, it may be useful to explain briefly the purpose of this phase and what is done in it. The concepts that will be shown later in this chapter are explained in more detail in the books [21, 23] and in the thesis [22].

As a starting point, it can be said that the numerical solution of a partial differential

equation consists of finding the values of a dependent variable ϕ at specified points called grid elements. Therefore the main goal is to replace the continuous exact solution of the considered equation with discrete values in those nodes. The process of converting the equation into a set of algebraic equations is called discretization process. The discrete values of ϕ are typically computed by solving this set of algebraic equations. This phase usually is called solver phase. Once the values of ϕ are discovered, data are processed to extract any relevant information.

3.1 Domain Discretization

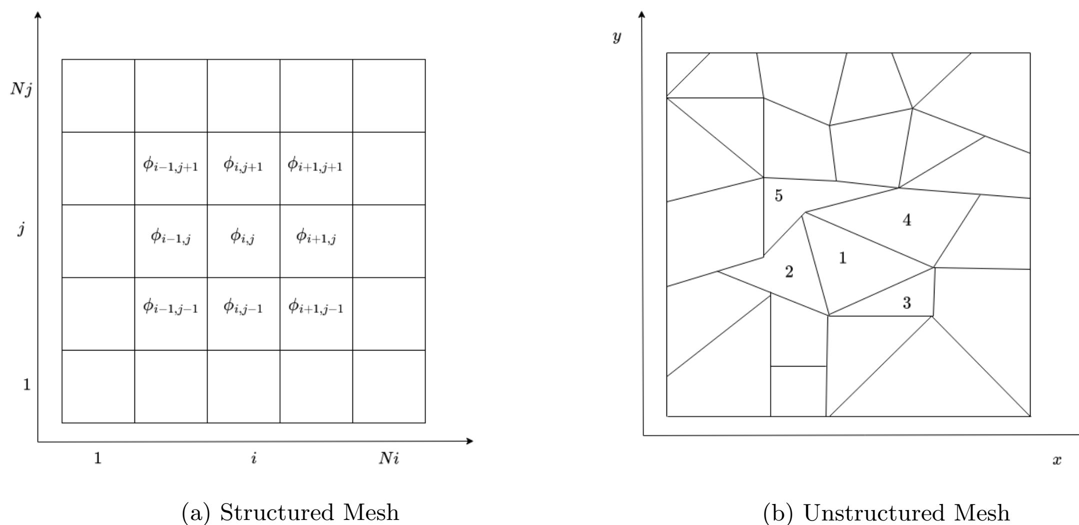
Typically the result of the geometric discretization is a mesh on which the algebraic equations are first derived and then solved. This mesh consists of a discrete set of non-overlapping elements that completely fill the entire physical domain. Since the finite volume method is chosen as discretization approach, the elements of a mesh can be of arbitrary convex polyhedral shapes. In general they are defined by a set of vertices, which represent locations in one, two or three dimensional space. However the elements can be defined also in terms of their bounded faces. Usually the faces are subdivided in the interior faces, which connect two elements, and in boundary faces, that coincide with the boundary of the domain.
The process from which algebraic equations are derived consists in two parts, usually called local and global assembly. During the local assembly the integration of the equations over each elements is performed. After this in the global assembly the system of linear equations can be construct using the previous contributes. In order to complete local and global assembly, topological information about elements, faces and vertices are needed. This information is represented in terms of connectivity lists. There are element, face and vertex connectivity. The first relates the local assembly to the global assembly. In particular, it relates an element to its neighbouring elements, its bounding faces and its vertices. The second stores information about the elements that share a face. The last relates the lists of elements and faces that share a vertex.
In general, meshes fall into two categories: structured and unstructured. An example of structured mesh can be seen in figure 3.1a. In this case elements are identified by their three local indices (i, j, k). This type of mesh has many computational advantages but is limited by the geometry of the problem under consideration. On the other hand, unstructured meshes, figure 3.1b, allow more complex and general problems to be studied, but at the cost of higher computational activity.
Due to the geometric symmetries of the domain under consideration, structured meshes do not need to define connectivity lists. In particular local indices are mapped into global indices

through the following relations:

$$
\begin{array}{ll}
(i,j,k) \rightarrow n & \\
(i+1,j,k) \rightarrow n+1 & (i-1,j,k) \rightarrow n-1 \\
(i,j+1,k) \rightarrow n+Ni & (i,j-1,k) \rightarrow n-Ni \\
(i,j,k+1) \rightarrow n+Ni*Nj & (i,j,k-1) \rightarrow n-Ni*Nj
\end{array}
$$

The previous relations allow the global matrix to be constructed from the local indices.

For unstructured meshes the situation is more complex. In these types of meshes the ele-



(a) Structured Mesh

(b) Unstructured Mesh

Figure 3.1: Example of structured and unstructured meshes

ments can be of arbitrary polyhedral shapes. For this reason it is not possible define indices that map local contributions into global ones and it is necessary to proceed by defining connectivity lists.

A polyhedron is a three-dimensional solid that can be represented as a set of plane polygons, joined at their edges. To handle meshes composed by general polyhedron OpenFOAM uses the so called face-addressing storage. It is based on the use of a set of lists (array) to storage points, faces and elements.

The list of points contains the vertices of the mesh, a set of three dimensional spatial coordinates defined as vectors. The faces are represented by a list of vertex labels that are order such that two adjacent points in the list are connected by an edge. Moreover this list is organized in a way that all the internal faces appear first in the list followed by the faces related to the boundaries. Lists of boundary faces are also named patches. Finally an element or cell of the mesh is defined by a list of indices where the first one is the number or faces composing that element and the others are the faces indices for that element.

All the relevant information about the mesh, represented by the previous lists, are contained in the folder *constant/polyMesh*. In this folder there are the following files [18]:

- points: the list of vectors describing the cell vertices.
- faces: the list of faces, each face is identified by a list of indices to vertices in the points list.
- owner: a list of owner cell labels, the index of entry relating directly of the face.
- neighbour: a list of neighbour cell labels.
- boundary: a list of patches.

The owner and neighbour lists are particularly important for this work. Figure 3.2 represents owner and neighbour cell labels for the blue face. It is worth noting that the normal vector to the surface is oriented from the owner to the neighbour. This is standard and applies to each face. Another thing worth noting is that for each face, the owner element has a smaller cell labels than the neighbour element.
The owner is a list of size the number of total faces. Each entry of this list represent the owner cell label of that face. This means that if the first entry has value 1 the owner of face 0 is the cell 1, if the second entry has value 5 then the owner of face 1 is the cell with label 5 and so on.
The neighbour is a list where each entry represents the cell label neighbour of that face. It is important to note that each face has an owner cell label but not every face has an neighbour cell label. Indeed boundary faces have not neighbours. For this reason the size of neighbour list is equal to the number of neighbour faces.

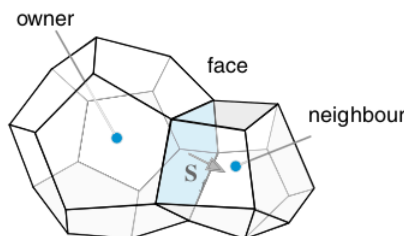


Figure 3.2: Owner and Neighbour

OpenFOAM managed the mesh using a class called *GeometricField<Type,... >*. This class stores data structure based on the characteristics of the mesh. In OpenFOAM a field is a list of tensors. In general a geometricField object is renamed using typedef C++ syntax as follows :

- *volField<Type >*: a field defined at cell centres;
- *surfaceField<Type>*: a field defined on cell faces;
- *pointField<Type>*: a field defined on cell vertices;

The Type which template all these classes can be: scalar, vector, tensor, symmTensor, tensorThird and a symmTensorThird. Specifing the template a volField of scalar type is also called volScalarField, a surfaceField becomes a surfaceScalarField and a pointField a pointScalarField. The same holds with vector, tensor and so on.
OpenFOAM uses the face addressing storage not only to handle the mesh but also to storage the matrices coefficients. In general matrices obtained from the discretization of the partial differential equations with the finite volume method are sparse. This means that they contains a lot of zero coefficients. There is no a precise definition of how many non-zero values a matrix must have to be defined as sparse. However, a matrix is commonly defined as sparse if it has a number of non-zeros values equal, in order of magnitude, to the number of rows or columns in the matrix.
From a computational point of view, a sparse matrix has many advantages when the problem size is large. In fact, instead of storing the entire matrix which would require the use of a large amount of memory, only the indices of the non-zero elements o interest can be saved. Over the years, many techniques have been developed to store sparse matrices, i.e. coordinate list (COO), compressed sparse row (CSR), compressed sparse column (CSC) and so on. As said before, OpenFOAM uses the face-addressing storage to matrix storage which is based on the owner and neighbour lists.
To store a general matrix OpenFOAM uses five lists: *diag*, *lower*, *upper*, *lowerAddr* and *upperAddr*. *LowerAddr* and *upperAddr* are the owner and neighbour lists. As shown in figure 3.3 the diagonal part of the matrix is stored in the *diag* list, the lower part is stored in the *lower* list and the upper part is stored in the *upper* list.

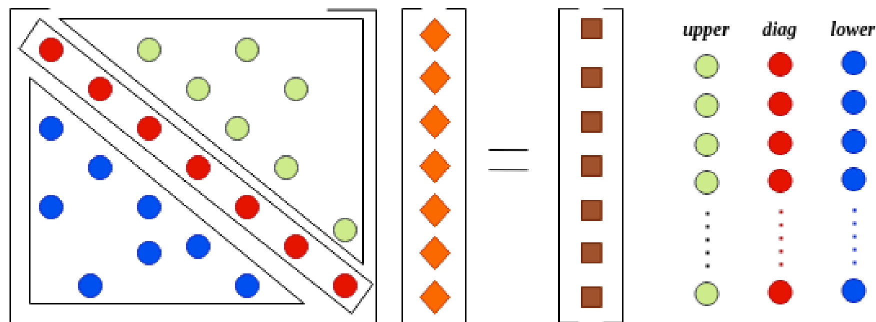


Figure 3.3: *diag*, *lower* and *upper* lists of a OpenFOAM matrix

3.2 Gauss-Green Gradient Computation

During the dicretization of the Navier-Stokes equation one computation particularly expensive is the evaluation of the gradient of a field. Many schemes were developed to do this, chapter 9 of the book [21] explain these schemes in detail. Here only the method based on the Green-Gauss theorem will be shown. This scheme is relatively straightforward, it can be used for a different grids and topologies (structured and unstructured meshes).

The goal is to compute the following integral:

$$\overline{\nabla\phi_C} = \frac{1}{V_C}\int_{V_C}\nabla\phi\,dV. \tag{3.2}$$

Here C and V_C are respectively the centroid and the volume of the cell while ϕ is the quantity whose gradient is to be calculated. Using the divergence theorem[1] the previous integral is transformed into the following surface integral:

$$\overline{\nabla\phi_C} = \frac{1}{V_C}\int_{\partial V_C}\phi\,d\vec{S}. \tag{3.3}$$

Here dS is the outward pointing surface vector. Since the element is bounded by faces, this integral can be rewritten as the sum of discrete integrals over the faces:

$$\overline{\nabla\phi_C} = \frac{1}{V_C}\sum_{\partial V_C}\int_{face}\phi\,d\vec{S}. \tag{3.4}$$

The integral over a cell face can be approximated using a numerical integration method. For example, using the mid-point integration the previous integral is evaluated at the face centroid:

$$\overline{\nabla\phi_C} = \frac{1}{V_C}\sum_{f=nb(C)}\overline{\phi_f}\vec{S_f}. \tag{3.5}$$

To evaluate the last expression the value of the variable ϕ at the face centroid (ϕ_f) and the value and the direction of the surface vector $\vec{S_f}$ are needed.
To compute the value of the variable ϕ_f at the face centroid it is necessary to assume that the variation of ϕ between the centroid of the element C and the centroid of its neighbour element F, with which it shares the face, follows some profile. This introduces an approximation in the gradient evaluation. There are different methods to make this approximation. In the following a brief explanation of the main ones will be given, more details can be found in [21].
One of the most popular method on which this work is focused is the cell based linear interpolation method, it assumes a linear variation between ϕ_C and ϕ_F. Hence the value of the variable in the face centroid $\overline{\phi_f}$ can be calculated using the following expression:

$$\overline{\phi_f} = g_F\phi_F + g_C\phi_C \qquad with \quad g_C = 1 - g_F. \tag{3.6}$$

G_F and g_C are called weight factors and can be computed starting from the position vector $\vec{r}$ in this way:

$$g_C = \frac{\parallel \vec{r_F} - \vec{r_f} \parallel}{\parallel \vec{r_F} - \vec{r_C} \parallel} = \frac{d_{Ff}}{d_{FC}}, \qquad g_F = 1 - g_C = 1 - \frac{d_{Ff}}{d_{FC}}, \tag{3.7}$$

where d_{Ff} is the distance between the cell centroid F and the face centroid f and d_{FC} is the distance between the cell centroid F and the cell centroid C.

[1] `https://en.wikipedia.org/wiki/Divergence_theorem`

This approach to evaluate the variable ϕ at the face centroid is simple to implement and not require additional connectivity lists. Generally this method leads to a first order of accuracy. However if the segment connecting the centroids C and F intersects the face f in its centroid the method leads a second order of accuracy. This condition is verified only if the mesh is orthogonal and structured. A correction and an iterative process is usually required to achieve greater accuracy.
Another method with which the value of ϕ at the face centroid can be computed is the nodal averaging method. In this case the value ϕ_f is computed as the mean of the values at the vertices defining the surface. Since the values of ϕ at the vertices are unknown, they must be estimate before to compute the mean. These values can be estimated using the weighted average of the properties within the cells surrounding that node.
Assuming n the vertex, F_k the neighbouring cell node, $NB(n)$ the total number of cell node surrounding the vertex n and $\| \vec{r}_n - \vec{r}_{F_k} \|$ the distance between the vertex and the neighbouring cell elements the resulting equation is he following:

$$\phi_n = \frac{\sum_{k=1}^{NB(n)} \frac{\phi_{F_k}}{\|\vec{r}_n - \vec{r}_{F_k}\|}}{\sum_{k=1}^{NB(n)} \frac{1}{\|\vec{r}_n - \vec{r}_{F_k}\|}}. \tag{3.8}$$

Once the previous values are computed, the value at the face center can be estimated by averaging over the values in the vertices:

$$\phi_f = \frac{\sum_{k=1}^{nb(f)} \frac{\phi_{n_k}}{\|\vec{r}_{n_k} - \vec{r}_f\|}}{\sum_{k=1}^{nb(f)} \frac{1}{\|\vec{r}_{n_k} - \vec{r}_f\|}}. \tag{3.9}$$

Chapter 4

Implementation

4.1 Previous works

In the past years several efforts [1],[2],[3] have identified as main bottleneck of a general OpenFOAM simulation the linear solver computation. For this reason, past efforts were mainly concentrated on improving exclusively it.

One particular effort that went beyond the solver computation is `RapidCFD` [25]. RapidCFD is an open-source OpenFOAM implementation capable of running almost entire simulations on NVIDIA GPUs. it is based on a old and fixed version of OpenFOAM codebase. For the assembly step, RapidCFD uses the `Thrust` library [16]. Thrust is a powerful library of parallel algorithms and data structures written in C++ library with a CUDA back-end. Including Thrust in the OpenFOAM framework is certainly effective for running code on NVIDIA GPUs but it creates a dependency and reduces the native portability of OpenFOAM. We will focus only on native C++ techniques are used in the Proof-of-Concept, so that the portability of the code is not affected in any way.

4.2 Identification of `simpleFOAM` hot-spots

`simpleFOAM` [24] is a popular OpenFOAM application which represents fairly well various well-known bottlenecks. This application implements a steady-state solver for incompressible, turbulent flows, using the `SIMPLE` (semi-implicit method for pressure linked equations) algorithm. This algorithm solves the Navier-Stokes equations decoupling pressure and velocity. This algorithm generates two linear systems, one for the velocity components and one for the pressure correction. Analyses revealed that most of the time is spent solving the system related to the pressure correction equation.

The resolution on GPU of the two linear systems can be performed by calling the `PETSc` solvers via the `PETSc4FOAM` [20]. `PETSc4FOAM` also supports NVIDIA `AmgX` (Algebraic multi-

grid Solver) linear solver library. This is an open-source GPU accelerated iterative solver library with special focus on multi-grid methods. Preliminary results show that solving the pressure equation with `PETSc4FOAM + NVIDIA AmgX` back-end on NVIDIA V100 GPUs is up to 7 times faster than the OpenFOAM GAMG-PCG.

We have decided to focus on the gradient computation. In order to apply the techniques explained in chapter 2, we have looked at the most computationally intensive functions beyond the solver itself. We used `Valgrind` [26] to identify precisely which routines prioritise to move from C++ to more modern and parallel-ready C++17.

Valgrind is an open-source instrumentation framework for building dynamic analysis tools. It comes with a set of tools each of which performs some kind of debugging, profiling and other similar tasks. The tool which was used to identify the main intensive routines was the callgrind tool.

The callgrind output can be visualized easily with the `kcachegrind` tool [27]. Figure 4.1 shows a partial callgrind output representing the gradient computation of a target execution.

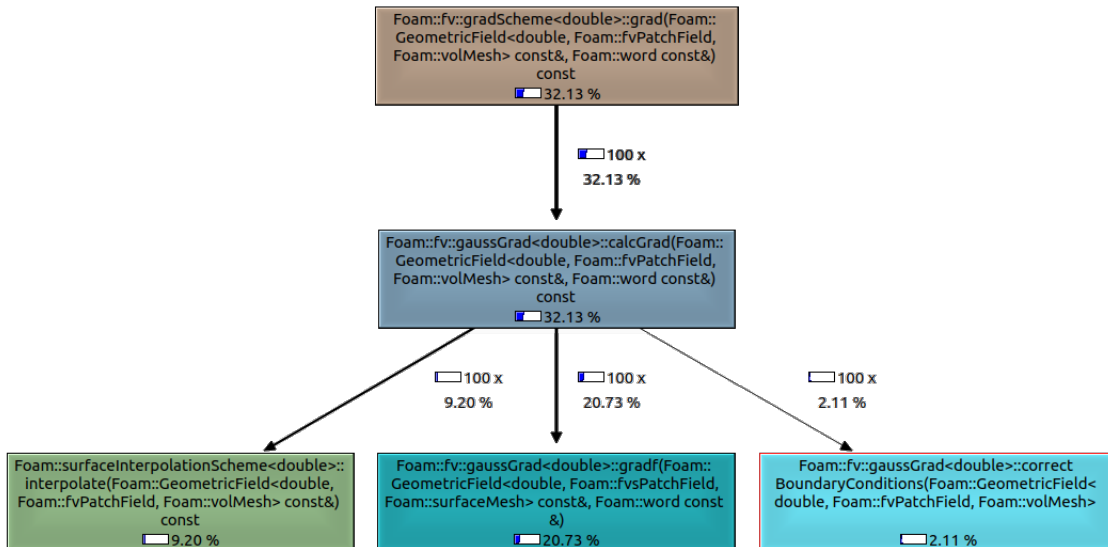


Figure 4.1: Partial Kcachegrind output of the gradient computation

From the figure 4.1 we observed that the `grad` routine calls the `calcGrad` routine which then calls three other routines together consuming 99% of total `grad` walltime:

- Routine **`interpolate`** from the `surfaceInterpolationScheme` class.
- Routine **`gradf`** from the `gaussGrad` class.
- Routine **`correctBoundaryConditions`** from the `gaussGrad` class.

From this output, the calculation of the gradient can be divided into three phases corresponding to the three mentioned functions. In the following, each phase will be analyzed in detail, explaining which functions and which classes have been modified to perform the computation in parallel.

4.3 Code Porting and Refactoring

The details of this work are explained by showing which functions have been modified in order to parallelize the gradient calculation and execute it in multi-threading on CPU or offload to GPU. In particular, this work focuses on the Gauss-Green method 3.2 for the gradient discretization.

4.3.1 Routine `SurfaceIntepolationScheme`

`SurfaceInterpolationScheme` [28] is a class implemented in the finite volume library which performs interpolation from volume fields to face fields. The member function `dotInterpolate` was modified. Since it is not possible to put the entire code here due to its length[1], we will be report only the modified parts.

The original OpenFOAM code is shown in 4.1. Here `P` and `N` are constant references to the owner and neighbour `labelUList`, `lambda` is a constant reference to a `surfaceScalarField`, `vfi` is a constant reference to a `GeometricField`, `Sfi` is a constant reference to an `SFType::Internal` and `sfi` is a reference to the output field.

```
for (label fi=0; fi<P.size(); fi++)
{
    sfi[fi] = Sfi[fi] & (lambda[fi]*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]]);
}
```

Listing 4.1: dotInterpolate cycle

The previous loop runs sequentially on the CPU. One way to execute that loop in parallel is by using the standard `for_each` algorithm 2.1, iterating in parallel over an iterator generated by `iota` and performing the calculation within a `lambda`. The objects required for the calculation are captured by value into the `lambda` passing the pointers to them. The previous code can be modified in the following way 4.2:

```
auto* Sfii = &Sfi;
auto iter=std::views::iota(0,P.size());
std::for_each(std::execution::par,iter.begin(),iter.end(),
    [N,P,l=lambda.cdata(),Sfii,v=vfi.cdata(),s=sfi.data()](const auto& fi){
        s[fi] = (*Sfii)[fi] & (l[fi]*(v[P[fi]] - v[N[fi]]) + v[N[fi]]);
    });
```

Listing 4.2: dotInterpolate cycle in parallel

[1]See `https://www.openfoam.com/documentation/guides/latest/api/surfaceInterpolationScheme_8C_source.html`

Another member function that is called in this phase is the `weights`. This function computes the weights required for the interpolation by the previous `dotInterpolate`. It is a member function of the class `basicFvGeometryScheme`[2]. In this function the following loop is present:

```
forAll(owner, facei)
{
    // Note: mag in the dot-product.
    // For all valid meshes, the non-orthogonality will be less than
    // 90 deg and the dot-product will be positive.  For invalid
    // meshes (d & s <= 0), this will stabilise the calculation
    // but the result will be poor.
    scalar SfdOwn = mag(Sf[facei] & (Cf[facei] - C[owner[facei]]));
    scalar SfdNei = mag(Sf[facei] & (C[neighbour[facei]] - Cf[facei]));
    if (mag(SfdOwn + SfdNei) > ROOTVSMALL)
    {
         w[facei] = SfdNei/(SfdOwn + SfdNei);
    }
    else
    {
        w[facei] = 0.5;
    }
}
```

Listing 4.3: weights cycle

The first thing that can be noticed in the previous loop is the `forAll` syntax. This is a macro definition that replaces in a more compact way the standard C++ "for" loop. It is defined in the file `UList.H` and it works in the following way:

```
#define forAll(list, i) \
    for (Foam::label i=0; i<(list).size(); i++)
```

Listing 4.4: forAll macro syntax

In the listing 4.3 the `owner` and `neighbour` are owner and neighbour `labelULists`, `Sf` is the face area vector, `C` is the cell centres vector, `Cf` is the face centres vector, `w` is the output surfaceScalarField representing the wanted weights. One way to parallelize the previous loop is by using the `transform` algorithm 2.2 in the following way:

```
auto iter=std::views::iota(0,owner.size());
std::transform(std::execution::par,iter.begin(),iter.end(),w.begin(),
    [ne=neighbour.cdata(),ow=owner.cdata(),s=Sf.cdata(),c=C.cdata(),cf=Cf.cdata()](const auto& facei
    ){
        scalar SfdOwn = mag(s[facei] & (cf[facei] - c[ow[facei]]));
        scalar SfdNei = mag(s[facei] & (c[ne[facei]] - cf[facei]));
        if (mag(SfdOwn + SfdNei) > ROOTVSMALL){
            return SfdNei/(SfdOwn + SfdNei);
        }else{
            return 0.5;
        }
   });
```

Listing 4.5: weights cycle in parallel

4.3.2 Routine gradf

The second and the most computational intensive phase in the gradient evaluation is the one which calls the `gradf` member function. This function is part of the `gaussGrad` class.

[2]https://www.openfoam.com/documentation/guides/latest/api/basicFvGeometryScheme_8C_source.html

Field construction

The first thing worth noting is the construction of the gradient field. This is done in the following listing:

```
typedef GeometricField<GradType, fvPatchField, volMesh> GradFieldType;
tmp<GradFieldType> tgGrad
(
    new GradFieldType
    (
        IOobject
        (
            name,
            ssf.instance(),
            mesh,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh,
        dimensioned<GradType>(ssf.dimensions()/dimLength, Zero),
        extrapolatedCalculatedFvPatchField<GradType>::typeName
    )
);
```

Listing 4.6: Gradient field constructor

This constructor calls other functions and, at the end, it invokes the field constructor of the class `Field`. This field constructor takes the length of the field to be built and the value at which to initialize its elements, listing 4.7. Then it calls the list constructor which allocates memory and initializes the elements using a `forAll`.

```
template<class Type>
inline Foam::Field<Type>::Field(const label len, const Type& val)
:
     List<Type>(len, val)
{}
```

Listing 4.7: Field constructor

This runs sequentially on the CPU and most of the time is spent initializing the elements. One way to parallelize this constructor is calling the default list constructor to allocate the memory and then use the `fill_n` algorithm to fill in parallel the field 4.8.

```
template<class Type>
inline Foam::Field<Type>::Field(const label len, const Type& val)
:
    List<Type>(len)
{
    std::fill_n(std::execution::par,this->begin(),len,val);
}
```

Listing 4.8: Field constructor in parallel

First cycle

The first cycle that can be found in the `gradf` function makes the sum over the faces of the cell. It is based on the global face numbering and it uses the upper and lower addressing list to add or subtract the flux to cell values, listing 4.9.

This loop is complex to parallelize because developing it in parallel by having each thread performing a different iteration would lead to a data race condition. In fact, if two threads

have different `facei` values at which correspond the same `owner` or `neighbour` element then a data race occurs.

```
forAll(owner, facei)
{
    const GradType Sfssf = Sf[facei]*issf[facei];
    igGrad[owner[facei]] += Sfssf;
    igGrad[neighbour[facei]] -= Sfssf;
}
```

Listing 4.9: First cycle in the gradf

There could be two different ways to parallelize the loop in 4.9. One way could be to use atomic operations, ensuring that each thread writes safely while the others wait. This solution can be applied but it is not recommended. One main reason is that inserting the atomic operations in the loop should be done taking into account the type of field which represents the gradient. For example it could be a vector or a tensor. Depending on this one should subdivide the cases by treating them differently. However, this would lead to a very long code, which would be difficult to read and maintain.

We proposed and implemented a re-organization of the owner and neighbour lists. By doing so only the owner list is taken into account. The same procedure is applied to neighbour list.

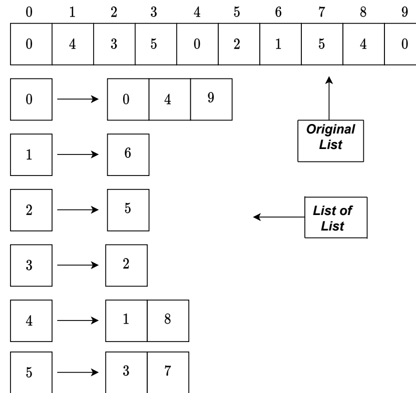


Figure 4.2: List of List

As said in chapter 2 the owner is a list with size equal to the number of faces, whose elements are the labels identifying the cell owner of the face. Hence the elements in the list vary in the range $[0, N_{cell}]$, with N_{Cell} the number of cells. Since each cell label owns different faces, the loop 4.9 can be safely parallelized by iterating over the number of cells and for each cell iterating over the faces that this cell owns. One way to organize the owner list in this way is to reorder the elements in a list of lists like in figure 4.2. For practical reasons the list on the left of the arrows will be called external list while the list on the right will be the internal list.

In the case depicted in the figure 4.2 the cell labels vary between 0 and 5, the loop 4.9 can be performed in parallel along this range, the external list, with each thread that executes an internal loop on the connected internal list and takes the values corresponding to these indices from the original list.
The focus now is on how to construct this list of lists in an optimized way. Taking inspiration from the RapidCFD software [29] one way to proceed is to create one list with the elements contained in the internal list and another list that contains the starting point of the previous list between two successive cell elements. Figure 4.3 shows these new lists called `Owner` and `OwnerStart`.

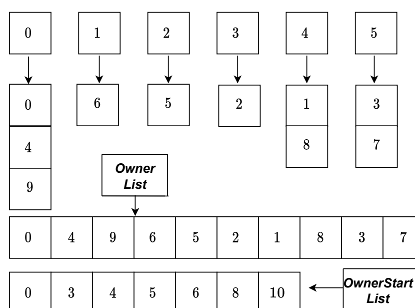


Figure 4.3: Optimized list of list

The next point to focus on is how to implement these lists. The goal is to create these lists in parallel using the same C++ techniques as before. It is very important to note that, in a general OpenFOAM simulation, these lists are used multiple times during the simulation but, unless the mesh geometry changes, they are only created once.

The first list can be easily computed in parallel using the standard sort algorithm. Specifically two lists are needed, the first list is the original owner list, the second list is the output and it is initialized as a list of indices from 0 to the size of the owner list. Then sorting the first list and rearranging the second one accordingly, the output is obtained, as it can be seen from figure 4.4.

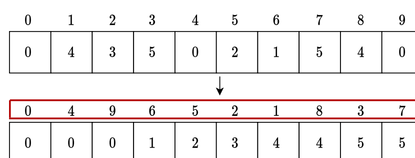


Figure 4.4: Sorting list

A possible implementation of the previous algorithm with the standard C++20 parallelization techniques is described in the listing 4.10. This function was created as member function of the class `lduAddressing`. One of the main classes that handles the pointers required to represent the typical OpenFOAM matrices in sparse format, as seen in the previous chapter.

```
void Foam::lduAddressing::sorting_pair(labelList& index, labelList& val) const
{
    const label N=index.size();
    auto iter=std::views::iota(0,N);
    std::vector<std::pair<label,label>> tmp_pair;
    tmp_pair.reserve(N);

    std::transform(std::execution::par,index.begin(),index.end(),val.begin(),tmp_pair.begin(),
                    [](const auto& i, const auto& v){return std::make_pair(i,v);});

    std::stable_sort(std::execution::par,tmp_pair.begin(),tmp_pair.begin()+N,
                [=](const auto& p1, const auto& p2 ){
                    return p1.second<p2.second;
                });

    std::for_each(std::execution::par,
                iter.begin(),
                iter.end(),
                [pr=tmp_pair.data(),id=index.data(),vl=val.data()](const auto& i){
                    id[i]=pr[i].first;
                    vl[i]=pr[i].second;
                });
}
```

Listing 4.10: Sorting lists

The second list is slightly more difficult to compute. Figure 4.5 shows a possible solution. Starting from the reordered owner list (it works even if the list is unordered) another list is appended at the end. This second list has indices from 0 to the number of cells N_{cell}. Another list of size the owner list size plus the N_{cell} is created initializing its elements to zero. Then all the elements from 0 to the owner size are set to 1. The second step is sorting the lists as previously. The third step is to reduce both lists according to the first list while the elements of the second list are added. Finally an `exclusive_scan` algorithm can be applied.

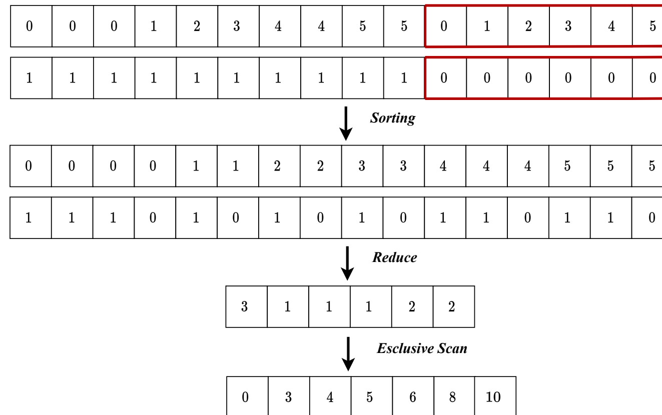


Figure 4.5: Start owner list

A possible implementation of the previous algorithm is shown in the listing 4.11. This function is called `csr_list` and it is a member function of the class `lduAddressing` as the previous function.

```
void Foam::lduAddressing::csr_list(labelList& neigh_sort, labelList& start, const label& n, const
     label& N) const
{
    atomic<label> *stmp=new atomic<label>[n];
    labelList ones(n+N);
    labelList nb_tmp(n+N);
```

```
    auto iter2=std::views::iota(0,n);

    std::fill(std::execution::par_unseq,ones.begin(),ones.end(),1);
    std::fill(std::execution::par_unseq,nb_tmp.begin(),nb_tmp.end(),0);

    std::copy(std::execution::par_unseq,neigh_sort.begin(),neigh_sort.end(),nb_tmp.begin());
    std::copy(std::execution::par_unseq,iter2.begin(),iter2.end(),nb_tmp.begin()+N);

    std::fill(std::execution::par_unseq,ones.begin()+N,ones.end(),0);
    std::fill(std::execution::par_unseq,stmp,stmp+n,0);

    sorting_pair(ones,nb_tmp);

    auto iter3=std::views::iota(0,n+N);
    std::for_each(std::execution::par,iter3.begin(),iter3.end(),
                  [=,nb=nb_tmp.data(),on=ones.data()](const auto& i){
                      stmp[nb[i]].fetch_add(on[i],memory_order_relaxed);
                  });

    std::copy(std::execution::par,stmp,stmp+n,start.begin());
    std::exclusive_scan(std::execution::par,start.begin(),start.end(),start.begin(),0);

    delete [] stmp;
}
```

Listing 4.11: Start list creation

These lists are useful to safely parallelize the first gradient cycle 4.9. To compute that cycle in parallel four lists are needed: the `ownerList_`, the `ownerStart_`, the `neighbourList_` and the `neighbourStart_`. For this reason four pointers to `labelList` were allocated in the `lduAddressing.H`. Those pointers represent the lists.
To allocate the lists, public functions `ownerList`, `ownerStart`, `neighbourList` and `neighbour Start` have been defined in `lduAddressing.H`. In the next only the `ownerList` is shown.

```
const Foam::labelUList& Foam::lduAddressing::ownerList() const
{
    if (!ownerList_)
    {
        calcownerList();
    }
    return *ownerList_;
}
```

Listing 4.12: OwnerList creation

As it can be seen in 4.12 the function checks if the pointer, in this case `ownerList_`, is allocated. If it is not allocated it calls the `calcownerList` function. The same happens for `ownerStart`, with `ownerStart_` instead of `ownerList_`. This function calls the same `calcownerList` because in that function both `ownerStart_` and `ownerList_` are allocated, but it is still useful because it helps to handle lists separately.

The function `calcownerList` is shown in the following listing 4.13, it is a member function of `lduAddressing` class. As it can be seen it allocates `ownerStart_` and `ownerList_` lists.

```
void Foam::lduAddressing::calcownerList() const
{
    if (ownerList_)
    {
        FatalErrorIn("lduAddressing::calcownerList() const")
            << "ownerList already calculated"
            << abort(FatalError);
    }
    const labelUList& owner=lowerAddr();
```

```
    const label N=owner.size();
    const label n=size();

    ownerList_ = new labelList(N);
    ownerStart_ = new labelList(n+1);

    labelList owner_sort(N);
    labelList& olist= *ownerList_;
    labelList& ostart= *ownerStart_;
    labelList& osort= owner_sort;

    std::copy(std::execution::par_unseq,std::views::iota(0).begin(),std::views::iota(N).begin(),
     olist.begin());
    std::copy(std::execution::par_unseq,owner.begin(),owner.end(),osort.begin());

    sorting_pair(olist,osort);

    csr_list(osort,ostart,n,N);
}
```

Listing 4.13: OwnerList and OwnerStart allocation

Finally with these lists the first cycle 4.9 in the gradient computation can be translated in parallel in the following way:

```
const labelUList& owlist=mesh.lduAddr().ownerList();
const labelUList& owstart=mesh.lduAddr().ownerStart();
const labelUList& nelist=mesh.lduAddr().neighbourList();
const labelUList& nestart=mesh.lduAddr().neighbourStart();

auto iter=std::views::iota(0,igGrad.size());
std::for_each(std::execution::par,iter.begin(),iter.end(),
    [ol=owlist.cdata(),os=owstart.cdata(),nl=nelist.cdata(),ns=nestart.cdata(),sf=Sf.cdata(),is=issf
     .cdata(),ig=igGrad.data()](const label& facei){
        for(int i=os[facei]; i<os[facei+1];++i){
            ig[facei]+= sf[ol[i]]*is[ol[i]];
        }
        for(int i=ns[facei]; i<ns[facei+1];++i){
            ig[facei]-= sf[nl[i]]*is[nl[i]];
        }
    });
```

Listing 4.14: first cycle gradient in parallel

Second cycle

The second cycle in the `gradf` takes into account the boundary of the domain, it is shown in the listing 4.15.

```
forAll(mesh.boundary(), patchi)
{
    const labelUList& pFaceCells =
    mesh.boundary()[patchi].faceCells();
    const vectorField& pSf = mesh.Sf().boundaryField()[patchi];
    const fvsPatchField<Type>& pssf = ssf.boundaryField()[patchi];

    forAll(mesh.boundary()[patchi], facei)
    {
      igGrad[pFaceCells[facei]] += pSf[facei]*pssf[facei];
    }
}
```

Listing 4.15: Second cycle gradient

The most computationally intensive cycle is the innermost one, so the goal is to parallelize it. However, as in the case of the first cycle, running the innermost cycle in parallel without any precautions would lead to a data race. To parallelize it the followed procedure was the

same of the first cycle. It was based on the creation of similar lists like before. Even these lists can only be created once and used in the code when needed.

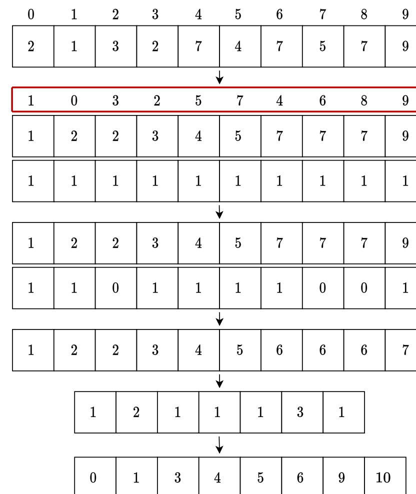


Figure 4.6: Start list creation

Two lists were created as before. One list, called `faceIndex`, represents the indices to take from the original `pFaceCells` list, the other list represents the starting point `faceStart`. The first list is obtained using the sorting function as before 4.10. To build the second list a different approach was followed.
Figure 4.6 shows the procedure to obtain the list. As it can be seen the first list is obtained after the sorting by key. Then a list initialized to one is allocated. The next step iterates over the sorted list and when two adjacent elements are equal, the element in the list of ones corresponding to the index of the second element found to be equal is set to 0. Then an `inlusive_scan` is applied. Hence the reduce is applied like in the first cycle algorithm 4.5. Before applying the reduce, 1 is subtracted from the elements of the list. Finally using an `exclusive_scan` the `faceStart` list is obtained.

A possible implementation of this algorithm is shown in listing 4.16. This function is a member function of the class `gaussGrad`.

```
template<class Type>
void Foam::fv::gaussGrad<Type>::csr_list2(const labelUList& list, labelList& lindex, labelList&
    start, label& n){
    const int N=list.size();
    labelList lsort;
    labelList ones;
    lindex.resize(N);
    lsort.resize(N);
    ones.resize(N);
    auto iter=std::views::iota(0,N);

    std::copy(std::execution::par_unseq,list.begin(),list.end(),lsort.begin());
    std::copy(std::execution::par_unseq,iter.begin(),iter.end(),lindex.begin());
    std::fill(std::execution::par_unseq,ones.begin(),ones.end(),1);
    sorting_pair2(lindex,lsort);
```

```
    std::for_each(std::execution::par,iter.begin(),iter.end()-1,
            [ns=lsort.data(),ts=ones.data()](const auto& x){
                if(ns[x]==ns[x+1]){
                    ts[x+1]=0;
                }
            });
    std::inclusive_scan(std::execution::par,ones.begin(),ones.end(),ones.begin());

    n=ones.last();
    atomic<int> *stmp=new atomic<int>[n+1];
    start.resize(n+1);

    std::fill(std::execution::par,stmp, stmp+n+1, 0);
    std::for_each(std::execution::par,iter.begin(),iter.end(),
                [ns=stmp,ts=ones.data()](const auto& x){
                        ns[ts[x]-1].fetch_add(1,memory_order_relaxed);
                });
    std::move(std::execution::par,stmp,stmp+n+1,start.begin());
    std::exclusive_scan(std::execution::par,start.begin(),start.end(),start.begin(),0);

    delete [] stmp;
}
```

Listing 4.16: List creation

With the previous lists, the second cycle in the **gradf** function can be parallelized in the following way 4.17:

```
forAll(mesh.boundary(), patchi)
{
  const labelUList& pFaceCells =
        mesh.boundary()[patchi].faceCells();
  const vectorField& pSf = mesh.Sf().boundaryField()[patchi];
  const fvsPatchField<Type>& pssf = ssf.boundaryField()[patchi];

  if(mesh.boundary()[patchi].size()!=0){
      labelList faceIndex;
      labelList faceStart;
          label n=0;

          csr_list2(pFaceCells,faceIndex,faceStart,n);

          std::for_each(std::execution::par,
                      std::views::iota(0).begin(),
                      std::views::iota(n).begin(),
                      [ig=igGrad.data(),f=pFaceCells.cdata(),fir=pSf.cdata(),sec=pssf.cdata(),pstr=
    faceStart.cdata(),plst=faceIndex.cdata()](const label& facei){
                          label id=f[plst[pstr[facei]]];
                          for(int i=pstr[facei]; i<pstr[facei+1];++i){
                              ig[id]+=fir[plst[i]]*sec[plst[i]];
                          }
                      });
  }
}
```

Listing 4.17: Second cycle in parallel

Third cycle

The third cycle in the **gradf** function is not really a cycle, it is an operation that is performed using a loop. The operation is the following:

```
igGrad /= mesh.V();
```

Listing 4.18: Third cycle

The previous operator divides each component of the `igGrad` field by each component of the `mesh.V()` field. As shown in listing 4.19 it can be easily parallelized using a transform algorithm.

```
auto& meshV=mesh.V();
std::transform(std::execution::par_unseq,igGrad.begin(),igGrad.end(),meshV.cbegin(),igGrad.begin(),
                [](auto& ig, const auto& val){return ig/val;});
```

Listing 4.19: Third cycle in parallel

4.3.3 Routine `CorrectBoundaryConditions`

The last function that is called from the `calcGrad` routine is the `CorrectBoundaryConditions`. In this function there is one main cycle over the boundary domain patches.

```
forAll(vsf.boundaryField(), patchi)
{
    if (!vsf.boundaryField()[patchi].coupled())
    {
        const vectorField n
        (
            vsf.mesh().Sf().boundaryField()[patchi]
          / vsf.mesh().magSf().boundaryField()[patchi]
        );

        gGradbf[patchi] += n *
        (
            vsf.boundaryField()[patchi].snGrad()
          - (n & gGradbf[patchi])
        );
    }
 }
```

Listing 4.20: Correction boundary condition

As it can be seen, this cycle calls different functions. Some of these functions are computationally intensive and they call other functions. During the execution of `magSf`, the function `mag` is called. This function is originally implemented in the following way:

```
template<class Type>
void mag
(
    Field<typename typeOfMag<Type>::type>& res,
    const UList<Type>& f
)
{
    typedef typename typeOfMag<Type>::type magType;

    TFOR_ALL_F_OP_FUNC_F(magType, res, =, mag, Type, f)
}
```

Listing 4.21: mag function

The loop is executed sequentially by the macro `TFOR_ALL_F_OP_FUNC_F`. This function can be parallelized as follow:

```
template<class Type>
void mag
(
    Field<typename typeOfMag<Type>::type>& res,
    const UList<Type>& f
)
{
    typedef typename typeOfMag<Type>::type magType;

    std::transform(std::execution::par_unseq,f.cbegin(),f.cend(),
                  res.begin(),[=](const auto& ff){
                      return mag(ff);
                  });
}
```

Listing 4.22: Parallel mag function

Another function called in the cycle 4.20 is `snGrad`. This function in turn calls two other functions: `patchInternalField` and `deltaCoeffs`. Originally this function runs sequentially as follow:

```
template<class Type>
Foam::tmp<Foam::Field<Type>> Foam::fvPatch::patchInternalField
(
    const UList<Type>& f,
    const labelUList& faceCells
) const
{
    auto tpif = tmp<Field<Type>>::New(size());
    auto& pif = tpif.ref();
    auto iter=std::views::iota(0,pif.size());

    std::transform(std::execution::par,
                    iter.begin(),
                    iter.end(),
                    pif.begin(),
                    [ff=f.cdata(),fc=faceCells.cdata()](const auto& facei){
                        return ff[fc[facei]];
                    });
    return tpif;
}
```

Listing 4.23: PatchInternalField function

It can be parallelized using the transform algorithm as follow:

```
template<class Type>
Foam::tmp<Foam::Field<Type>> Foam::fvPatch::patchInternalField
(
    const UList<Type>& f,
    const labelUList& faceCells
) const
{
    auto tpif = tmp<Field<Type>>::New(size());
    auto& pif = tpif.ref();

    auto iter=std::views::iota(0,pif.size());

    std::transform(std::execution::par,
                    iter.begin(),
                    iter.end(),
                    pif.begin(),
                    [ff=f.cdata(),fc=faceCells.cdata()](const auto& facei){
                        return ff[fc[facei]];
                    });
    return tpif;
}
```

Listing 4.24: PatchInternalField function in parallel

Inside `DeltaCoeffs`function there is an intensive computation given by the loop below:

```
forAll(owner, facei)
{
    deltaCoeffs[facei] = 1.0/mag(C[neighbour[facei]] - C[owner[facei]]);
}
```

Listing 4.25: Deltacoeff function

This loop can be parallelized using another transform algorithm as follow:

```
std::transform(std::execution::par,
                owner.begin(),
                owner.end(),
                neighbour.begin(),
                deltaCoeffs.begin(),
                [CC=C.cdata()](const auto& nf, const auto& no){
                    return 1.0/mag(CC[nf]-CC[no]);
```

```
            });
```

Listing 4.26: Deltacoeff function in parallel

4.4 OpenFOAM gradient mini-app

Since this work is focused on the gradient part, we built a Proof-of-Concept application that only invokes the gradient computation (**grad**).

```
// ************************************************************************* //

#include "fvCFD.H"
#include "fvOptions.H"
#include "simpleControl.H"
#include "FixedList.H"
#include "Pair.H"

int main(int argc, char *argv[])
{
    argList::addNote
    (
        "Laplace equation solver for a scalar quantity."
    );

    #include "addCheckCaseOptions.H"
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createMesh.H"
    simpleControl simple(mesh);
    #include "createFields.H"

    unsigned int i = 0;
    volScalarField yPos = T.mesh().C().component(vector::Y).ref();
    volScalarField xPos = T.mesh().C().component(vector::X).ref();

    for (auto i = 0; i < T.size(); i++)
    {
        T[i] = std::exp( - 2 * std::pow(xPos[i] - 0.45, 2) -
                         2 * std::pow(yPos[i] - 0.45, 2));
    }

    Info<< "\nCalculating temperature distribution\n" << endl;

    // Lists cnstruction
    const labelUList& owlist=T.mesh().lduAddr().ownerList();
    const labelUList& owstart=T.mesh().lduAddr().ownerStart();
    const labelUList& nelist=T.mesh().lduAddr().neighbourList();
    const labelUList& nestart=T.mesh().lduAddr().neighbourStart();

    // main loop (2 iterations, hard-coded)
    auto start = std::chrono::steady_clock::now();
    while(i<2){
        fvc::grad(T);
        ++i;
    }
    auto end = std::chrono::duration<double>(std::chrono::steady_clock::now() -
               start).count();

    Info<<"TOTAL TIME GRADIENT COMPUTATION (in [s]) :"<<end <<endl;

    return 0;
}
// ************************************************************************* //
```

Listing 4.27: Test Application

This application starts by reading the mesh then creates the scalar temperature field and initializes its values following a Gaussian distribution. Finally it computes the gradient two

times.

4.5 Profiling a GPU application

The Unified Memory approach made the development of standard C++ parallelism on GPUs possible. One of the main aspects to be taken into account when using this parallel approach is the page faults that can occur if data has to be moved between CPU and GPU. These page faults can be detected using NVIDIA Nsight Systems [9], a performance analysis tool, part of the powerful debugging and profiling NVIDIA Nsight Tools suite [10], designed to visualize an application's algorithm.

Another tool useful to store events happening in the GPU is the `NVIDIA Tools Extension` (NVTX) [11]. It is composed by a set of APIs for annotating events, code ranges and resources in the applications. These information are visualized in the NVIDIA Nsight Systems GUI to help improving analysis and visualization of data, simplifying the correlation between what the code does versus the behaviour of the compute platform (CPU and GPU). In particular, markers denote specific moments in time and are useful to catch specific sections of the code during the application running.

The library introduces close to zero overhead if no profiling is activated (so the execution is not performed with the `nsys` tool). The overhead when profiling is active varies according to multiple factors but, if tuned, it can be considered negligible.

4.6 Tuning data movement with explicit pre-fetching

By analyzing the `nsys` profile reports, it is possible to understand the origin of page faults and, in general, the data transfer that takes place during the execution of the application. Usually page faults and data transfer are expensive, especially if access are randoms and page fault is triggered multiple times within a kernel execution stalling kernel progression while waiting data to be moved from CPU to GPU.

It is possible to get rid of them by using data pre-fetching APIs that are part of CUDA C [8]. Pre-fetching helps moving big chunk of contiguous data to the GPU on-demand rather than rely on the automatic Unified Memory management mechanism. By doing so, better data transfer bandwidth is achieved. CUDA provides several routines to perform and control pre-fetching behaviour it, i.e. *cudaMemPrefetchAsync*. When the data required by a kernel is fully on the GPU, it can be processed by a kernel without generating page faults resulting

in a better kernel time and efficient use of GPU hardware.

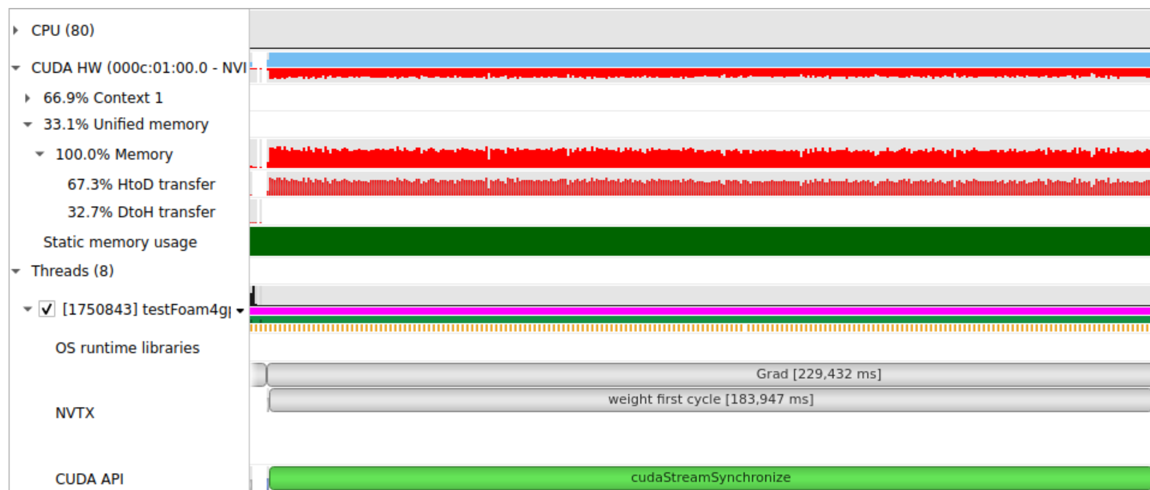


Figure 4.7: `Weight first cycle` without pre-fetching tools

Figure 4.7 illustrates a section of the NVIDIA Nsight Systems profile representing a code function in which no pre-fetching techniques were added. In particular, it can be observed from the first blue line that while the routine, whose NVTX tag is called `weight first cycle`, is running on the GPU, many page faults occur. These page faults can be identified from the red lines.

Figure 4.8 shows the same code executed by inserting the pre-fetching techniques before executing the `weight first cycle` tag. As can be seen, the red markers are no longer present. So there are no more page faults and the entire calculation is performed on the GPU without interruption.

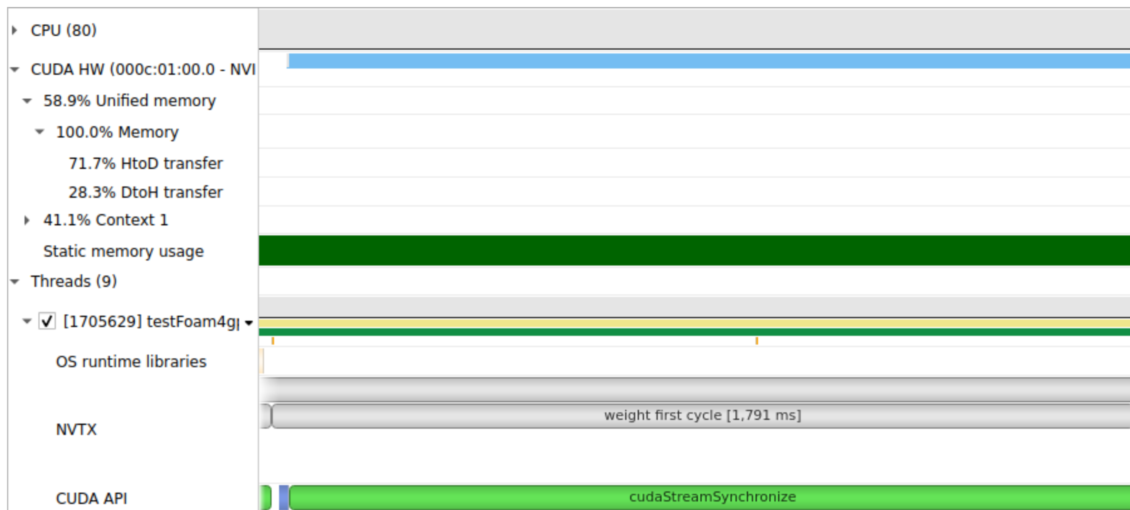


Figure 4.8: `Weight first cycle` with pre-fetching tools

Figure 4.9 shows how pre-fetching appears in a general profile. It is identify by the green lines. To use pre-fetching in this work, the CUDA function `cudaMemPrefetchAsync()` was called up by passing it the pointer to be pre-fetched, the container size in bytes and the destination device to be pre-fetched to. One might think that including pre-fetching within the code restricts the portability of the code. However, our goal is to demonstrate that using the new C++ standard, it is possible to run code on GPUs. Perhaps in the future, the entire simulation could be run on the GPU using these techniques. At that point, pre-fetching would not be necessary since, if everything is executed on the GPU from the beginning, the data needed for the gradient calculation would already be on the GPU.

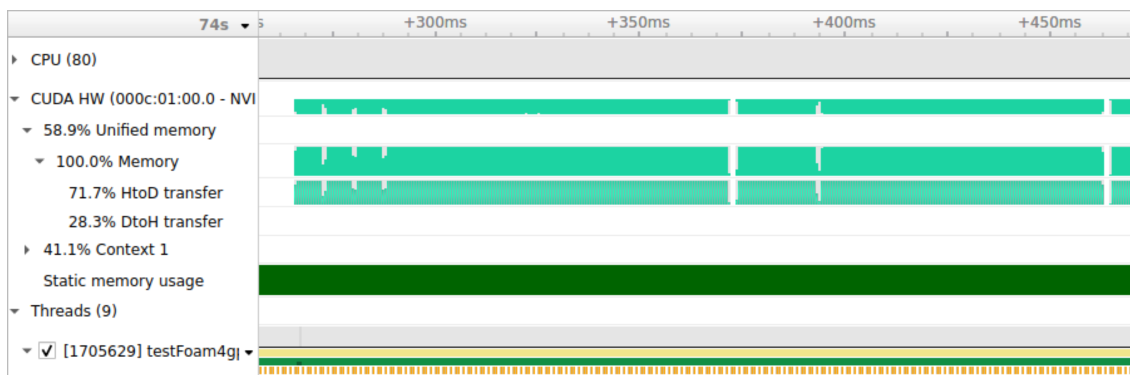


Figure 4.9: Pre-fetching section

Chapter 5

Results

5.1 Mini-app input cases

All meshes considered in this work are 2D meshes, which in OpenFOAM means setting the third dimension to one. Although tests on 3D meshes were done correctly, these are not shown here for reasons of time and space. We pick 3 different sizes as per number of unknown, details are described in table 5.1.

	X-SIZE	Y-SIZE	Z-SIZE	Unknowns [millions]
`Mesh 16M`	4000	4000	1	16
`Mesh 32M`	8000	4000	1	32
`Mesh 64M`	8000	8000	1	64

Table 5.1: Mesh specifications

5.2 Computing platform

The simulations that will be shown in the next sections, were performed on a heterogeneous platform called **NVIDIA Arm HPC Developer Kit** [5]. Figure 5.1 shows a diagram of this system. The platform main compute engines are:

- A single-socket ARM Ampere Computing `Altra` processor with 80 cores based on Arm Neoverse N1 running at 3.0 GHz.
- Two Ampere GPUs A100 PCIe with 40 GByte HBM2 memory not connected via NVlink due to the server form factor.

The `Altra` CPU is custom built for large-scale public and private cloud environments [4]. This processor has only NEON SIMD (2×128 bits) but competitive memory bandwidth.

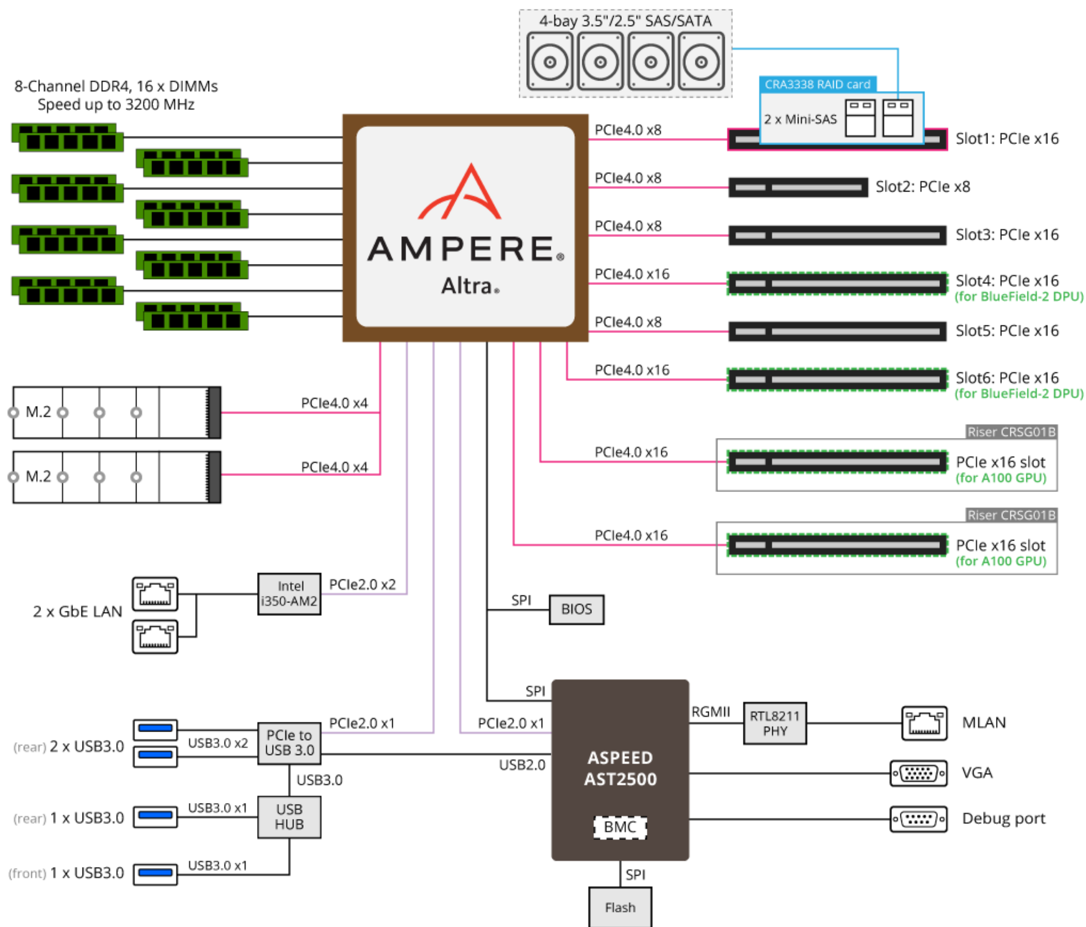


Figure 5.1: The NVIDIA Arm HPC Developer Kit platform.

The A100 GPU supports PCIe Gen4 which doubles the bandwidth of PCIe Gen3. The faster speed is especially beneficial for A100 GPUs connecting to PCIe Gen4 CPUs like the Ampere Computing Altra. At the time of the start of this project, there were no x86 platforms on the market with PCIe Gen4.

5.3 Results obtained

Using NVIDIA Nsight Systems the application was profiled in order to identify the macro routines called during the gradient computation. Not all functions that are executed within the gradient computation have been parallelized. This is due to limitations of time and difficulty. For this reason, a global comparison, although necessary, may not be very indicative. Therefore, to analyze the advantages and disadvantages of this new parallel programming model, the times of each individual routine were taken and are shown in the next tables. The NVIDIA Nsight Systems tool in combination with NVTX was used to collect these times.

Table 5.2 shows the shortcuts which will be used in the next tables to identify the times

in milliseconds taken by each routine.

NVTX tag	Description	Algorithm	GPU?
`WB`	Weight Built		No
`WFC`	Weight First Cycle	4.3	Yes, Full
`WSC`	Weight Second Cycle		Yes, Full
`DB`	`DotInterpolate` Built		No
`DFC`	`DotInterpolate` First Cycle	4.1	Yes, Full
`DSC`	`DotInterpolate` Second Cycle		No
`GB`	Gradient Built	4.6	Yes, Partial
`GFC`	Gradient First Cycle	4.9	Yes, Full
`GSC`	Gradient Second Cycle	4.15	Yes, Full
`GD`	Gradient Division	4.18	Yes, Full
`GBC`	Gradient Boundary Condition		Yes, Partial
`CBC`	Correct Boundary Condition	4.20	Yes, Partial

Table 5.2: Explanations of various NVTX tags acronyms used.

As briefly presented in Chapter 4, the calculation of the gradient can be divided into 3 parts.

- Routine **`interpolate`** includes the calculation of weights (`WB`, `WFC`, `WSC`) and interpolation (`DB`, `DFC`, `DSC`);
- Routine **`gradf`** computes the gradient field (`GB`, `GFC`, `GSC`, `GD`, `GBC`);
- Routine **`correctBoundaryConditions`** takes into account the boundary conditions (`CBC`), within part there is the calculation of the delta coefficients, which, however, is only performed during the first iteration.

In the following Tables we will only report timings and speed-ups related to fully accelerated tagged sections. All timings shown in the following tables are averaged values across minimum 3 executions.

5.3.1 Performance running `Mesh 16M` input case

The first mesh presented is a mesh of 16 million unknowns. The simulations that were performed here are on 1 node with 80 PURE MPI (original version of the code), 1 node with 1 MPI and 1 GPU, 1 node with 2 MPI and 2 GPUs and 1 node with 10 MPI and 8 threads. Since not all routines were modified to run in parallel, in order to analyze the effectiveness of this new programming model, the execution times of the individual functions making up

the code were taken using NVTX markers. The global times will be analyzed in the next sections. Tables 5.3 and 5.4 represent the speed-up obtained from the simulations. The first three columns compare PURE MPI, 80 MPI processes, with three different settings: the hybrid one, composed of 10 MPI and 8 threads, the 1 MPI process with 1 GPU and the 2 MPI processes with 2 GPUs. The last column compares the execution times obtained with 2 MPI and 2 GPUs with the one resulting from the hybrid approach. Execution times can be seen in the appendices of this document A. Tables A.3 and A.4 show the times spent by these four settings: PURE MPI, Hybrid, 1 GPU and 2 GPUs.
The application performs two iterations of the gradient computation. This is because there are routines calculated only during the first iteration and then recalculated only if the mesh is changed. In an application where the calculation of the gradient is done iteratively several times, the time of the second iteration is more indicative.
As said before, in order to isolate the actual calculation time performed on the GPUs, pre-fetching routines were called explicitly to initiate the transfer of necessary data to the GPU before the actual kernel execution. This is not too restrictive: indeed, one of the possible big goals in the future could be to develop the code so that it runs entirely on different GPUs, and in that case, this pre-fetching would not be necessary and the performance would be as shown in the tables.

In Table 5.3 speed-ups of the first iteration are shown.

NVTX tag	**HYBRID vs PURE MPI**	**1 MPI + 1 GPU vs PURE MPI**	**2 MPI + 2 GPU vs PURE MPI**	**2 MPI + 2 GPU vs HYBRID**
`WFC`	3.13	8.40	16.03	5.12
`WSC`	44.41	60.78	41.98	0.95
`DFC`	3.17	8.20	15.80	4.98
`GFC`	3.19	7.10	10.09	3.17
`GSC`	4.35	1.14	0.30	0.07
`GD`	2.66	5.68	10.79	4.06

Table 5.3: speed-up first iteration `Mesh 16M` (HYBRID = 10 MPI + × 8 CPU threads)

Considering the weight part it can be noticed that a good speed-up is obtained parallelizing the first and second cycle (**`WFC-WSC`**) on both multiple GPUs and the multi-core approach. Even in the interpolation part, a great advantage can be observed when parallelizing the first cycle loop (**`DFC`**). In particular, when executed on 1 GPU it is 8 times faster than 80 MPI. Coherently, the same loop executed on 2 GPUs is 2 times faster than the loop executed on 1 GPU and, therefore, 15 times faster than the loop executed on 80 MPI.
The second macro part computes the gradient field. The first routine shown in the table is

the first gradient cycle (**GFC**). This routine is the most computationally intensive of this phase and one of the most expensive in the gradient calculation. It was parallelized with the new lists defined in chapter 4. As it can be seen using 1 GPU there is a speed-up of 7 and using 2 GPU a speed-up of 10. Using the multi-core gives a speed-up of 3 compared to 80 MPI, which is half of the one obtained with 1 GPU.
The next routine is the second gradient cycle (**GSC**), in this case no great advantage is obtained executing it on GPUs. This for two main reasons. The first is the data transfer between CPU and GPUs. The second is because in this simulation that cycle is not big enough to take full advantage of the GPUs.
The next routine is the gradient division (**GD**). This routine lends itself very well to parallelization and even if there is partial data transfer, very good speed-ups are achieved. On 1 GPU a speed-up of approximately 5x is obtained and on 2 GPUs is doubled. Even with multi-core you get a speed-up of 2.5x, roughly half that of 1 GPU.

NVTX tag	**HYBRID vs PURE MPI**	**1 MPI + 1 GPU vs PURE MPI**	**2 MPI + 2 GPU vs PURE MPI**	**2 MPI + 2 GPU vs HYBRID**
DFC	2.87	8.70	16.75	5.83
GFC	3.07	6.86	11.62	3.78
GSC	0.66	1.75	0.49	0.74
GD	2.43	6.11	11.83	4.86

Table 5.4: speed-up second iteration `Mesh 16M` (HYBRID = 10 MPI × 8 threads)

Table 5.4 shows the performance of the second iteration. As mentioned above, the calculation of weights is only done in the first iteration. So the second iteration starts with interpolation. The speed-up trend is similar to that of the previous iteration. Again, a significant speed-up can be seen in the execution of the first cycle, while the second cycle, which runs sequentially, doesn't show a speed-up but it takes small time.
The gradient part also shows a similar trend to that of the first iteration. Gradient first cycle and gradient division are the routines in which there is a significant speed-up, both when running them on GPUs and on multi-core. The parallelization of the gradient second cycle doesn't show relevant performance since there is data transfer and the loop size is not so large to exploit the gpu.
Finally the last routine, correct boundary condition, as before doesn't show relevant speed-up mainly due to data transfer.

5.3.2 Performance running 32M mesh input case

The second mesh considered in this work is composed of 32 million unknowns. The speed-ups of the NVTX markers are shown in tables 5.5 and 5.6. As before, the former represents the speed-ups of the first iteration while the latter shows the speed-ups of the second iteration. Tables A.5 and A.7 in the appendix A show the corresponding execution times.

NVTX tag	HYBRID vs PURE MPI	1 MPI + 1 GPU vs PURE MPI	2 MPI + 2 GPU vs PURE MPI	2 MPI + 2 GPU vs HYBRID
WFC	2.88	8.40	16.56	5.74
WSC	13.68	91.31	92.78	6.78
DFC	3.91	8.62	16.52	4.22
GFC	3.03	6.82	13.75	4.54
GSC	0.77	1.83	1.07	1.40
GD	3.77	5.88	10.66	2.83

Table 5.5: speed-up first iteration `Mesh 32M` (HYBRID = 10 MPI $\times$ 8 threads)

In the first iteration, looking at weight first cycle routine, a speed-up can be clearly observed, which is more or less the same as in the `Mesh 16M` case.

The same happens in the interpolation part. As it can be observed the speed-ups of this part are not very different from those found by running the simulation on the previous mesh. In particular the parallelized first cycle shows the same trend as before both on GPUs and multi-core.

In the second macro part of the gradient computation, the first gradient cycle has a speed-up of 7 on 1 GPU and 14 on 2 GPUs. Even in multi-core there is a speed-up of 3, about half that with 1 GPU as in the results of the previous mesh. The gradient division shows the same behaviour. The performance of the second gradient cycle improves on that of the previous mesh. This could be because by increasing the mesh size, the GPU is better exploited. The performance of the gradient boundary condition and the correct boundary condition improves. However, this improvement stems from a deterioration in original MPI.

In table 5.6 results of second iteration are shown. As it can be seen the dotinterpolate first cycle shows a speed-up behaviour similar to the previous ones. The same holds for the gradient first cycle and the gradient division.

NVTX tag	HYBRID vs PURE MPI	1 MPI + 1 GPU vs PURE MPI	2 MPI + 2 GPU vs PURE MPI	2 MPI + 2 GPU vs HYBRID
`DFC`	2.66	8.68	17.05	6.42
`GFC`	2.78	6.90	13.58	4.89
`GSC`	2.11	3.64	1.65	0.78
`GD`	2.95	5.27	9.48	3.22

Table 5.6: speed-up second iteration `Mesh 32M` (HYBRID = 10 MPI × 8 threads)

The behaviour of the gradient second cycle function also seems to improve, consistently with the first iteration. However, the times are still too low to see a clear improvement, and there is data transfer in the GPUs case, which limits performance.

5.3.3 Performance running 64M mesh input case

The last mesh on which the modifications were tested is composed of 64 million of unknowns. The size of this mesh doesn't fit on 1 GPU so the simulations were done only on 2 GPUs. As usual tables 5.7 and 5.8 report the speed-ups of the first and second iterations, while tables A.7 and A.8 represent the execution times.

Considering the first iteration 5.7 the weight part shows a behavior similar to that seen in previous meshes. In particular the weight first cycle speed-up is approximately the same of `Mesh 32M` and `Mesh 16M`. In fact, doubling the mesh size doubles the time to perform that cycle from 1.78[ms] (`Mesh 32M`) to 3.52 [ms]. The same holds for `dotinterpolate` first cycle for which the time taken by mesh 32 is 0.69[ms] and here is 1.35 [ms].

NVTX tag	HYBRID vs PURE MPI	2 MPI + 2 GPU vs PURE MPI	2 MPI + 2 GPU vs HYBRID
`WFC`	3.20	17.08	5.35
`WSC`	29.04	154.27	5.31
`DFC`	3.44	17.36	5.05
`GFC`	2.62	13.13	5.01
`GSC`	7.35	2.45	0.33
`GD`	2.55	10.99	4.32

Table 5.7: speed-up first iteration `Mesh 64M` (HYBRID = 10 MPI × 8 threads)

In the gradient part a good speed-up can be observed on gradient first cycle and gradient division routines. Even in this case the speed-up is the same of `Mesh 32M`. It can also be noticed that although the times for calculating the gradient second cycle routine increase using 80 MPI, the times taken by the GPU do not show the same speed-up seen in the previous cycle (**`GFC`**). The speed-up of this part is 2.45. This is due to data transfer. In fact, looking at the speed-up obtained in multi-core, where the data transfert is not present, one can see an increase both in the first and in the second iteration, compared to the speed-ups obtained from previous meshes.

Table 5.8 shows the second iteration. Again, the first `dotinterpolation` cycle shows a significant speed increase on the GPU. Performance is also good in multi-core. The same behaviour can be said for the first gradient cycle, the second gradient cycle and the gradient division. It can be notice that these are the main routines that occupy most of the time in a general PURE MPI gradient calculation.

NVTX tag	**HYBRID vs PURE MPI**	**2 MPI + 2 GPU vs PURE MPI**	**2 MPI + 2 GPU vs HYBRID**
`DFC`	2.83	17.18	6.06
`GFC`	2.59	14.68	5.68
`GSC`	4.55	2.11	0.46
`GD`	1.68	9.35	5.56

Table 5.8: speed-up second iteration `Mesh 64M` (HYBRID = 10 MPI $\times$ 8 threads)

5.4 General considerations

The previous sections have highlighted the speed-ups of the code sections provided by the NVTX tag. Tables 5.9 and 5.10 show instead the full speed-ups results for the execution of the first and the second iteration of the gradient computation on the previous meshes 16M, 32M and 64M. These speed-ups take into consideration also portions of code that have not been parallelized using C++11 parallel constructs.

Table 5.9 shows the overall results of the second iteration. The highest speed-up on `Mesh 16M` was obtained using 1 MPI and 1 GPU. In this case the memory occupied on the GPU is about half of the total memory ($\sim$ `17GB`). In this case the speed-up obtained using 2 GPUs is slightly lower. There could be different reasons for that. One cause could be not being able to fully utilize the power of the GPUs, another could be increased data transfer and MPI communications. The comparison between PURE MPI versus HYBRID shows no relevant

speed-up. The reasons of this behaviour will be investigated as continuation of this thesis work.

	HYBRID vs PURE MPI	**1 MPI + 1 GPU vs PURE MPI**	**2 MPI + 2 GPU vs PURE MPI**	**2 MPI + 2 GPU vs HYBRID**
`Mesh 16M`	1.14	3.56	2.03	1.79
`Mesh 32M`	0.77	2.14	7.05	9.17
`Mesh 64M`	1.52	-	4.27	2.81

Table 5.9: Full speed-ups of `GRAD` during the First Iteration (HYBRID = 10 MPI × 8 threads)

Table 5.10 shows the overall results of the second iteration. It can be seen that on average these are slightly lower than in the first iteration. However, the performance is in line with that obtained in the previous iteration. In particular, it can be observed that the speed-up in the 16 mesh case with 1 GPU is about half that obtained on 32 mesh and 2 GPUs. Moreover a decrement in the performance in the case of mesh 32 and 1 GPU and mesh 64 and 2 GPU can be noticed.

	HYBRID vs PURE MPI	**1 MPI + 1 GPU vs PURE MPI**	**2 MPI + 2 GPU vs PURE MPI**	**2 MPI + 2 GPU vs HYBRID**
`Mesh 16M`	1.02	2.74	2.82	2.78
`Mesh 32M`	2.12	0.85	5.11	2.41
`Mesh 64M`	0.73	-	1.66	2.26

Table 5.10: Full speed-ups of `GRAD` during the Second Iteration (HYBRID = 10 MPI × 8 threads)

Chapter 6

Conclusions

During this project a simple Proof-of-Concept mini-app has been developed and uses aiming to demonstrate that adopting standard C++ parallelization techniques in the OpenFOAM framework is possible and performance improvements can be achieved. We selected the gradient computation, a calculation really frequent in OpenFOAM simulations, and analyzed it in detail. The most computationally expensive routines were identified, analyzed and modified to be compliant with C++11 standard and so executed in parallel. After checking the correctness of the results, benchmark simulations were done on the system NVIDIA Arm HPC Developer Kit.

From the benchmark simulations, it can be observed that the performance of the gradient computation can be improved using both multi-threading as well as GPU offloading. From tables 5.9 and 5.10, the best performance on average is that obtained by performing the calculation on the GPUs. Speed-ups varying from 1.66x to 5.11x compared to PURE MPI and from 2.26x to 2.78x compared to HYBRID. These speed-ups include compute time on GPU, time spent in kernel launch and synchronisation, time spent in managing memory accesses (with or without pre-fetching), overheads added by the profiling tools. In all scenarios, enabling GPU offload has produced a positive overall speed-up.

This work has given us an in-depth insight into the latest technological developments in High Performance Computing. It has demonstrated the possibility to port key OpenFOAM routines on GPUs using ISO C++ stdpar and language parallelism with some modification in the source code but without requiring to change completely the underlying algorithm, the data structure and the execution flow.

Currently only a small part of the overall OpenFOAM codebase runs on GPUs (the gradient evaluation). In the near future, we plan to extend to other routines of immediate interest and perform scalability tests on multiple nodes. The work caught the attention of OpenCFD, the company maintaining OpenFOAM codebase, confirming the approach based on standard

ISO C++ parallelism has potential to become mainstream and widely adopted. Considering the size of OpenFOAM codebase, we envision the involvement of a larger group of active and interested developers to increase the pace of development and expand quickly code capabilities on GPU. We aim to publish an extension of this work in a journal or conference proceeding.

Appendix A

Execution Times for

The present Appendix collects all timings (in milliseconds) of all experiments conducted. These timings are average of minimum three independent executions.

A.1 Global Execution Times for

	PURE MPI	**HYBRID**	**1 MPI + 1 GPU**	**2 MPI + 2 GPU**
`Mesh 16M`	59.45	52.30	16.71	29.29
`Mesh 32M`	145.44	189.11	68.12	20.63
`Mesh 64M`	292.78	192.86	-	68.64

Table A.1: Global Execution Times for first iteration (HYBRID = 10 MPI $\times$ 8 threads)

	PURE MPI	**HYBRID**	**1 MPI + 1 GPU**	**2 MPI + 2 GPU**
Mesh 16	24.794	24.3744	9.0392	8.7784
Mesh 32	48.738	22.9572	57.3998	9.5362
Mesh 64	98.548	134.2658	-	59.506

Table A.2: Global Execution Times for second iteration (HYBRID = 10 MPI $\times$ 8 threads)

A.2 Execution Times for `Mesh 16M`

NVTX tag	PURE MPI	HYBRID	1 MPI + 1 GPU	2 MPI + 2 GPU
`WFC`	14.843	4.7424	1.767	0.926
`WSC`	10.151	0.2286	0.167	0.2418
`DFC`	5.583	1.7616	0.681	0.3534
`GFC`	12.232	3.8398	1.723	1.2128
`GSC`	0.352	0.081	0.310	1.1864
`GD`	3.746	1.4086	0.660	0.3472

Table A.3: Times first iteration `Mesh 16M` (HYBRID = 10 MPI × 8 threads)

NVTX tag	PURE MPI	HYBRID	1 MPI + 1 GPU	2 MPI + 2 GPU
`DFC`	5.829	2.0298	0.67	0.348
`GFC`	11.984	3.9002	1.748	1.0314
`GSC`	0.343	0.518	0.196	0.6968
`GD`	4.023	1.653	0.658	0.3402

Table A.4: Times second iteration `Mesh 16M` (HYBRID = 10 MPI × 8 threads)

A.3 Execution Times for `Mesh 32M`

]NVTX tag	PURE MPI	HYBRID	1 MPI + 1 GPU	2 MPI + 2 GPU
`WFC`	29.502	10.229	3.512	1.7814
`WSC`	20.727	1.5152	0.227	0.2234
`DFC`	11.488	2.9356	1.333	0.6956
`GFC`	24.308	8.0206	3.562	1.7674
`GSC`	0.756	0.9828	0.414	0.7042
`GD`	7.757	2.057	1.319	0.7274

Table A.5: Time first iteration `Mesh 32M` (HYBRID = 10 MPI × 8 threads)

NVTX tag	PURE MPI	HYBRID	1 MPI + 1 GPU	2 MPI + 2 GPU
DFC	11.627	4.3764	1.339	0.6818
GFC	24.542	8.8412	3.559	1.8068
GSC	0.765	0.3626	0.210	0.4624
GD	6.92	2.347	1.314	0.7296

Table A.6: Times second iteration `Mesh 32M` (HYBRID = 10 MPI $\times$ 8 threads)

A.4 Execution Times for `Mesh 64M`

NVTX tag	PURE MPI	HYBRID	2 MPI + 2 GPU
WFC	60.198	18.8354	3.5236
WSC	43.135	1.4854	0.2796
DFC	23.39	6.8086	1.347
GFC	47.222	18.0108	3.5964
GSC	1.248	0.1698	0.5092
GD	14.593	5.728	1.3274

Table A.7: Times first iteration `Mesh 64M` (HYBRID = 10 MPI $\times$ 8 threads)

NVTX tag	PURE MPI	HYBRID	2 MPI + 2 GPU
DFC	23.246	8.2082	1.3534
GFC	52.78	20.4146	3.5952
GSC	1.117	0.2456	0.5284
GD	12.355	7.3548	1.3218

Table A.8: Times second iteration `Mesh 64M` (HYBRID = 10 MPI $\times$ 8 threads)

54

Bibliography

[1] Massimiliano Culpo. "Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters". In: *Partnership for Advanced Computing In Europe* (2012).

[2] I. Spisso , G. Amati , V. Ruggero , C. Fiorina. *Porting, optimization and bottleneck of openFOAM in KNL*. Tech. rep. Intel eXtreme Performance Users Group (IXPUG), 2018.

[3] Giorgio Amati Ivan Spisso. *HPC Comparison of Hypre vs Pstream as external linear algebra library for OpenFOAM*. ESI get it right, 2018.

[4] *Ampere Altra Multi-Core Processor Features*. Available on line. 2022. URL: `https://d1o0i0v5q5lp8h.cloudfront.net/ampere/live/assets/documents/Altra_Rev_A1_DS_v1.30_20220728.pdf`.

[5] *NVIDIA Arm HPC Developer Kit*. Available on line. URL: `https://developer.nvidia.com/arm-hpc-devkit`.

[6] Anthony Williams. *C++ Concurrency In Action*. Manning, 2019.

[7] Rainer Grimm. *Concurrency with Modern C++*. Packt, 2019.

[8] Nikolay Sakharnykh. *Beyond GPU Memory Limits with Unified Memory on Pascal*. Tech. rep. NVIDIA, 2016. URL: `https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/`.

[9] *NVIDIA Nsight Systems*. Available on line. URL: `https://developer.nvidia.com/nsight-systems`.

[10] *NVIDIA Developer Tools*. Available on line. URL: `https://developer.nvidia.com/tools-overview`.

[11] *NVIDIA Tools Extension (NVTX)*. Available on line. URL: `https://docs.nvidia.com/nvtx/index.html`.

[12] Mark Harris. *Unified Memory for CUDA Beginners*. Tech. rep. NVIDIA, 2017. URL: `https://developer.nvidia.com/blog/unified-memory-cuda-beginners/`.

[13] Jeff Larkin. *Developing Accelerated Code with Standard Language Parallelism*. Tech. rep. NVIDIA, 2017. URL: `https://developer.nvidia.com/blog/developing-accelerated-code-with-standard-language-parallelism/`.

[14] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. "Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems". In: *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems held in conjunction with Supercomputing (PMBS)*. in press. IEEE, 2022.

[15] Jonas Latt, Christophe Coreixas , Joel Beny. "Cross-platform programming model for many-core lattice boltzmann simulations". In: *PLoS ONE* (2021). URL: `https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0250306`.

[16] *Thrust*. Available on line. URL: `https://developer.nvidia.com/thrust`.

[17] Mark Harris. *Unified Memory in CUDA 6*. Tech. rep. NVIDIA, 2013. URL: `https://developer.nvidia.com/blog/unified-memory-in-cuda-6/`.

[18] Christopher J. Greenshields. *OpenFOAM, The OpenFOAM Foundation. User Guide version 10*. OpenFOAM Foundation Ltd, 2022.

[19] Simone Bnà, Ivan Spisso, Mark Olesen and Giacono Rossi. "PETSc4FOAM: A Library to plug-in PETSc into the OpenFOAM Framework". In: *Partnership for Advanced Computing In Europe* (2020).

[20] Matt Martineau, Stan Posey and Filippo Spiga. *AmgX GPU Solver Developments for OpenFOAM*. ESI get it right, 2021.

[21] F. Moukalled, L.Mangani and M.Darwish. *The Finite Volume Method in Computational Fluid Dynamics. An advanced Introduction with OpenFOAM and Matlab*. Springer, 2016.

[22] Jasak Hrvoje. "Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows". PhD thesis. Imperial College of Science, Technology and Medicine, 1996.

[23] H.K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics. THE FINITE VOLUME METHOD*. PEARSON, 2007.

[24] *SimpleFOAM*. URL: `https://openfoamwiki.net/index.php/SimpleFoam`.

[25] Daniel Molinero Hernandez et al. "Multi GPU Implementation to Accelerate the CFD Simulation of a 3D Turbo-Machinery Benchmark Using the RapidCFD Library". In: Dec. 2019, pp. 173–187. ISBN: 978-3-030-38042-7. DOI: `10.1007/978-3-030-38043-4_15`.

[26] Julian Seward, Nicholas Nethercote, Tom Hughes, Jeremy Fitzhardinge, Josef Weidendorfer, Paul Mackerras, Greg Parker, Dirk Mueller, Robert Walsh, Bart Van Assche , Cerion Armour-Brown et al. *Valgrind Documentation*. English. Version Version 3.20.0. GNU Free Documentation License. 396 pp. October 24, 2022.

[27] Josef Weidendorfer. *Kcachegrind*. URL: `https://kcachegrind.sourceforge.net/html/Home.html`.

[28] *OpenFOAM guide/SurfaceInterpolation*. Available on line. URL: https://openfoamwiki.net/index.php/OpenFOAM_guide/SurfaceInterpolation.

[29] *RapidCFD Github Repository*. Available on line. URL: https://github.com/Atizar/RapidCFD-dev.

Acknowledgments

In this work, I had the pleasure of working closely with Filippo Spiga, Giovanni Stabile and Matthew Martineau.
To all of them goes my sincere thanks, without their ideas, support and knowledge this work would not have been possible.
In particular, I would like to thank Filippo for the original idea of including the new parallelism standard in OpenFOAM and for his support throughout the work. Thanks go to Giovanni for sharing so much knowledge of OpenFOAM in order to better understand the code and be able to modify it effectively. Thanks to Matthew for providing excellent technical support by accelerating the changes needed to bring the code to the GPU.