Escola Tècnica Superior d'Enginyeries Industrial i Aeronàutica de Terrassa

UNIVERSITAT POLITÈCNICA DE CATALUNYA

POLYTECHNICAL UNIVERSITY OF CATALONIA

A PROJECT FOR THE TITLE OF
INDUSTRIAL ENGINEER

Study of an efficient parallel implementation of a CFD code based on algebraic operators

author
Xavier Álvarez Farré

supervised by
Dr. A. Oliva, Dr. F. X. Trias and Dr. R. Borrell

June 16, 2016

Abstract

The present project is about developing, implementing and evaluating an efficient parallel CFD code based on algebraic operators. The principal algebraic kernels required are: i) the dot product, ii) the generalized vector addition, iii) the sparse matrix-vector product. The code developed for this project must meet three requirements to suit current needs. Namely, apart from being modular in order to be suitable for different applications and engineering problems, it must also be efficient and portable for different parallel computing systems.

Acknowledgement

The completion of this project could not have been possible without the generous support of so many people whose names may not all be mentioned. Their contributions are sincerely appreciated and gratefully acknowledged. However, I would like to express my deep appreciation and indebtedness to:

All the members of the CTTC (Heat and Mass Transfer Technological Center) for their help and assistance along this year and also for the great moments like the coffee breaks, the weekend getaways and the summer beers and barbecues.

Dr. Asensi Oliva for betting on me and allowing me in the CTTC.

Dr. Ricard Borrell and Dr. Xavier Trias for guiding me through this undertaking and for the knowledge and motivation they have taught to me.

My friends and specially Mr. Javi Arribas for countless moments together from eternal nights studying at the library to unforgettable trips.

Ms. Elisabeth Pino for her endless love and support. Thanks for making me look at the beautiful side of life.

My family, specially my mother for her unconditional love and sacrifice and my father for his overcoming spirit. Thanks for giving me the best life and education I could wish.

“The world is a fine place and worth fighting for.”

- Ernest Hemingway

Nomenclature

Symbol	Definition
	Basic Nomenclature
$\mathbf{u}$	Bold font is used to represent a vector.
$\hat{u}$	The small hat is used to represent a dimensionless parameter.
$\overline{u}$	The over-line is used to represent a matrix.
$\dot{u}$	The dot is used to represent a temporal derivative.
	Numerical Analysis Nomenclature
φ^n	Super-index of time. Value of φ at present time.
φ^{n+1}	Super-index of time. Value of φ at later time.
φ^{n-1}	Super-index of time. Value of φ at earlier time.
φ_i	Spatial sub-index. Value of φ in the control volume i.
φ_f	Spatial sub-index. Value of φ at the control surface f.
φ_{nb}	Spatial sub-index. Value of φ at the neighbor node nb.
δ	Spatial increment.
Δt	Increment of time.
S_f	Surface f.
	Mathematical Nomenclature
$\partial/\partial t$	Partial derivative respect to time.
∇	Gradient.
$\nabla\cdot$	Divergence.
∇^2	Laplace operator.
$\hat{\mathbf{n}}$	Unit vector normal to surface.
$\sum\limits_{f\in F(c)}$	Summation for all the surfaces on the control volume.

Symbol	Definition	Units
	Fluid Dynamics Nomenclature	
Φ	Generic extensive variable.	
φ	Generic intensive variable.	
m	Mass.	$[kg]$
ρ	Density.	$[kg/m^3]$
μ	Dynamic viscosity.	$[Pas]$
ν	Cinematic viscosity.	$[m^2/s]$
λ	Thermal conductivity.	$[W/mK]$
α	Thermal diffusivity.	$[m^2/s]$
β	Thermal expansion coefficient.	$[K^{-1}]$
c_P	Specific heat capacity at a constant pressure.	$[J/kgK]$
$\mathbf{u}$	Velocity.	$[m/s]$
$\mathbf{g}$	Gravity.	$[m/s^2]$
T	Temperature.	$[K]$
p	Pressure.	$[Pa]$
t	Time.	$[t]$
$\dot{Q}$	Energy source.	$[W/m^3]$
$\dot{q}$	Flow rate.	$[m^3/s]$
Γ	Generic diffusive factor.	

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Physics, in Ancient Greek, means knowledge of nature. Physics is a natural science that involves the study of matter and its motion through space and time. Engineering is the application of mathematics, empirical evidences and scientific knowledge in order to research, improve or invent things. In order to describe a system or a phenomenon, scientists and engineers design and use mathematical models.

Many of the engineering studies lead to mathematical models whose analytical resolution is unknown. The need to provide by the community of physicists and engineers, a useful solution to these problems promotes the study and development of numerical methods and computing systems. Numerical analysis is the study of algorithms or numerical methods able to obtain an approximate solution of complex mathematical models. The fact that numerical methods calculate an approximate solution of the problem implies that there is always an associated error. The two main factors related to the quality of the result obtained from the numerical analysis are accuracy and precision. In one hand, better accuracy is obtained using a better mathematical model. On the other hand, better precision is obtained using a better numerical method.

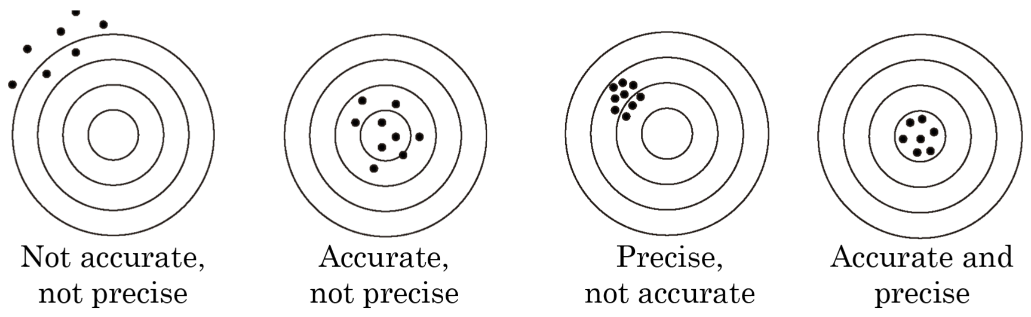


Figure 1.1: Accuracy vs Precision

The emergence of computers (machines capable of performing millions of simple binary operations per second) boosted the use and development of numerical methods because together they allow to quickly perform huge mathematical calculations. Thereby computers reduce the need for analytical calculations or experimental results for the development of engineering projects; thermal loads, structural or financial calculations, among others, are increasingly fast and accurate thanks to the constant evolution of computers and numerical methods.

Gordon Moore, co-founder of Intel, predicted in 1965 [1] that every year the number of transistors inside processors will double as well as computing power. This prediction was called “Moore’s Law”. Later, Moore readjusted the magnitude of its prediction several times but in essence it remained the same: computing progresses at a frantic pace. The main drawback of this constant growth is that the energetic cost is no exception; it also increases in almost the same magnitude.

The fact that the required electric power increases proportionally with computing capacity generates several constraints when performing numerical calculations; when a project is carried out, it is necessary to decide the mathematical model and numerical method depending on the accuracy and precision desired and assess whether the costs for calculating are worth it. At this point, the efficiency of the code implemented for carrying out the simulation plays a key role: it is the only remaining factor which could minimize the energetic cost of the computation. For this reason, the code should take full advantage of the available computing resources.

1.1 Object

The object of this project is to implement an efficient parallel CFD code based on three algebraic operators: dot product, generalized vector addition and sparse matrix-vector product.

1.2 Motivation

One of the sciences most benefited by numerical analysis is fluid mechanics. The Navier-Stokes equations describe the behavior of fluids and are considered the most challenging equations of classical physics. The Computational Fluid Dynamics (CFD) is responsible for solving, using a numerical method, the system of partial differential equations mentioned above (Navier-Stokes) to try to understand the behavior of fluids in various fields of knowledge such as: aeronautics, automotive, combustion, blood flow, weather and many more. In an operator-based formulation, the solution of the finite-volume discretization of Navier-Stokes equations involves three main algebraic operators: i) the dot product, ii) the generalized vector addition, iii) the sparse matrix-vector product. Hence developing a computational application with an algebraic kernel able to perform these three operations is very meaningful. Furthermore, this computing tool is very suitable not only for CFD applications, but also for every engineering problem which requires solving partial differential equations that can be written in an operator-based formulation, thus it is very convenient that codes are modular in order to be easily implemented into different engineering applications.

The discretized Navier-Stokes system of equations results in a large system of simple algebraic equations. This system could have millions, billions or even more equations depending on the size of the problem. Such a large set of equations is impossible to solve by a human but not by a computer: a current laptop is able to perform the order of 5.000.000.000 floating point operations per second. However, as demand for computing power grows in the fields of science and engineering in order to solve more challenging cases and to obtain more precise results, the computing power of a laptop is nowadays considered tiny. Simultaneously, high performance computing (HPC) is becoming very important. HPC most generally refers to the practice of aggregating computing power in order to solve large problems efficiently, reliably and quickly. To achieve this, the HPC relies on technologies such as parallel computing for developing computer clusters and supercomputers. A supercomputer could become up to 1,000,000 times faster than a laptop.

A powerful hardware is not the only requirement for a fast calculation; the efficiency of the software must be taken into account too. Every advance in computing technologies adds new features and instructions to computing systems which make it possible for programmers to improve the efficiency of their codes despite those new features are very complex and require a high level of computing and programming skills. Moreover, although there are some standards in this industry, this new features and instructions are not all the same from every manufacturer and architecture. For this reason, high efficiency in software depends heavily on the features of the computing system used such as its manufacturer, architecture, components and operative system among others. The term portability referred to the software means that it is able to perform well in many different platforms and architectures. To design a portable and efficient code is usually hard since it must have implemented every function in the most efficient way for each type of architecture of destiny.

There are some linear algebra standards already implemented in many libraries which are efficient and optimized for particular systems. They contain lots of algebraic operators for all kind of applications. One of the best known is the Basic Linear Algebra

Subprograms (BLAS); BLAS is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations and matrix multiplication. Nowadays there are a lot of different BLAS implementations, such as ATLAS, CBLAS and GOTOBLAS among others. The CBLAS implementation is an interface between the Fortran BLAS and C language and has been used in this project as reference for performance tests. Most of the implementations of the BLAS are Open Source code libraries, that means everybody can download, install, use and improve them freely. This fact allows programmers from all over the world to implement and optimize the BLAS for every existing platform.

In conclusion, an algebraic library suitable for CFD problems should provide, at least, the following algebraic operators:

- Dot Product,
- Generalized Vector Addition,
- Sparse Matrix-Vector Product,

and meet the following requirements:

- To be Modular,
- To be Portable,
- To be Efficient.

To design this library once and keep it updated and maintained would make it possible for other engineers to develop simulation tools without worry anymore about the implementation of the algebraic operators.

1.3 Scope

The intended tasks for this project are:

- To study the physics and mathematics related to CFD laminar problems.
- To study the numerical methods that are required for solving a CFD problem.
- To write the discrete governing equations in an operator-based formulation.
- To study the computing system as an abstract object: black-box model.
- To study the different bottlenecks involved in computational science.
- To develop an efficient, parallel computational application able to perform the three algebraic operators object of this project.
- To evaluate the efficiency and performance achieved with every developed operator.

1.4 Basic Specifications

The basic specifications for this project are:

- To discretize the governing equations in the forced convection scenario.
- To use two-dimensional Cartesian grids for the spatial discretizations.
- To program in C++ language.
- To use SIMD instructions for single core parallelization.
- To use OpenMP interface for shared memory parallelization.

- To use OpenMP+MPI interface for distributed memory parallelization.
- To evaluate the main-memory bounded performance of the operators.

Chapter 2

Previous Concepts

Throughout this chapter some previous concepts that are necessary for a better understanding of the later topics are detailed. Sparse matrices and its different structures are analyzed. Also the basics of open multi-processing platform (OpenMP) and message passing interface (MPI) are introduced.

2.1 Sparse Matrices

A sparse matrix is a matrix in which most of the elements are zeroes. By contrast, if most of the elements are non-zero, then the matrix is called dense. The amount of non-zero elements is known as number of non-zero (NNZ). Sparsity is defined as the ratio of NNZ to the total number of elements of the matrix (see Equation 2.1).

$$\text{Sparsity} = \frac{\text{NNZ}}{\text{N}_{\text{elements}}} \tag{2.1}$$

When solving engineering problems using numerical methods it is common to work with large sparse matrices. In addition, these matrices usually have symmetrical properties. Storing and manipulating sparse matrices as dense matrices, that is storing and manipulating all the zeroes, results in a waste of computing capacity, memory and electric power. In order to avoid this heavy waste of resources, sparse data structures are designed. The Figure 2.1 exposes three different types of sparse matrices; only the black dots represent non-zero elements hence the white space is full of zeroes.

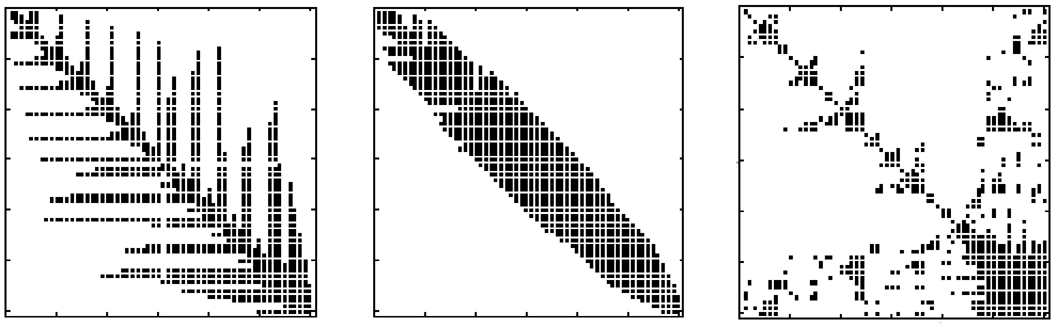


Figure 2.1: Sparse Matrix examples.

In computing science, sparse matrices must be stored within specific data structures. A data structure is a particular way of organizing data inside a computer so that the computer is able to use the data efficiently. In this project, data structures are implemented as C++ objects that aims to store all the information of the matrix (i.e. the non-zero elements and the corresponding row and column indexes among many other parameters) in the most efficient way. There are many different types of data structures for storing sparse matrices but there is not any which is the best. Furthermore, the effectiveness of the data structure depends heavily on many characteristics of the sparse matrix (e.g. its sparsity, symmetry or how the NNZ are distributed within). In addition, the efficiency of the data structure also depends on the application in which it is used. The Table 2.1 introduces the four data structures have been analyzed in the sections below.

Format	Number of Entries
COO	3nnz
CRS	$2\text{nnz} + \text{n}_{\text{rows}}$
CDS	$\text{nnz} + \sum_{i=1}^{p-1} i + \sum_{i=1}^{q-1} i$
sparse_xlf_2d	$\text{nnz} + 2\text{n} + 2\text{m}$

Table 2.1: Sparse matrix data structures summary.

2.1.1 COO, Coordinate Format

The COO data structure consists in an associative array which stores as keys the row-column pairs and links them to their corresponding non-zero value. Each key (row-column pair) can only be associated to one entry [2]. Then, using a COO structure, the matrix

$$\begin{pmatrix} 3 & 5 & 0 & 0 & 0 & 0 \\ 2 & 1 & 6 & 0 & 0 & 0 \\ 0 & 3 & 9 & 1 & 0 & 0 \\ 0 & 0 & 1 & 7 & 3 & 0 \\ 0 & 0 & 0 & 8 & 2 & 3 \\ 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix},$$

is stored as exposed in Table 2.2.

Value	3	5	2	1	6	3	9	1	1	7	3	8	2	3	1	2
Row	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6	6
Column	1	2	1	2	3	2	3	4	3	4	5	4	5	6	5	6

Table 2.2: COO data structure.

A matrix stored with COO structure has to store 3NNZ entries: NNZ for the non-zero elements and 2NNZ for the corresponding pair of row-column keys. The COO format is a good format for constructing randomly any sparse matrix but it is not efficient enough when iterating over non-zero values. Furthermore, after constructing the sparse matrix with a COO structure it is usually converted into a format more efficient for processing.

2.1.2 CRS, Compressed Row Storage

The CRS structure consists of three one-dimension arrays. The first array stores the non-zero values of the matrix row wise. The second one stores the column index associated to each non-zero element. Finally, the third array only stores one value for each row that is the accumulated amount of non-zero elements in the previous rows; in other words, the third array stores for each row the counter of the first non-zero element in the row [3]. This way, if the third value in the row array is a 5 and the fourth is an 8 (see Table 2.3 for instance), it is trivial to know that the 6th, 7th and 8th non-zero elements belong to the third row. Then, using a CRS structure, the matrix

$$\begin{pmatrix} 3 & 5 & 0 & 0 & 0 & 0 \\ 2 & 1 & 6 & 0 & 0 & 0 \\ 0 & 3 & 9 & 1 & 0 & 0 \\ 0 & 0 & 1 & 7 & 3 & 0 \\ 0 & 0 & 0 & 8 & 2 & 3 \\ 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix},$$

is stored as exposed in Table 2.3.

Value	3	5	2	1	6	3	9	1	1	7	3	8	2	3	1	2
Column	0	1	0	1	2	1	2	3	2	3	4	3	4	5	5	6
Row	0	2	5	8	11	14										

Table 2.3: CRS data structure.

A matrix stored with CRS structure have to store 2NNZ + N_{row} entries (where N_{row} is the number of rows): NNZ for the non-zero elements, NNZ for the column indexes and N_{row} for the row values. Then, the CRS structure uses less memory than COO structure. However, saving memory is not its biggest advantage but the fact that the processing of the data using CRS becomes much more efficient. Moreover, CRS structure is one of the most efficient data structures when calculating the sparse matrix-vector product row wise as it facilitates iterating over entire rows.

2.1.3 CDS, Compressed Diagonal Storage

The CDS structure consists of a set of one-dimension arrays. Each array stores all the elements of one diagonal of the matrix. This structure works pretty well for banded matrices, that is matrices that have a constant bandwidth from row to row.

The matrix $A = (a_{i,j})$ is banded if there are two non-negative integer constants p and q, called left and right half-bandwidth, such that $a_{i,j} = 0$ if $j < i - p$ or $j > i + q$.

The storage element for this structure is usually a two-dimension array which contains the non-zero values such as A[d][p+1+q], where [d] refers to the number of elements in the main diagonal and [p+1+q] is the total number of diagonals. Notice that, since the rest of diagonals have less elements than the main one the CDS structure may contain some fictitious zero elements which do not correspond to any real matrix location and this implies a small waste of memory. However, adding these non-existing zeroes is not detrimental: it facilitates the processing of the matrix by making all elements of the same row to have the same array index and hence reducing the pointer arithmetic's overhead. Then, using a CDS structure, the matrix

$$\begin{pmatrix} 3 & 5 & 0 & 0 & 0 & 0 \\ 2 & 1 & 6 & 0 & 0 & 0 \\ 0 & 3 & 9 & 1 & 0 & 0 \\ 0 & 0 & 1 & 7 & 3 & 0 \\ 0 & 0 & 0 & 8 & 2 & 3 \\ 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix},$$

is stored as exposed in Table 2.4.

$A[i][0]$	0	2	3	1	8	1
$A[i][1]$	3	1	9	7	2	2
$A[i][2]$	5	6	1	3	3	0

Table 2.4: CDS data structure.

A matrix stored with CDS structure have to store NNZ $+\sum_{i=1}^{p-1} i + \sum_{i=1}^{q-1} i$ entries: NNZ for the non-zero elements and the rest for the fictitious zeroes. The big advantage of using a CDS structure is that the sparse matrix-vector product, when calculated diagonal wise becomes much more efficient than other. Also the memory requirements for a CDS structure are pretty lower than for the structures analyzed previously. However, the CDS structure presents a big disadvantage: it can only be used for banded matrices.

2.1.4 sparse_xlf_2d, Sparse XLF Two-Dimensional Structure

Basic previous knowledge about finite volume discretization is needed for a better understanding of the concepts concerning this structure. The penta-diagonal coefficient matrix that appears in two-dimensional CFD cases discretized with structured meshes do not fit correctly into the CDS structure due to its characteristics: two of the five diagonals are separated from the main one, so it does not satisfy the condition for banded matrix. However, with some small modifications on the CDS it can be easily applied to this specific CFD cases. The sparse_xlf_2d structure is the proposal for this project. Since the number of diagonals and its location are known from the discretization, the diagonals c be initially labeled as south, west, center, east and north referring to how the nodes are related between them. Then, using the sparse_xlf_2d structure, the matrix

$$\begin{pmatrix}
a_{c1} & a_{e1} & 0 & a_{n1} & 0 & 0 & 0 & 0 & 0 \\
a_{w2} & a_{c2} & a_{e2} & 0 & a_{n2} & 0 & 0 & 0 & 0 \\
0 & a_{w3} & a_{c3} & 0 & 0 & a_{n3} & 0 & 0 & 0 \\
a_{s4} & 0 & 0 & a_{c4} & a_{e4} & 0 & a_{n4} & 0 & 0 \\
0 & a_{s5} & 0 & a_{w5} & a_{c5} & a_{e5} & 0 & a_{n5} & 0 \\
0 & 0 & a_{s6} & 0 & a_{w6} & a_{c6} & 0 & 0 & a_{n6} \\
0 & 0 & 0 & a_{s7} & 0 & 0 & a_{c7} & a_{e7} & 0 \\
0 & 0 & 0 & 0 & a_{s8} & 0 & a_{w8} & a_{c8} & a_{e8} \\
0 & 0 & 0 & 0 & 0 & a_{s9} & 0 & a_{w9} & a_{c9}
\end{pmatrix},$$

is stored as exposed in Table 2.5.

South	0	0	0	a_{s4}	a_{s5}	a_{s6}	a_{s7}	a_{s8}	a_{s9}
West	0	a_{w2}	a_{w3}	0	a_{w5}	a_{w6}	0	a_{w8}	a_{w9}
Center	a_{c1}	a_{c2}	a_{c3}	a_{c4}	a_{c5}	a_{c6}	a_{c7}	a_{c8}	a_{c9}
East	a_{e1}	a_{e2}	0	a_{e4}	a_{e5}	0	a_{e7}	a_{e8}	0
North	a_{n1}	a_{n2}	a_{n3}	a_{n4}	a_{n5}	a_{n6}	0	0	0

Table 2.5: sparse_xlf_2d data structure

This way, the sparse_xlf_2d consists of a set of five arrays: south, west, center, east, north. Each array contains one of the diagonals of the previous coefficient matrix. Then, a matrix stored with the sparse_xlf_2d structure have to store NNZ+2N+2M entries, where N and M are the number of vertical and horizontal nodes respectively: NNZ for the non-zero elements, 2N for the interleaved zeroes corresponding to the non-existing east/west wall coefficients and 2M for the zeroes corresponding to the non-existing north/south wall coefficients. Note that this structure also stores some fictitious zeroes but again, adding these non-existing zeroes is not detrimental: it facilitates the processing of the matrix by making all elements of the same row to have the same array index and this helps reducing the pointer arithmetic's overhead.

In conclusion, the sparse matrix-vector product using the sparse_xlf_2d data structure is very efficient in comparison to COO and CRS. In one hand, performing the operation with the COO data structure requires many tasks for each entry since the elements are not stored following a pattern. Then, the row and column indexes must be read for every product during the operation. On the other hand, the CRS data structure allows to operate very efficiently since it facilitates to iterate over all the elements of a row and the operation is performed by carrying out as many loops as rows has the matrix. Finally, the sparse matrix-vector product using the sparse_xlf_2d does not read either the row or the column indexes since the pattern of the matrix is previously known hence the entire operation can be performed by carrying out a single loop.

2.2 Open Multi-Processing for Shared Memory Systems

Open multi-processing (OpenMP or OMP) is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments and Oracle Corporation. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. In Figure 2.2 a shared memory system is represented schematically. The API supports C/C++ and Fortran on a wide variety of architectures [4] [5].

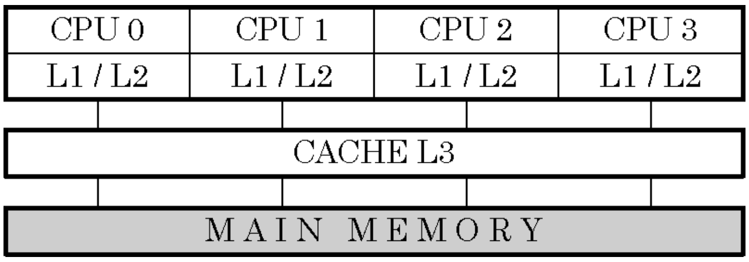


Figure 2.2: Scheme of a shared memory system.

In other words, OpenMP is an implementation of multi-threading, a method of parallelizing whereby a master thread sets a specified number of slave threads and the system divides a task among them. The threads then run concurrently with the runtime environment allocating threads to different processing units [6]. The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that causes the threads to form before the section is executed. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program. The Figure 2.3 shows the OpenMP threading process.

By default in an OpenMP application, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

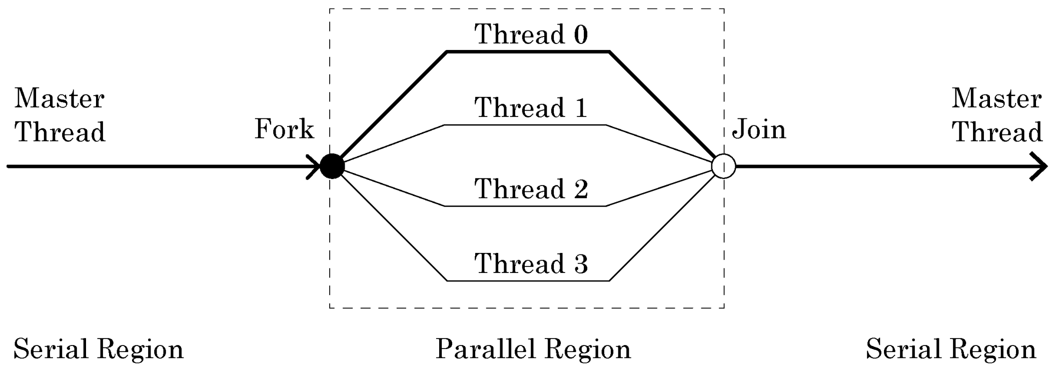


Figure 2.3: Scheme of the OpenMP threading.

To compile a program using OpenMP, the system must have an implementation of OpenMP installed. The header omp.h must be included in the C++ file and the flag -fopenmp needs to be added to the compiling line in order to link the library and create the executable. The example in Code 2.1 shows how to compile and execute a simple OpenMP application in the Ubuntu terminal.

```
$ g++ -c -fopenmp executable.cpp -o executable
$ ./executable
```

Code 2.1: Compiling and executing an OpenMP application in the Ubuntu terminal.

Below is a short summary of the concepts about OpenMP required for the realization of the project's code and examples of its implementation. See the OpenMP manuals for further information [7, 4].

2.2.1 The Parallel Region

To create a parallel region within the code, a defined preprocessor directive called pragma must be used enclosing all the parallel work. The example in Code 2.2 shows how to create the parallel region in C++. All threads created within the parallel region execute concurrently the parallel_work function.

```
#pragma omp parallel
{
  parallel_work ();
}
```

Code 2.2: OMP Example. Creating the parallel region.

The number of threads created by default by the directive is set with an environment variable called *OMP_NUM_THREADS*. If the environment variable is not modified manually by the user, it usually takes the value of the total number of processors available in the node. However, the environment variable can be set before executing the program as shown in Code 2.3 in order to force the desired number of threads.

```
$ export OMP_NUM_THREADS=4
$ ./executable
```

Code 2.3: Setting the multi-threading environment variable in Ubuntu terminal.

In addition, the programmer is able to specify the number of threads created in a particular parallel region by using an OpenMP clause called num_threads (the OpenMP clauses are introduced later in Section 2.2.2). The example in Code 2.4 shows how to create 8 threads within a parallel region using the num_threads clause. The number can be higher than the number of processors but this does not grant further improvement in performance.

```
#pragma omp parallel num_threads (8)
{
  parallel_work ();
}
```

Code 2.4: OMP Example. Using the num_threads clause.

Finally, a very useful directive is introduced: the #pragma omp for. This directive automatically divides the work within the next loop by distributing the number of iterations between the threads. The OpenMP interface offers many clauses for properly distributing the work within the loop, which are exposed in Section 2.2.2. The example in Code 2.5 shows how to create the parallel loop using the #pragma omp for directive.

```
#pragma omp parallel
  #pragma omp for
  for (int i=0; i<N; ++i) {
    parallel_work ();
  }
}
```

Code 2.5: OMP Example. Using the #pragma omp for directive.

2.2.2 OpenMP Clauses

In order to modify the default parameters of the parallel region and also to decide how does it manage data and work, the user is provided with tools called OMP clauses. These clauses are kind of words that are able to change the default settings when added to the OMP pragmas as shown in the example in Code 2.6. In the Section 2.2.1, the clause num_threads was introduced to show how to modify the default number of threads created within the parallel region.

```
#pragma omp parallel omp_clause (value)
{
  parallel_work ();
}
```

Code 2.6: OMP Example. Using OpenMP clauses.

Clauses for Managing Variables

The clauses default, shared and private are used to specify if a variable is going to be shared between threads, that is visible and accessible by all simultaneously, or if each thread will create a local copy and use it as a temporary variable instead. In one hand, the clauses shared and private can have as input the list of variables for which a specific behavior is desired. On the other hand, the clause default can have the keywords shared, private or none as input in order to set the default behavior for the non-specified variables in the parallel region. If default(none) is set, all the variables that are used inside the parallel region must be previously included in a shared or private clause since no default behavior is known for the variables. To use well this clauses is very important because it can help the programmer to avoid undesirable behaviors such as race condition. In software, race condition happens when multiple threads attempts to access and modify the same variable at the same time causing random results.

The example in Code 2.7 shows a parallel program in which every thread executes a N-step loop with some parallel work inside. In the example, the OpenMP clauses are used to set the integer as private and the other variable as shared within the parallel region. The Figure 2.4 represents how the variables are managed between different threads. Notice that setting the integer as private forces each thread to execute the entire loop. In contrast, if the integer is also declared as shared all threads increment it simultaneously instead of individually and thus the iterations are randomly distributed between all threads.

```
int i;
double x = 1.0;
#pragma omp parallel default (none) shared (x) private (i)
{
  for (i=0; i<N; ++i) {
    parallel_work (x);
  }
}
```

Code 2.7: OMP Example. Setting the behavior of variables.

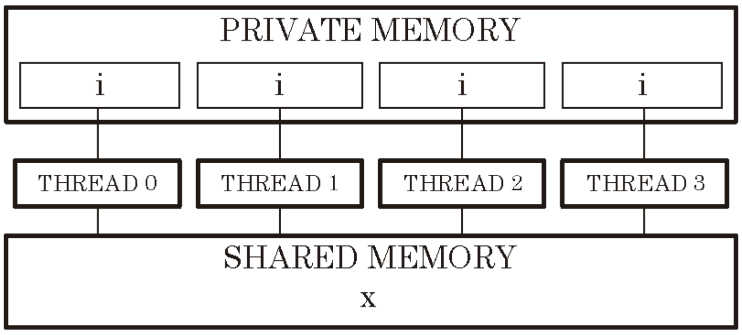


Figure 2.4: Scheme of OpenMP management of variables.

Another important clause for managing the use of the variables in a multi-thread program is the reduction. This clause is a safe way of joining work from all threads after the parallel region. It specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region. Data reduction involves reducing a set of numbers into a smaller set of numbers via a function. For example, reducing the list of numbers [1, 2, 3, 4, 5] with the sum function would produce add([1, 2, 3, 4, 5]) = 15. Similarly, the multiplication reduction would yield mul([1, 2, 3, 4, 5]) = 120. This way, when multiple threads must perform the same operation on the same variable, the clause reduction allows them to operate in a correct order. In the example in Code 2.8, all threads add a to x. Thus as higher is the number of threads, higher is the expected result for the reduction.

```
int x = 0;
int a = 1;
#pragma omp parallel default (none) shared (x,a) reduction (+:x)
{
  x = x + a;
}
```

Code 2.8: OMP Example. Using the reduction clause.

Clauses for Synchronizing

In parallel computing, a barrier is a programming line which forces a group of threads or processes to wait for everyone in the group reaching the barrier. The OpenMP barrier clause makes the threads to wait until all the other threads of the team have reached it. In the example in Code 2.9, any thread starts the parallel_work_2 until the entire parallel group finishes the parallel_work_1.

```
#pragma omp parallel
{
  parallel_work_1 ()
  #pragma omp barrier
  parallel_work_2 ();
}
```

Code 2.9: OMP Example. Using the barrier clause.

In contrast to the OpenMP barriers, there is a clause which allows the threads to keep working without waiting the rest of the group. After some of the OpenMP directives (e.g. #pragma omp for) there is an implied barrier. The nowait clause specifies that threads completing an assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads may encounter a barrier synchronization at the end of the parallel region. In the example in Code 2.10, threads are allowed to start the second loop without waiting the rest of the group to finish the first loop.

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int i=0; i<N; ++i) {
    parallel_work_1 ();
  }
  #pragma omp for
  for (int i=0; i<N; ++i) {
    parallel_work_2 ();
  }
}
```

Code 2.10: OMP Example. Using the nowait clause.

Clauses for Work Sharing

Distributing the tasks of a loop can be a hard job. The effectiveness of the work-sharing depends on many factors as the size of the loop, the size of the chunks, the tasks executed in each iteration and the number of threads among many others. Furthermore, the performance of a parallel code can be even lower than the sequential code if the work-sharing is not properly defined.

The OpenMP platform offers the clause schedule and many options for defining how the threads share the work load. The syntax of the clause is: schedule (type, chunk). The iterations in the parallel region are assigned to threads depending on the scheduling method defined. Four different loop scheduling types can be provided: static, dynamic, guided and auto. When specified, the optional parameter chunk must be a positive integer [8]. In the example in Code 2.11, the scheduling is set as automatic in order to let the compiler decide the work-sharing option.

```
#pragma omp parallel
{
  #pragma omp for schedule(auto)
  for (int i=0; i<N; ++i) {
    parallel_work ();
  }
}
```

Code 2.11: OMP Example. Using the schedule clause.

2.2.3 Manual Work-Sharing

In Section 2.2.2, the clause for distributing the loop within a parallel region and its options has been exposed. However, there is still the possibility of distributing it manually. The OpenMP function omp_get_thread_num returns a positive integer which identifies the number of the thread that is calling the function and is required for manually distributing the work as well as the function omp_get_max_threads, which returns the total number of threads within the parallel region. It is possible to assign each thread a specific part of work if its specific identifier is known. The example in Code 2.12 shows an very simple but illustrative way of manually distributing some generic parallel work.

```
#pragma omp parallel num_threads(2)
{
  int thread_id = omp_get_thread_num ();
  if (thread_id == 1) parallel_work_1 ();
  if (thread_id == 2) parallel_work_2 ();
}
```

Code 2.12: OMP Example. Manual work-sharing.

In this project, the manual work-sharing has also been implemented to the parallel algebraic operators in order to compare the performance obtained with the manual method to the performance obtained with the scheduled method in order to check whether there is any overhead affecting the performance due to the OpenMP automatic scheduling. An example of manually distributing the work within a loop is shown in the example in Code 2.13.

```
#pragma omp parallel default(none) shared(N)
{
  int threads = omp_get_max_threads ();
  int thread_id = omp_get_thread_num ();
  int ini = thread_id*N/threads;
  int end = (thread_id+1)*N/threads;
  for (int i=ini; i<end; ++i) {
    parallel_work ();
  }
}
```

Code 2.13: OMP Example. Manual work-sharing.

2.3 MPI, Message Passing Interface for Distributed Memory Systems

Message Passing Interface (MPI) is a standardized and portable message-passing interface designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open source or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

In contrast to a program parallelized with OpenMP (Section 2.2) in which only the master thread executes the code until it meets the definition of a parallel region and then distributes tasks among multiple processes, a program parallelized with MPI must be run entirely by all processes. The MPI calls are functions that makes any process to send/receive information to/from other process or processes. In order to manage the origin and destination of the messages, each process has a unique identifier. It is very important to understand that all processes are running the same program so every process must know its identifier as well as the total number of processes in order to execute only its corresponding MPI calls. The Figure 2.5 represents a generic distributed memory system which is required for MPI implementations.

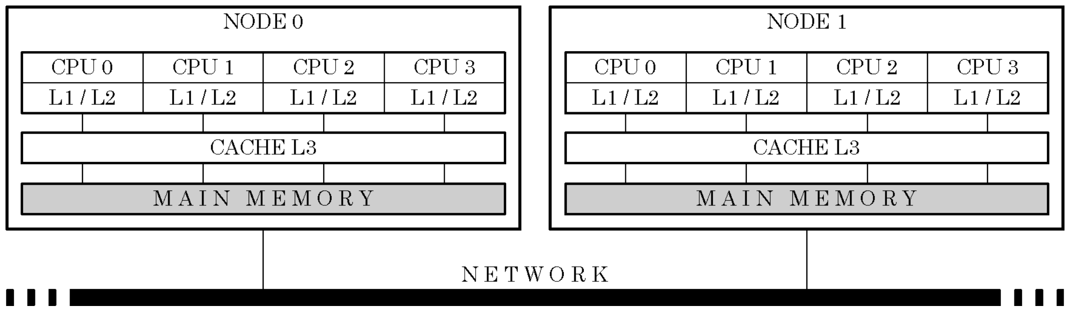


Figure 2.5: Scheme of a distributed memory system.

To compile a program using MPI, the system must have an implementation of MPI installed. First of all, the header mpi.h must be included in the C++ file. Then, the compiler must be changed to mpic++. Finally, the command mpirun and also the required flags must be used for running the application. The flag -n X, where X is a positive integer higher than zero, is added to specify the desired number of processes that will execute the program. The number of requested processes should be as much the number of processors available in the system as shown in Figure 2.6. This way, all requested processes will execute a copy of the same application. The example in Code 2.14 shows how to compile and run a MPI application in the Ubuntu terminal.

```
$ mpic++ -c executable.cpp -o executable
$ mpirun -n 4 ./executable
```

Code 2.14: Compiling and executing a MPI application in the Ubuntu terminal.

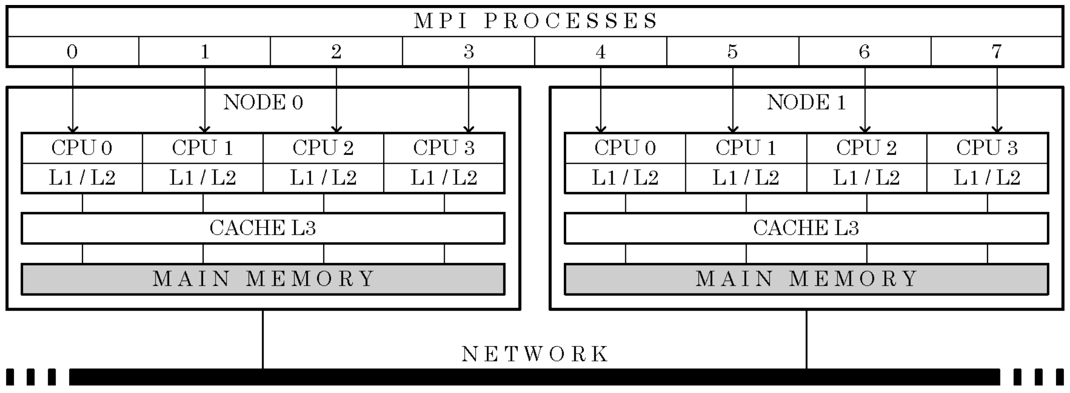


Figure 2.6: Scheme of a MPI process distribution.

Below is a short summary of the concepts of MPI which have been used for the parallelization of the operators and some examples of its implementation. See the MPI user guide and its manpage for further information [9, 10, 11].

2.3.1 The MPI Groups and Communicators

In MPI, a group is an ordered set of process identifiers [11]. The ordering is given by associating with each process identifier a unique rank from 0 to the size of the group minus 1. More specifically, an MPI group is a local representation of a set of MPI processes. MPI groups are represented by the opaque type MPI_Group in C++ applications. This way a process can contain local representations of many MPI groups some of which may not include itself.

MPI defines a rich set of operations on groups; since a group is essentially an ordered set (in the algebraic sense of the word), an application can perform group unions, intersections, inclusions, exclusions, comparisons, and so on. These operations, while not commonly invoked in many user applications, form the backbone of communicator functionality and may be used by the MPI implementation itself.

A communicator is an object that encapsulates all communication among a set of processes. Communicators are represented in MPI C++ programs by the type MPI_Comm. Although communicator is a local MPI object (i.e. it physically resides in the MPI process), it represents a process' membership in a larger process group. Specifically, even though MPI_Comm objects are local, they are always created collectively between all members in the group that the communicator contains. Hence, a process can only have an MPI_Comm handle for communicators of which it is a member.

The MPI default communicator called MPI_COMM_WORLD is created when the function MPI_Init is called. In the MPI manual, the MPI_COMM_WORLD is defined as "all processes the local process can communicate with after initialization (including itself), and is defined once MPI_INIT has been called". Although the specific meaning of this statement varies between different MPI implementations, it generally means that all MPI processes started via mpirun are included in MPI_COMM_WORLD together. At this point, the processors are allowed to call other MPI functions until the function MPI_Finalize is called, which ends with the possibility to call any MPI function. Furthermore, both MPI_Init and MPI_Finalize can be called only once in the whole program so it is advisable that all the tasks of the application are located between them.

The example in Code 2.15 exposes a basic way of initializing MPI and getting the rank of the process as well as the number of processes within the default communicator.

```
int main () {
  int rank;
  int world;
  MPI_Init (NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &world);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Finalize ();
}
```

Code 2.15: MPI Example. Initializing MPI.

2.3.2 The MPI Send & Receive

Sending and receiving are the two foundational concepts of MPI. Almost every single function in MPI can be implemented with basic send and receive calls. MPI's send and receive calls operate in the following manner. First, process A decides a message needs to be sent to process B. Process A then packs up all of its necessary data into a buffer for process B. These buffers are often referred to as envelopes since the data is being packed into a single message before transmission (similar to how letters are packed into envelopes before transmission to the post office). After the data is packed into a buffer, the communication device (which is often a network) is responsible for routing the message to the proper location. The location of the message is defined by the process's rank [12].

Even though the message is routed to B, process B still has to acknowledge that it wants to receive A's data. Once it does this, the data has been transmitted. Process A is acknowledged that the data has been transmitted and may go back to work.

Sometimes there are cases when A might have to send many different types of messages to B. Instead of B having to go through extra measures to differentiate all these messages, MPI allows senders and receivers to also specify message IDs with the message (known as tag). When process B only requests a message with a certain tag number, messages with different tag will be buffered by the network until B is ready for them.

The prototypes of the MPI_Send and MPI_Recv functions are shown in Code 2.16. The first argument is the data buffer, that is a pointer to the variable which is going to be sent. The second and third arguments describe the count and type of elements that reside in the buffer. MPI_Send sends the exact count of elements, and MPI_Recv will receive at most the count of elements. The fourth and fifth arguments specify the rank of the sending/receiving process and the tag of the message. The sixth argument specifies the communicator and the last argument (for MPI_Recv only) provides information about the received message.

```
MPI_Send (
  void* data,
  int count,
  MPI_Datatype datatype,
  int destination,
  int tag,
  MPI_Comm communicator
)
MPI_Recv (
  void* data,
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm communicator,
  MPI_Status* status
)
```

Code 2.16: Prototypes of MPI_Send and MPI_Recv.

2.3.3 The MPI Reduce

In some MPI parallel applications, there is need to send/receive information to/from all the processes in the communicator. The function MPI_Reduce and MPI_AllReduce are examples of functions for performing collective communications simultaneously. Recall from 2.2.2 that a reduction involves transforming a list of numbers into a shorter list of numbers or even a single number. This way, the MPI_Reduce performs a reduction operation in a single process, and the MPI_AllReduce performs the reduction in all processes at once.

This functions are very useful for the parallel dot product implementation. After finishing the local dot product, the reduction is desired for calculating the result for the global dot product operation. The example in Code 2.17 shows how the reduction is used for calculating the global result after calculating the local one.

```
  double result_loc=0;
  double result_glb=0;
  result_loc = dot_product (x, y, leng);
  MPI_Reduce(&result_loc, &result_glb, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

Code 2.17: MPI Example. Using the MPI_Reduce.

The first argument is the send-data buffer, that is a pointer to the variable which is going to be reduced. The second argument is the receive-data, that is a pointer to variable in which the reduced value is going to be stored. The third and fourth arguments describe the count and type of elements that reside in the buffers. The fifth argument specifies the reduction operation. The sixth argument specifies the root, that is the process which will perform the reduction. Finally, the last argument specifies the communicator.

2.3.4 The MPI Barrier

Similar to the OpenMP implementation, the parallelization with MPI also needs to be synchronized. The function MPI_Barrier is used in this project for synchronization purposes. Recall from Section 2.2.2 that a barrier is a programming line that forces a group of threads or processes to wait for everyone in the group reaching the barrier.

2.4 Hybrid Parallel Programming

The hybrid parallel programming refers to the practice of using two or more parallel interfaces within the same code. Specifically, the hybrid MPI+OpenMP is a parallelization paradigm commonly used in computational applications that are to be run in computer clusters or supercomputers and it has been used in this project for the third stage implementation of the operators in Section 6.4.3. The Figure 2.7 represents how OMP threads and MPI processes are distributed. In one hand, the MPI distributes one process per node. On the other hand, OpenMP saturates each node by creating as many threads as processors has the node.

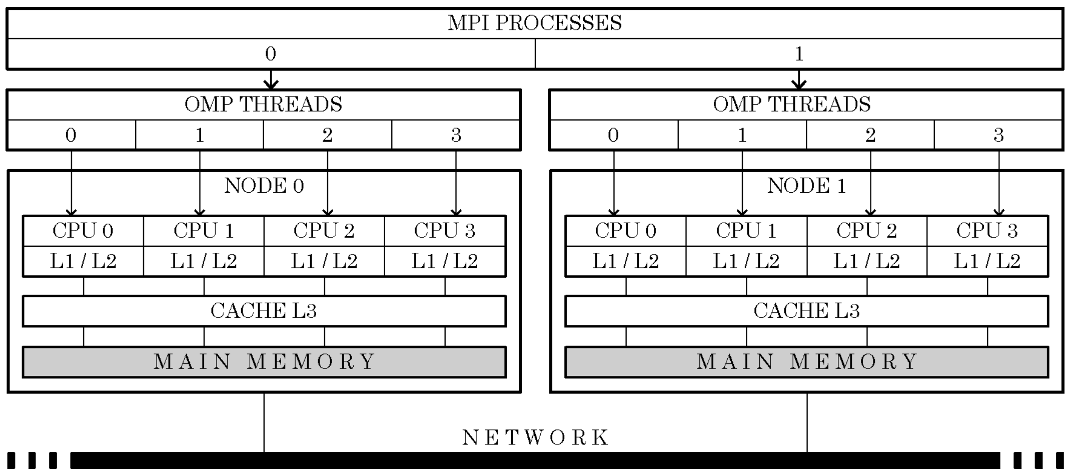


Figure 2.7: Scheme of a hybrid task distribution.

Chapter 3

Mathematical Formulation

In physics, fluid dynamics is a sub-discipline of fluid mechanics that deals with fluid flow, that is liquids and gases in motion. Fluid dynamics has a wide range of applications, including determining the turbulent eddies generated inside the combustion chamber of an engine, calculating the rate of heat dissipation of a CPU or predicting weather patterns among many others.

The solution to a fluid dynamics problem typically involves calculating various properties of the fluid, such as flow velocity, pressure, density, and temperature, as functions of space and time. Although fluids are composed of molecules that collide with one another and solid objects, they are assumed to obey the continuum assumption so, the fact that the fluid is made up of discrete molecules is ignored. Therefore continuum assumption considers fluids as continuous medium rather than discrete. Consequently, properties such as density, pressure, temperature, and flow velocity are supposed well-defined at infinitesimally small points, and are assumed to vary continuously from one point to another.

The hypotheses considered hereinafter are:

- **Continuum assumption**: The sample of fluid has a sufficiently large number of molecules and, therefore, fluid properties vary continuously in space and time.
- **Two-dimensional flow**: The properties of the fluid only vary in the plane.
- **Incompressible flow**: The flow is composed of an incompressible fluid.
- **Newtonian fluid**: The viscosity of the fluid is considered constant.
- **Boussinesq approximation**: Density differences are ignored except in terms where it is multiplied by g (acceleration due to gravity).
- **Non-participating medium**: The fluid neither emits, absorbs, nor scatters radiation and hence has no effect on radiation exchange.
- **Absence of viscous dissipation**: The viscous dissipation term in energy equation is zero.
- **Absence of volumetric energy sources**: The volumetric source terms in energy equation are zero, that is no energy is generated within the control volume.

From this point, the formulation of the governing equations for the present project can be started.

3.1 Reynolds Transport Theorem

Physical laws are stated in terms of various physical parameters (e.g. mass, velocity or temperature). Consider an extensive parameter Φ. Thus, the intensive parameter φ represent the amount of the parameter per unit mass. That is,

$$\Phi = m\varphi$$

where m is the mass of the portion of fluid of interest. This way, the amount of an extensive parameter in a determined portion of fluid can be determined as

$$\Phi_{sys} = \int_{sys} \rho\varphi d\,\text{\rlap{--}V}$$

where ρ is the density of the fluid of interest.

Sometimes it is interesting to measure the variation of the property in a system. A system is an identifiable collection of mass that moves with the fluid (indeed it is a specified portion of the fluid). Other times it is interesting to see what effect the fluid has on a particular fixed region in space called control volume. A control volume is a geometrically defined volume in space through which fluid particles may flow.

Most of the laws governing fluid motion involve the time rate of change of an extensive property of a fluid system. Thus, it is usual to encounter terms such as

$$\frac{d\Phi_{sys}}{dt} = \frac{d\left(\int_{sys} \rho\varphi d\,\text{\rlap{--}V}\right)}{dt}. \tag{3.1}$$

To formulate the laws into a control volume approach, the expression for the time rate of change of an extensive property within a control volume is required. This can be written as

$$\frac{d\Phi_{cv}}{dt} = \frac{d\left(\int_{cv} \rho\varphi d\,\text{\rlap{--}V}\right)}{dt}. \tag{3.2}$$

Although Equations 3.1 and 3.2 may look very similar, the physical interpretation of each is quite different. Mathematically, the difference is represented by the difference in the limits of integration. Recall that the control volume is a volume in space (in most cases stationary). On the other hand, the system is a portion of fluid. The Reynolds transport theorem provides the relationship between the time rate of change of an extensive property for a system and that for a control volume, that is the relationship between Equations 3.1 and 3.2 [13].

3.1.1 Derivative of the Reynolds Transport Theorem

A simple version of the Reynolds transport theorem relating system concepts to control volume concepts can be obtained easily for the flow through an arbitrary, fixed control volume shown in Figure 3.1. The control volume is considered stationary. The system considered is that fluid occupying the control volume at some initial time t. A short time later, at time $t + \delta t$, the system has moved slightly to the right and the system is no longer coincident with the control volume.

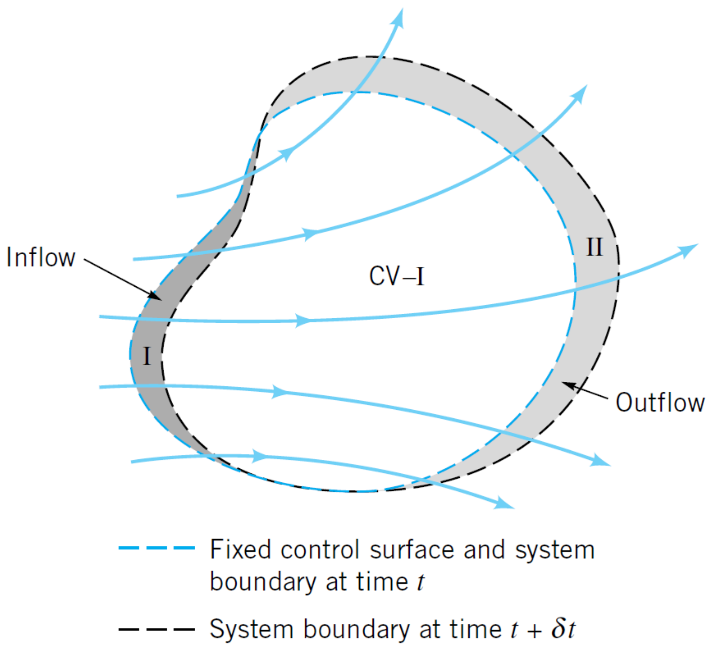

Inflow
CV–I
II
I
Outflow
Fixed control surface and system boundary at time t
System boundary at time $t + \delta t$

Figure 3.1: Control volume and system for flow through an arbitrary, fixed control volume.

In the Figure 3.1, the outflow from the control volume from time t to $t + \delta t$ is denoted as volume II, the inflow as volume II, and the control volume itself as CV. Thus, the system at time t consists of the fluid in section CV (SYS = CV at time t), while at time $t + \delta t$ the system consists of the fluid that now occupies sections (CV - I) + II.

If Φ is an extensive parameter of the system, then the value of it for the system at time

t is

$$\Phi_{sys}(t) = \Phi_{cv}(t)$$

since the system and the fluid within the control volume coincide at this time. Its value at time $t + \delta t$ is

$$\Phi_{sys}(t + \delta t) = \Phi_{cv}(t + \delta t) - \Phi_{I}(t + \delta t) + \Phi_{II}(t + \delta t).$$

Thus, the change in the amount of Φ in the system in the time interval δt divided by this time interval is given by

$$\frac{\delta \Phi_{sys}}{\delta t} = \frac{\Phi_{sys}(t + \delta t) - \Phi_{sys}(t)}{\delta t} = \frac{\Phi_{cv}(t + \delta t) - \Phi_{I}(t + \delta t) + \Phi_{II}(t + \delta t) - \Phi_{sys}(t)}{\delta t}.$$

By using the fact that at the initial time $\Phi_{sys}(t) = \Phi_{cv}(t)$, this ungainly expression may be rearranged as follows.

$$\frac{\delta \Phi_{sys}}{\delta t} = \frac{\Phi_{cv}(t + \delta t) - \Phi_{cv}(t)}{\delta t} - \frac{\Phi_{I}(t + \delta t)}{\delta t} + \frac{\Phi_{II}(t + \delta t)}{\delta t}. \tag{3.3}$$

In the limit $\delta t \to 0$, the left-hand side of Equation 3.3 is equal to the time rate of change of Φ for the system and is denoted as

$$\frac{D\Phi_{sys}}{Dt}.$$

The material derivative notation ($D\Phi_{sys}/Dt$) is used to denote this time rate of change to emphasize the Lagrangian character of this term.

In the limit $\delta t \to 0$, the first term on the right-hand side of Equation 3.3 is seen to be the time rate of change of the amount of Φ within the control volume

$$\frac{\partial \Phi_{cv}}{\partial t} = \frac{\partial \left(\int_{cv} \rho \varphi d \forall \right)}{\partial t}. \tag{3.4}$$

The third term on the right-hand side of Equation 3.3 represents the rate at which the extensive parameter Φ flows from the control volume, across the control surface II (the control surface II is the surface of CV through which the fluid leaves the control volume). Hence, in the limit $\delta t \to 0$, this term can be expressed as

$$\frac{\partial \Phi_{II}}{\partial t} = \dot{\Phi}_{out} = \int_{cs_{II}} \rho\varphi(\mathbf{V} \cdot \hat{\mathbf{n}}) dA. \tag{3.5}$$

where $\mathbf{V}$ is the velocity of the flow through the differential area element dA and $\hat{\mathbf{n}}$ is the outward pointing vector normal to the surface [13].

Similarly, the second term on the right-hand side of Equation 3.3 represents the inflow of Φ into the control volume across the control surface I (the control surface I is the surface of CV through which the fluid enters the control volume). Hence, in the limit $\delta t \to 0$, this term can be expressed as

$$\frac{\partial \Phi_{I}}{\partial t} = \dot{\Phi}_{in} = -\int_{cs_{I}} \rho\varphi(\mathbf{V} \cdot \hat{\mathbf{n}}) dA. \tag{3.6}$$

The net flux (flow-rate) of parameter Φ across the entire control surface is

$$\dot{\Phi}_{out} - \dot{\Phi}_{in} = \int_{cs_{II}} \rho\varphi(\mathbf{V} \cdot \hat{\mathbf{n}}) dA - \left(-\int_{cs_{I}} \rho\varphi(\mathbf{V} \cdot \hat{\mathbf{n}}) dA \right) = \int_{cs} \rho\varphi(\mathbf{V} \cdot \hat{\mathbf{n}}) dA, \tag{3.7}$$

where the integration is over the entire control surface.

Finally, by combining Equations 3.4 and 3.7, the time rate of change of Φ for the system is

$$\frac{D\Phi_{sys}}{Dt} = \frac{\partial}{\partial t}\left(\int_{\Omega} \rho\varphi d\Omega \right) + \int_{cs} \rho\varphi(\mathbf{V} \cdot \hat{\mathbf{n}}) dA. \tag{3.8}$$

This can be written in a slightly different form by using Ω as a generic fixed control volume, $\mathbf{u}$ as a generic velocity vector, and also introducing the derivative in the first term in the right-side (this is possible because the control volume is static), and applying the Gauss theorem to the second term in the right-hand side of the equation so that

$$\frac{D\Phi_{sys}}{Dt} = \int_{\Omega} \frac{\partial}{\partial t}(\rho\varphi) d\Omega + \int_{\Omega} \nabla \cdot (\rho\varphi\mathbf{u}) d\Omega. \tag{3.9}$$

Thus, the Reynolds transport theorem equation (Equation 3.9) represent a way to transfer from the Lagrangian viewpoint (follow a particle or follow a system) to the Eulerian viewpoint (observe the fluid at a given location in space or observe what happens in the fixed control volume). Because the system is moving and the control volume is stationary, the time rate of change of the amount of Φ within the control volume is not necessarily equal to that of the system.

3.2 Conservation Laws

The fundamental axioms of fluid dynamics are the conservation laws, specifically, conservation of mass, conservation of linear momentum (also known as Newton's second law of motion), and conservation of energy (also known as first law of thermodynamics). In addition, these laws are typically expressed using the Reynolds transport theorem.

The law of conservation for any extensive parameter, Φ, states that the total amount of the parameter within a system must remain constant in the absence of sources producing that physical property. In other words, the time rate of change in the amount of Φ in the system is equal to the time rate of generation of Φ within the system

$$\frac{D\Phi_{sys}}{Dt} = \int_{sys} \dot{Q} d\,\forall, \tag{3.10}$$

where $\dot{Q}$ represents the time rate of generation of Φ. Hence, the parameter Φ remains constant in the system if the sources are equal to zero.

The Equation 3.10, which is expressed from a Lagrangian viewpoint, can be transferred to a Eulerian viewpoint using the Reynolds transport theorem in Equation 3.9 so that

$$\int_\Omega \frac{\partial}{\partial t}(\rho\varphi) d\Omega + \int_\Omega \nabla \cdot (\rho\varphi\mathbf{u}) d\Omega = \int_\Omega \dot{Q} d\Omega, \tag{3.11}$$

The expression above is the integral conservation law that expresses the variation of Φ

within the control volume, general for any conservative and continuous parameter, Φ. As the Equation 3.11 holds for any arbitrary volume Ω, it must be valid locally at any point within control volume so that

$$\rho\left(\frac{\partial\varphi}{\partial t} + \nabla\cdot(\varphi\mathbf{u})\right) = \dot{Q}, \tag{3.12}$$

The Equation 3.12 is the general differential expression for the conservation law. Recall that the density is considered constant.

3.3 Convection-Diffusion Equation

The objective of this section is to introduce the diffusive term into the conservation law. The diffusive term arises from the source term of the conservation equation but it is usually considered as a contribution to the flow in addition to the convective term. Thus, for any conservative property Φ there is a flow that is made up of two contributions: the convective and the diffusive. However, the contribution of the diffusive term does not depend on the movement of the fluid.

The expression for the diffusive term is given by a constitutive law assuming the following experimental observations:

- the diffusive flow is proportional to the gradient of the physical magnitude,
- the gradient as directional operator needs to be negative in order to point in the direction of minimization of the function,
- the contribution must be proportional to a diffusivity factor, which is an inherent property of the physical quantity considered and its value usually relies on experimental results. In addition, its value is considered constant.

Thus, the diffusive term is mathematically defined as

$$-\rho\Gamma\nabla^2\varphi,$$

where Γ is the diffusivity factor. Inserting the expression above into the conservation law in Equation 3.12, the convection-diffusion equation results in

$$\rho\left(\frac{\partial\varphi}{\partial t} + \nabla\cdot(\varphi\mathbf{u}) - \Gamma\nabla^2\varphi\right) = \dot{Q}. \tag{3.13}$$

Recall that the convection-diffusion (Equation 3.13) is no more than the conservation equation (Equation 3.12) for which a specific contribution of the source term is considered independent from it in the form of the diffusive term. This term is very useful for formulating the Navier-Stokes equations.

3.4 Navier-Stokes Equations

The Navier-Stokes system of equations is a set of partial, differential conservation equations that describe the behavior of fluids. The system is composed of:

- conservation of mass,
- conservation of linear momentum,
- conservation of energy.

The structure for each equation is similar to the convection-diffusion expression in Equation 3.13. Further, the Navier-Stokes equations can be deduced from it by just replacing its generic parameters by the specific parameters for each conservation law listed in Table 3.1 [14] [15].

Equation	φ	Γ	$\dot{Q}$
Continuity	1	0	0
Momentum	$\mathbf{u}$	ν	$-\nabla p + \mathbf{F}$
Energy	T	α	Θ / c_P

Table 3.1: Navier-Stokes replacing parameters.

3.4.1 Continuity

The law of conservation of mass or principle of mass conservation states that for any system closed to all transfers of matter, the mass of the system must remain constant over time, as system mass can not change quantity if it is not added or removed. The law implies that mass can neither be created nor destroyed. Thus the quantity of mass is conserved over time hence there is no diffusive flow neither source term in the convection-diffusion equation. The property Φ for this case is the mass m, then $\varphi = 1$. Replacing the specific parameters for the continuity equation in Table 3.1 into the Equation 3.13, the resulting expression for continuity is

$$\nabla \cdot \mathbf{u} = 0, \tag{3.14}$$

also expressed in Cartesian coordinates as

$$\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} = 0. \tag{3.15}$$

3.4.2 Momentum

The law of conservation of linear momentum states that in a closed system (i.e. one that does not exchange any matter with its surroundings and is not acted on by external forces) the total linear momentum is constant. This fact is implied by Newton's second law of motion. The property Φ for this case is the linear momentum, $m\mathbf{u}$, then $\varphi = \mathbf{u}$. Replacing the specific parameters for the momentum equation in Table 3.1 into the

Equation 3.13, the resulting expression for momentum is

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} - \nu\nabla^2\mathbf{u}\right) = -\nabla p + \mathbf{F}, \tag{3.16}$$

where the product $\rho\nu$ is the dynamic viscosity μ, and $\mathbf{F}$ represents the sum of body forces. If considering only the gravitational body force $\mathbf{F} = \rho\mathbf{g}$, and using the Boussinesq approximation which only considers important the variation of the density if it is multiplied by the gravity, the body force term results in

$$\mathbf{F} = \rho(1 - \beta\Delta T)\mathbf{g}, \tag{3.17}$$

where β is the thermal expansion coefficient and $\mathbf{g}$ is the gravity vector. Finally, the linear momentum equation results in

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u}\right) = \mu\nabla^2\mathbf{u} - \nabla p + \rho(1 - \beta\Delta T)\mathbf{g}, \tag{3.18}$$

also expressed in Cartesian coordinates as

$$\rho\left(\frac{\partial u_x}{\partial t} + u_x\frac{\partial u_x}{\partial x} + u_y\frac{\partial u_x}{\partial y}\right) = \mu\left(\frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2}\right) + \rho(1 - \beta\Delta T)g_x - \frac{\partial p}{\partial x}, \tag{3.19}$$

$$\rho\left(\frac{\partial u_y}{\partial t} + u_x\frac{\partial u_y}{\partial x} + u_y\frac{\partial u_y}{\partial y}\right) = \mu\left(\frac{\partial^2 u_y}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2}\right) + \rho(1 - \beta\Delta T)g_y - \frac{\partial p}{\partial y}. \tag{3.20}$$

3.4.3 Energy

The law of conservation of energy states that the total energy of an isolated system remains constant, it is said to be conserved over time. Energy can neither be created nor destroyed; rather, it transforms from one form to another. The property Φ for this case is the enthalpy, H, then $\varphi = c_P T$. Replacing the specific parameters for the energy equation in Table 3.1) into the Equation 3.13,

$$\rho\left(\frac{\partial T}{\partial t} + (\mathbf{u}\cdot\nabla)T - \alpha\nabla^2 T\right) = \frac{\Theta}{c_P}, \tag{3.21}$$

where $\alpha = \lambda/\rho c_P$ is the thermal diffusivity, then $\rho\alpha = \lambda/c_P$. The coefficient λ is the thermal conduction coefficient, and the c_P is the isobaric heat capacity, both are flow properties. Finally, if the source term is neglected, the resultant expression is

$$\rho\left(\frac{\partial T}{\partial t} + (\mathbf{u}\cdot\nabla)T\right) = \frac{\lambda}{c_P}\nabla^2 T, \tag{3.22}$$

or in Cartesian coordinates

$$\rho\left(\frac{\partial T}{\partial t} + u_x\frac{\partial T}{\partial x} + u_y\frac{\partial T}{\partial y}\right) = \frac{\lambda}{c_P}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right). \tag{3.23}$$

Note that because Boussinesq hypotheses is considered, the density variations are taken into account in the buoyancy term with the objective of simulating the natural convection of the fluid. Hence, the motion of the fluid may be affected by its gradient of temperatures.

3.5 Governing Equations

In this section, the governing system of equations is exposed considering two different CFD scenarios that are: natural convection and forced convection. Their differences are due to the contribution of the buoyancy term.

3.5.1 Natural Convection

In natural convection, the buoyancy term has a significant contribution. Hence, the motion of the fluid is affected by the buoyancy. In other words, the motion of the fluid depends on its gradient of temperatures.

The governing system of equations in natural convection consists of the continuity, momentum and energy equations, that is

$$\nabla \cdot \mathbf{u} = 0, \tag{3.24}$$

$$\rho \frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u} \cdot \nabla)\mathbf{u} = \mu \nabla^2 \mathbf{u} - \nabla p + \rho(1 - \beta \Delta T)\mathbf{g}, \tag{3.25}$$

$$\rho \frac{\partial T}{\partial t} + \rho(\mathbf{u} \cdot \nabla)T = \frac{\lambda}{c_P} \nabla^2 T, \tag{3.26}$$

3.5.2 Forced Convection

Forced convection happens when the buoyancy term can be neglected due to its insignificant contribution. Hence, the motion of the fluid is not affected by the buoyancy. In this case, the flow does not depend on its gradient of temperature and the energy equation becomes useless. In addition, the flow can be assumed isotherm.

The governing system of equations in forced convection consists of only the continuity and momentum equation, that is

$$\nabla \cdot \mathbf{u} = 0, \tag{3.27}$$

$$\rho \frac{\partial \mathbf{u}}{\partial t} + \rho \nabla \cdot (\mathbf{u}\mathbf{u}) = \mu \nabla^2 \mathbf{u} - \nabla p. \tag{3.28}$$

3.6 Non-Dimensionalization

Non-dimensionalization is the partial or full removal of units from an equation involving physical quantities by a suitable substitution of variables [16, 17]. Dimensionless numbers are obtained in this process, which are very useful, both for understanding the behavior of the equations and to synthesize information.

The steps needed to non-dimensionalize a system of equations are:

- identify all the independent and dependent variables,
- replace each of them with a quantity scaled relative to a characteristic unit of measure to be determined,
- choose judiciously the definition of the characteristic unit for each variable so that the coefficients of as many terms as possible become 1,
- rewrite the system of equations in terms of their new dimensionless quantities.

The last two steps are usually specific to the problem where non-dimensionalization is applied.

3.6.1 Dimensionless Governing Equations

Both governing system of equations exposed in Section 3.5 are non-dimensionalized in this section.

Natural Convection

The parameters used in this project for non-dimensionalizing the governing system of equations in the natural convection scenario (Equations 3.24, 3.25, and 3.26) are proposed in [18] and attached in Table 3.2. Thus, the resulting dimensionless system of equations is

$$\nabla \cdot \hat{\mathbf{u}} = 0, \tag{3.29}$$

$$\frac{\partial \hat{\mathbf{u}}}{\partial \hat{t}} + (\hat{\mathbf{u}} \cdot \nabla)\hat{\mathbf{u}} = \frac{Pr}{Ra^{0.5}}\nabla^2 \hat{\mathbf{u}} - \nabla \hat{p} + \mathbf{f}, \tag{3.30}$$

$$\frac{\partial \hat{T}}{\partial \hat{t}} + (\hat{\mathbf{u}} \cdot \nabla)\hat{T} = \frac{1}{Ra^{0.5}}\nabla^2 \hat{T}. \tag{3.31}$$

where $\mathbf{f} = (\sin\gamma Pr\hat{T}, \cos\gamma Pr\hat{T})$ represents the body forces (the buoyancy) term for a control volume with a γ degrees tilt respect to the gravity's direction. The dimensionless numbers that appear applying this non-dimensionalization are:

- **Rayleigh (Ra)**: Is the ratio of the heat transfer in the form of conduction to the heat transfer in the form of convection. The Rayleigh number is defined as

$$Ra = \frac{g\beta\Delta T L_0^3}{\nu\alpha}. \tag{3.32}$$

- **Prandtl (Pr)**: Is the ratio of the momentum diffusivity to the thermal diffusivity. The Prandtl number is defined as

$$Pr = \frac{\nu}{\alpha}. \tag{3.33}$$

Variable	Units	Symbol	Reference Quantity	Dimensionless
Distance	$[m]$	x, y	L_0	$\hat{x}, \hat{y}$
Velocity	$[{}^m/_s]$	$\mathbf{u}$	$({}^\alpha/_{L_0})Ra^{0.5}$	$\hat{\mathbf{u}}$
Pressure	$[Pa]$	p	$\rho({}^{\alpha^2}/_{L_0^2})Ra$	$\hat{p}$
Temperature	$[K]$	T	$T_i - T_0$	$\hat{T}$
Time	$[s]$	t	$({}^{L_0^2}/_\alpha)Ra^{-0.5}$	$\hat{t}$

Table 3.2: Definition of the non-dimensionalization parameters.

3.6.2 Forced Convection

The parameters used in this project for non-dimensionalizing the governing system of equations in the forced convection scenario (Equations 3.27, and 3.28) are attached in Table 3.3. Thus, the resulting dimensionless system of equations is

$$\nabla \cdot \hat{\mathbf{u}} = 0, \tag{3.34}$$

$$\frac{\partial \hat{\mathbf{u}}}{\partial \hat{t}} + (\hat{\mathbf{u}} \cdot \nabla)\hat{\mathbf{u}} = \frac{1}{Re}\nabla^2\hat{\mathbf{u}} - \nabla\hat{p}, \tag{3.35}$$

where the only dimensionless number appearing is the Reynolds number:

- **Reynolds (Re)**: Is the ratio of inertial forces to viscous forces and quantifies the relative importance of these two types of forces for given flow conditions. The Reynolds number is defined as

$$Re = \frac{\rho U_0 L_0}{\mu}. \tag{3.36}$$

Variable	Units	Symbol	Reference Quantity	Dimensionless
Distance	$[m]$	x, y	L_0	$\hat{x}, \hat{y}$
Velocity	$[^m/_s]$	$\mathbf{u}$	U_0	$\hat{\mathbf{u}}$
Pressure	$[Pa]$	p	$\rho_0 U_0^2$	$\hat{p}$
Time	$[s]$	t	$^{L_0}/_{U_0}$	$\hat{t}$

Table 3.3: Definition of the non-dimensionalization parameters.

Chapter 4

Numerical Solution of the Governing Equations

The governing equations described in Chapter 3 only have analytic solution in cases where the boundary conditions are very restrictive. The spatiotemporal discretization techniques are used to get a mathematical model that can be treated by a computer in order to deal with a CFD problem that is defined in the continuous field. It is important to note that discretizing implies truncation errors that should be analyzed carefully.

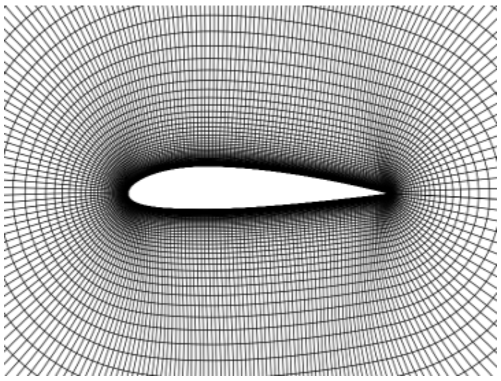
4.1 Grids

Discretization grids are defined as the discrete set of control volumes belonging to the continuous domain for which solutions are obtained. As a general classification, there are two types of grids, structured and unstructured (see Figure 4.1).

- **Structured Grid**: is a tessellation of n-dimensional Euclidean space by congruent parallelotopes (e.g. bricks). Its main feature is that each node can be accesed by the indexes (i, j) in two-dimensional problems, (i, j, k) in three-dimensional problems and so on, only depending on the dimension of the problem. This fact simplifies the programming code as well as facilitates the classification of nodes. Furthermore, the system of equations obtained by applying this technique allows to use more efficient algorithms which require less computing power.

  – **Cartesian Grid**: it is a particular type of structured grid for which two sets of lines perpendicular to each other define the whole structure. Hence, the flows are perpendicular to the all control surfaces.

- **Unstructured Grid**: is a tessellation of a part of the Euclidean plane or Euclidean space by simple shapes, such as triangles or tetrahedra, in an irregular pattern. The programming code for this type of mesh becomes more difficult and less efficient, also the algorithm requires more computing power. However, this type of grid allows a great flexibility in geometry and also to locally densify a specific part of the grid.


(a) Structured grid.

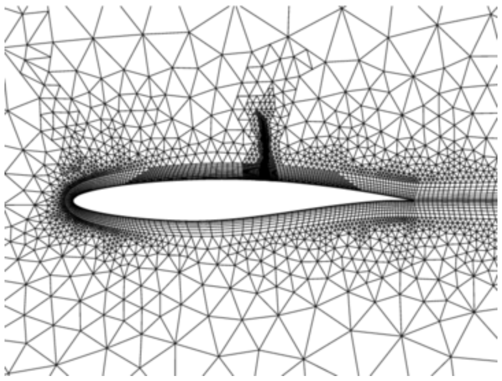

(b) Unstructured grid.

Figure 4.1: Examples of grids around an airfoil.

The parallel operators of this project are implemented for cartesian grids. In addition, the evaluation of the parameters within the grid is done in a staggered way. In the staggered treatment, the velocities of the flow are evaluated at the faces of the control volume but the rest of parameters are evaluated in the center in order to avoid an effect known as checkerboard [19]. The Figure 4.2 shows the three different types of control volumes that are involved in a two-dimensional staggered grid: one for the horizontal component of the velocity, another for the vertical component of the velocity and finall and finally one for the centered parameters such as pressure or temperature.

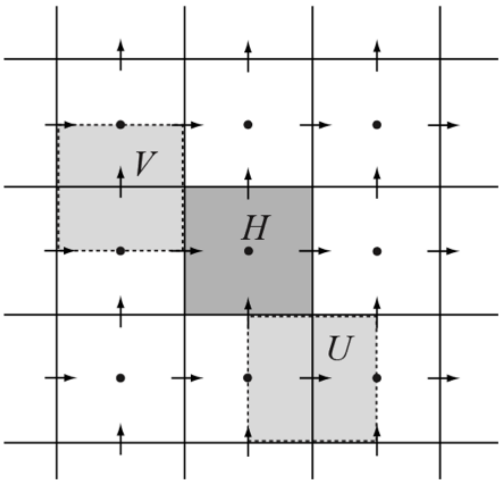


Figure 4.2: Different control volumes in a staggered grid.

4.2 Discretization

In this section, the dimensionless governing equations in the forced convection scenario are discretized (Equation 3.34 and 3.35). The discretization is finally represented in an operator-based formulation in order to design a CFD algorithm using the parallel algebraic operators programmed in this project.

Hats denoting non-dimensional variables will be dropped hereafter for simplicity.

4.2.1 Temporal Discretization

Temporal discretization consists of integrating over time an equation continuous in time. The Equation 3.34 does not have any temporary term. Hence, only the Equation 3.35 is discretized in time.

Before starting to integrate the equation, the momentum equation is re-written as follows

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{R}(\mathbf{u}) - \nabla p, \tag{4.1}$$

where

$$\mathbf{R}(\mathbf{u}) = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \frac{1}{Re}\nabla^2 \mathbf{u} \tag{4.2}$$

Integrating the Equation 4.1 over time results in

$$\int_n^{n+1} \frac{\partial \mathbf{u}}{\partial t} dt = \int_n^{n+1} \mathbf{R}(\mathbf{u}) dt - \int_n^{n+1} \nabla p dt. \tag{4.3}$$

Assuming that the temporal gradient of the velocity is constant over a specific time interval Δt, the term on the left-hand side of the Equation 4.3 is

$$\int_n^{n+1} \frac{\partial \mathbf{u}}{\partial t} dt \approx \frac{\partial \mathbf{u}}{\partial t}^{n+\frac{1}{2}} \Delta t, \tag{4.4}$$

where

$$\frac{\partial \mathbf{u}}{\partial t}^{n+\frac{1}{2}} \Delta t \approx \frac{(\mathbf{u}^{n+1} - \mathbf{u}^n)}{\Delta t} \Delta t. \tag{4.5}$$

Thus,

$$\int_n^{n+1} \frac{\partial \mathbf{u}}{\partial t} dt \approx (\mathbf{u}^{n+1} - \mathbf{u}^n). \tag{4.6}$$

In one hand, the variable n represents the instant of time in which the value of the parameters is known. On the other hand, $n + 1$ represents the parameters at a later time $t + \Delta t$ in which the parameters are unknown.

The term on the right-hand side of the Equation 4.3 is analyzed using a second order explicit method known as Adams-Bashforth scheme. Using this method, the function $\mathbf{R}(\mathbf{u})$ is evaluated at two instants of time at which the parameters are already known: n and $n - 1$. Thus, the temporal-discretized momentum equation reads

$$(\mathbf{u}^{n+1} - \mathbf{u}^n) = \Delta t \left(\frac{3}{2}\mathbf{R}(\mathbf{u}^n) - \frac{1}{2}\mathbf{R}(\mathbf{u}^{n-1}) \right) - \Delta t \nabla p^{n+1}. \tag{4.7}$$

Finally, the value for the Δt must meet the Courant-Friedrichs-Lewy (CFL) condition [20]. In mathematics, the CFL condition is a necessary condition for convergence while solving certain partial differential equations. It arises in the numerical analysis of explicit time integration schemes, when these are used for the numerical solution. As a consequence, the time step must be less than a certain time in many explicit time-marching computer simulations, otherwise the simulation will produce incorrect results. Thus, the Δt must meet the following conditions

$$\Delta t\left(\frac{|\mathbf{u}_i|}{\Delta x_i}\right)_{max} \leq C_{conv},$$

$$\Delta t\left(\frac{\nu}{\Delta x_i^2}\right)_{max} \leq C_{visc},$$

where C_{conv} and C_{visc} must be smaller than 1. It is recommended to use $C_{conv} = 0.35$ and $C_{visc} = 0.2$.

4.2.2 Solution of the Momentum Equation

The pressure-velocity coupling in the momentum equation is solved by means of a classical fractional step projection method (known as FSM) [21]. The FSM introduces a predictor velocity, $\mathbf{u}^p$, mathematically expressed as

$$\mathbf{u}^p = \mathbf{u}^n + \Delta t\left(\frac{3}{2}\mathbf{R}(\mathbf{u}^n) - \frac{1}{2}\mathbf{R}(\mathbf{u}^{n-1})\right). \tag{4.8}$$

Thus, the Equation 4.7 can be expressed as following.

$$\mathbf{u}^{n+1} = \mathbf{u}^p - \Delta t \nabla p^{n+1}. \tag{4.9}$$

The velocity field at the instant of time $n+1$ must satisfy the continuity equation (Equation

3.34). Hence, if the divergence is applied into the Equation 4.9 as following

$$\nabla \cdot \mathbf{u}^{n+1} = \nabla \cdot \mathbf{u}^{p} - \Delta t(\nabla \cdot \nabla)p^{n+1}, \tag{4.10}$$

the term on the left-hand side of the resulting equation must be equal to zero. Then,

$$\nabla^2 p^{n+1} = \frac{1}{\Delta t}\nabla \cdot \mathbf{u}^{p}. \tag{4.11}$$

The Equation 4.11 is known as Poisson equation. In order to numerically solve it, it must be spatially discretized. There are several techniques for carrying out the spatial discretization. In this project, the finite volume method is used. Thus, the values of the physical properties are calculated at discrete control volumes within a discrete grid. In the finite volume method, volume integrals of terms that contain a divergence are converted to surface integrals using the divergence theorem (also known as Gauss theorem). These terms are evaluated as fluxes at the surfaces of each finite volume. Because the flux entering a given volume is identical to that leaving the adjacent volume, this method is conservative.

The first step for numerically solving the Equation 4.11 by means of the finite volume method is to integrate over a generic dimensionless control volume Ω small enough to consider the variation of the parameters within the control volume as uniform as well as the variation of the parameters over the faces of the control volume. Then,

$$\int_{\Omega} \nabla^2 p^{n+1} d\Omega = \frac{1}{\Delta t}\int_{\Omega} (\nabla \cdot \mathbf{u}^{p}) d\Omega. \tag{4.12}$$

The divergence theorem (also known as Gauss theorem) is used to recast volumetric integrals into surface integrals as follows

$$\oint_{S} (\nabla p_f^{n+1}) \cdot \hat{\mathbf{n}} dS = \frac{1}{\Delta t}\oint_{S} (\mathbf{u}_f^{p} \cdot \hat{\mathbf{n}}) dS, \tag{4.13}$$

where the sub-index f specifies that the parameter is evaluated at the faces, S_f, of the control volume.

Recall that the control volume is small enough to consider the variations of the parameters as uniform. Thus, the expression above can be written as following

$$\sum_{f\in F(c)} (\nabla p_f^{n+1}) \cdot \hat{\mathbf{n}} S_f = \frac{1}{\Delta t} \sum_{f\in F(c)} (\mathbf{u}_f^p \cdot \hat{\mathbf{n}}) S_f, \tag{4.14}$$

where, applying the direct gradient evaluation results in

$$(\nabla p_f^{n+1}) \cdot \hat{\mathbf{n}} = \frac{\partial p_f^{n+1}}{\partial n} \approx \frac{p_{nb}^{n+1} - p^{n+1}}{\delta_{nb}}. \tag{4.15}$$

The pressure p^{n+1} is evaluated at the center of the control volume and p_{nb}^{n+1} is the pressure at the center of the neighbor node with which it shares the same face f. In addition, δ_{nb} is the distance from the center of the control volume to the center of the neighbor node nb. Thus, the Equation 4.14 reads

$$\sum_{f\in F(c)} \left(p_{nb}^{n+1} - p^{n+1}\right) \frac{S_f}{\delta_{nb}} = \frac{1}{\Delta t} \sum_{f\in F(c)} (\mathbf{u}_f^p \cdot \hat{\mathbf{n}}) S_f. \tag{4.16}$$

Notice that because of the use of the staggered mesh, the values of the velocities are already evaluated at the faces of the pressure's control volume.

Finally, the expression to obtain the new velocity field at time $n+1$ is the Equation 4.9 which, after integrating over the control volume Ω and applying the Gauss theorem, results in

$$\mathbf{u}^{n+1} = \mathbf{u}^p - \frac{\Delta t}{\Omega} \sum_{f\in F(c)} p_f^{n+1} \hat{\mathbf{n}} S_f. \tag{4.17}$$

Notice that because of the use of the staggered mesh, the values of the pressure are already evaluated at the faces of the velocity's control volume.

At this point, the Equation 4.8 must be spatially discretized in order to be able to evaluate the diffusive and the convective terms. Thus, the equation is integrated over the control volume Ω as follows

$$\int_{\Omega} \mathbf{u}^p d\Omega = \int_{\Omega} \mathbf{u}^n d\Omega + \Delta t \int_{\Omega} \left(\frac{3}{2}\mathbf{R}(\mathbf{u}^n) - \frac{1}{2}\mathbf{R}(\mathbf{u}^{n-1}) \right) d\Omega. \tag{4.18}$$

Recall that the control volume is small enough to consider the variations of the parameters as uniform. Thus, the equations above can be written as following

$$\mathbf{u}^p = \mathbf{u}^n + \frac{\Delta t}{\Omega} \int_{\Omega} \left(\frac{3}{2}\mathbf{R}(\mathbf{u}^n) - \frac{1}{2}\mathbf{R}(\mathbf{u}^{n-1}) \right) d\Omega. \tag{4.19}$$

The integral of the term $\mathbf{R}(\mathbf{u})$ is analyzed below.

$$\int_{\Omega} \mathbf{R}(\mathbf{u}) d\Omega = -\int_{\Omega} (\mathbf{u} \cdot \nabla)\mathbf{u} d\Omega + \frac{1}{Re} \int_{\Omega} \nabla^2 \mathbf{u} d\Omega. \tag{4.20}$$

Using the divergence theorem, the term on the right-hand side of the equation above result in

$$-\int_{\Omega} (\mathbf{u} \cdot \nabla)\mathbf{u} d\Omega + \frac{1}{Re} \int_{\Omega} \nabla^2 \mathbf{u} d\Omega = -\oint_S \mathbf{u}_f (\mathbf{u}_f \cdot \hat{\mathbf{n}}) dS + \frac{1}{Re} \oint_S (\nabla \mathbf{u}_f) \cdot \hat{\mathbf{n}} dS. \tag{4.21}$$

Then, since the control volume is small enough to consider the variations of the parameters as uniform, the expressions above can be written as following

$$\sum_{f \in F(c)} \mathbf{u}_f (\mathbf{u}_f \cdot \hat{\mathbf{n}}) S_f, \tag{4.22}$$

$$\frac{1}{Re} \sum_{f \in F(c)} (\nabla \mathbf{u}_f) \cdot \hat{\mathbf{n}} S_f. \tag{4.23}$$

4.2.3 Analysis of the Convective Term

The convective term is the first term in the $\mathbf{R}(\mathbf{u})$ function and is mathematically expressed as

$$\sum_{f \in F(c)} \mathbf{u}_f (\mathbf{u}_f \cdot \hat{\mathbf{n}}) S_f. \tag{4.24}$$

The product $(\mathbf{u}_f \cdot \hat{\mathbf{n}}) S_f$ can be related to the flow rate (also called discharge), $\dot{q}_f$, that is the amount of fluid flowing through a surface per unit of time. Thus,

$$\sum_{f \in F(c)} \dot{q}_f \mathbf{u}_f. \tag{4.25}$$

Notice that in this term, the velocities are evaluated at the faces of the control volume instead of being evaluated at the center. However, the velocities are only known at the center of their control volumes.

Numerical schemes are designed in order to approximate the value of the parameters from the center of the control volumes to its faces. There are many different numerical schemes which can be suitable in different CFD applications. However, only the central difference scheme (CDS) is introduced in this project as a general purpose. The CDS is mathematically defined as

$$\varphi_f = \varphi + \frac{\delta_f}{\delta_{nb}} (\varphi_{nb} - \varphi), \tag{4.26}$$

where φ is a generic parameter at the center of the control volume, φ_f is the parameter at the face f, and φ_{nb} is the parameter at the center of the neighbor node, nb, with which it shares the face f. In addition, δ_f is the distance from the center of the control volume to the face f and δ_{nb} is the distance from the center of the control volume to the center of the neighbor node nb.

4.2.4 Analysis of the Diffusive Term

The diffusive term is the second term on the $\mathbf{R}(\mathbf{u})$ function and is mathematically expressed as

$$\frac{1}{Re}\sum_{f\in F(c)}(\nabla\mathbf{u}_f)\cdot\hat{\mathbf{n}}S_f. \tag{4.27}$$

There are many different ways to calculate the diffusive term. In this case, the direct gradient evaluation has been used to solve the diffusive term as follows

$$\frac{1}{Re}\sum_{f\in F(c)}(\nabla\mathbf{u}_f)\cdot\hat{\mathbf{n}}S_f=\frac{1}{Re}\sum_{f\in F(c)}\left(\frac{\partial\mathbf{u}_f}{\partial n}\right)S_f\approx\frac{1}{Re}\sum_{f\in F(c)}\frac{(\mathbf{u}_{nb}-\mathbf{u})}{\delta_{nb}}S_f, \tag{4.28}$$

where $\mathbf{u}$ is the velocity at the center of the control volume, $\mathbf{u}_f$ is the velocity at the face f, and $\mathbf{u}_{nb}$ is the velocity at the center of the neighbor node, nb, with which it shares the face f. In addition, δ_{nb} is the distance from the center of the control volume to the center of the neighbor node nb.

4.2.5 Operator-Based Formulation

Below is summarized the algorithm which should be used in order to solve the momentum equation for each time step.

1. Evaluate the predictor velocities:

$$\mathbf{u}^p=\mathbf{u}^n+\frac{\Delta t}{\Omega}\left(\frac{3}{2}\mathbf{R}(\mathbf{u}^n)-\frac{1}{2}\mathbf{R}(\mathbf{u}^{n-1})\right),$$

where

$$\mathbf{R}(\mathbf{u})=-\frac{1}{Re}\sum_{f\in F(c)}\frac{(\mathbf{u}_{nb}-\mathbf{u})}{\delta_{nb}}S_f+\sum_{f\in F(c)}\dot{q}_f\mathbf{u}_f.$$

2. Calculate the pressure field by solving the Poisson equation:

$$\sum_{f \in F(c)} \left(p_{nb}^{n+1} - p^{n+1}\right) \frac{S_f}{\delta_{nb}} = \frac{1}{\Delta t} \sum_{f \in F(c)} (\mathbf{u}_f^p \cdot \hat{\mathbf{n}}) S_f.$$

3. Calculate the velocity field at time $n + 1$:

$$\mathbf{u}^{n+1} = \mathbf{u}^p - \frac{\Delta t}{\Omega} \sum_{f \in F(c)} p_f^{n+1} \hat{\mathbf{n}} S_f.$$

4. Evaluate the Δt based on the CFL condition:

$$\Delta t = CFL(\mathbf{u}^{n+1})$$

If the algorithm is carefully observed, it is easy to see that the parameters of each node depend only on the parameters of its neighbors. In addition, since a Cartesian grid is used, the nodes and the neighbors are easily located. Then, for a generic control volume (i, j), the neighbors $(i + 1, j)$, $(i - 1, j)$, $(i, j + 1)$, and $(i, j - 1)$ correspond to the east, west, north and south couplings respectively. In order to make benefit of this fact, the algorithm above may be expressed using algebraic operators (e.g. the *dotp*, *axpy* or *smvp*) [22].

In a matrix-vector notation, the finite-volume discretization of the Navier-Stokes equations can be written as

$$\overline{\Omega}\frac{(\mathbf{u}^{n+1} - \mathbf{u}^n)}{\Delta t} = \left(\frac{3}{2}\mathbf{R}(\mathbf{u}^n) - \frac{1}{2}\mathbf{R}(\mathbf{u}^{n-1})\right) - \overline{\Omega}\overline{G}\mathbf{p}^{n+1} \tag{4.29}$$

$$\overline{M}\mathbf{u}^{n+1} = \mathbf{0}. \tag{4.30}$$

where

$$\mathbf{R}(\mathbf{u}) = -\overline{C}\mathbf{u} + \overline{D}\mathbf{u}.$$

The vectors $\mathbf{p} = (p_1, p_2, ..., p_n) \in \mathbb{R}^n$ and $\mathbf{u} \in \mathbb{R}^{3n}$ are the pressure and velocity fields and n is the number of control volumes. The matrix $\overline{\Omega}$ is a diagonal matrix with the sizes of the control volumes on the diagonal, $\overline{C}$ and $\overline{D}$ are the convection and diffusion operators, and finally, $\overline{M}$ and $\overline{G}$ are the divergence and gradient operators respectively.

Chapter 5

Abstract Modelling of Hardware

Computer Hardware is the set of tangible elements that constitutes a computer system such as the monitor, mouse, keyboard, computer data storage, graphic cards, sound cards, memory, motherboard, and so on. In contrast, software is the set of instructions that can be stored and run by hardware.

In numerical analysis, computing systems are used as calculation tools. Then, the hardware parameters and features that have a direct impact on the calculation time and efficiency will be discussed, from an abstract point of view, throughout this chapter. Thus it can be studied as a black box capable to perform simple algebraic operations regardless of the internal operations performed by the system.

5.1 Central Processing Unit

The active part of the computer, the part that does calculations and controls all the other components, is called processor or central processing unit (CPU). The CPU contains electronic clocks that control the timing of all operations; electronic circuits that carry out arithmetic operations like addition and multiplication; circuits that identify and execute the instructions that make up a program; and circuits that fetch the data from memory. Instructions and data are stored in main memory. The CPU fetches them as needed.

The CPU in modern computers is physically implemented as a single silicon chip. This chip has engraved on it more than a million of transistors and the interconnecting wiring that define the CPU's circuits. The chip has more than a hundred of pins around its rim. Some of the pins are connection points to the bus, others are connection to electrical power supply. The bus is a communication system that transfers data between components inside a computer.

The most relevant characteristics of the CPU used for this project are listed in Table 5.1. The number of cores is higher than one if the processor is multi-core, that is a single chip with two or more independent processing units within. The clock speed is defined as the number of cycles per second measured in Hertz, that the CPU is able to perform. The CPU's memory bandwidth is the rate at which the processor is able to read/write data from/to main memory. The cache memory and the instructions are detailed in the following sections.

CHARACTERISTIC	VALUE
Manufacturer	Intel
Model	i5 4670k
Architecture	Haswell
Number of Cores	4
Clock Speed	4.00 GHz
Cache L1 Size	64 KB/core
Cache L2 Size	256 KB/core
Cache L3 Size	6 MB
Memory's Bandwidth	25.60 GB/s

Table 5.1: Characteristics of the available CPU.

5.2 Memory Hierarchy

There are many different types of memory inside a computer. The two major differences between them is their size (also called capacity) and speed (also called bandwidth). The size of the memory is the amount of binary data they are able to store, measured in Bytes (B). The speed is the rate at which they are able to write and read binary data measured in Bytes per second (B/s).

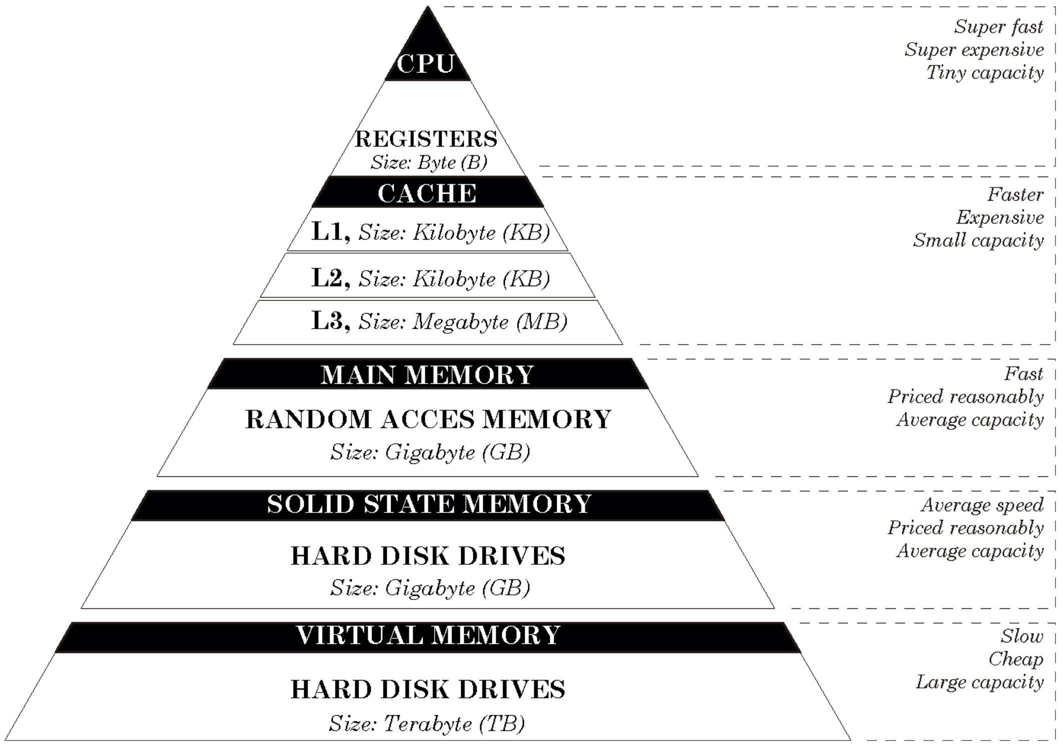


Figure 5.1: Memory hierarchy scheme.

Imagine a student who is going to solve a mathematical problem. The room where the student is going to work has a big shelf with a variety of books, pictures and other objects which he does not need to study. There is a drawer next to the table. The student stores the schoolbooks, notebooks and pens there. When the student starts working, he gets a schoolbook and a pen from the drawer and put they on the desk. Finally, the student writes the statement of the problem in a blank sheet and starts solving it, using his brain for mental calculations. Notice that, as the storing element is closer to the student, the capacity is smaller but accessing, searching and getting information is easier and faster.

Nowadays, hard disk drives (HDD) or mechanical drives are the slowest and cheapest devices in the hierarchy. They are dedicated to store data which do not require accessing very often, such as backups, movies, images, installers, etc. The next devices in the ranking are the solid state drives (SSD) and they usually store the operative system and program files in order to improve the performance of the computers. The next type of memory in the hierarchy is random access memory (RAM), also known as main memory. Main memory is used by the programs during runtime in order to store the data they need for operating. It is much faster than SSD devices. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory addresses. Further, cache memory can be subdivided into different levels, usually L1, L2 and L3. Finally, the fastest and also the most expensive types of memory are the processor's registers. The CPU uses them to store the data that it needs in order to perform instructions and operations.

When the processor needs to read from or write to an address in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. If cache were large, the CPU would waste much time checking inside it. In one hand, the major reason not to make all memories as big as possible is about data logistics: smaller memories are faster to check. On the other hand, the main reason not to make all memories as fast as possible is no other economic: the price of the memory increases proportionally to its speed.

CHARACTERISTIC	VALUE
Manufacturer	G.Skill
Model	Sniper
Memory Speed	1866 MHz (14.9 GB/s)
Channels	2
Total Bandwidth	29.80 GB/s

Table 5.2: Characteristics of the available main memory.

5.3 Throughput

In general terms, throughput is the rate of production or the rate at which something can be processed. In computational science, the throughput of a computing system can be quantified by calculating the number of floating point operations (FLOP) that the processor is able to execute per second, or also by measuring the amount of data flowing through the CPU.

Imagine you have a factory with four employees. Each employee makes a different product: the first manufactures wooden furniture, the second manufactures electronic devices, the third manufactures clothing and the fourth manufactures plastic bottles. The rate of production of each employee is one product per day, then the total throughput of the factory as system of production is four products per day. However, this value is not totally representative of the factory and should be considered cautiously to avoid confusion. If a customer requests the production of wooden furniture, the factory's pace of production for the specific order is constrained to one product per day because only one employee is able to manufacture wooden furniture.

In the same way the total throughput value can be calculated for a given computing system but this usually is not representative. Then an specific maximum throughput must be defined for every application in order to correctly evaluate its performance.

Before starting this section, the acronyms used through it are introduced:

- TPP (Theoretical Peak Performance): Is the absolute maximum throughput of a processor.
- TMP (Theoretical Maximum Performance): Is the maximum throughput of a processor when running an specific application which may not take advantage of all its features.

- TBP (Theoretical Bounded Performance): Is the maximum throughput of a processor when it is bounded by the speed of some component.
- REP (Real Performance): Is the real performance obtained when an application is run.

5.3.1 Theoretical Peak Performance, TPP

The theoretical peak performance (TPP) of a processor is usually expressed as the theoretical maximum rate of execution of 64-bit floating point operations, either addition or multiplication, per second. The acronym FLOP is used to refer the floating-point operations and the acronym FLOPS is used to refer FLOP per second (also flop/s). Then, the TPP is the result of the product of two characteristics of the CPU as expressed in Equation 5.1,

$$TPP = Clock\ Speed \cdot IPC_{ARCH}, \tag{5.1}$$

where IPC_{ARCH} is the maximum number of instructions per cycle that the processor is able to execute depending on its architecture. The clock speed is the amount of cycles per second at which the processor is able to operate measured in Hertz. Then, the theoretical peak performance of the processor described in Table 5.1 (Intel i5 4670k) is calculated as follows,

$$TPP = 4.000GHz \cdot (16 \cdot 4) = 256.000GFLOPS. \tag{5.2}$$

The IPC_{ARCH} of the Haswell architecture is 16 and it is achieved when the CPU performs two 256-bit FMA instructions per cycle (see [23]). The IPC_{ARCH} is multiplied by 4 because the processor is quad-core.

5.3.2 Theoretical Maximum Performance, TMP

Many algorithms and applications do not allow to implement the necessary instructions to obtain the IPC_{ARCH} of a given CPU's architecture (e.g. the FMA instruction cannot be implemented in an application which only performs the addition of two vectors because FMA involves both addition and multiplication). For this reason, defining the theoretical maximum performance (TMP) for specific applications or implementations that are going to run in a specific computing system is convenient for further performance evaluation. The expression for TMP is very similar to the expression for TPP. The only difference lies in the IPC value which in this case is specific for the application. Thus, the expression for the TMP is

$$TMP = Clock\ Speed \cdot IPC_{APP}. \tag{5.3}$$

5.3.3 Instruction Pipelining

Imagine a car washing with five services: 1) rinse, 2) soap, 3) scrub, 4) rinse, 5) dry. Each service is done in a different room and takes 5 minutes. If there is only one customer allowed at the same time in the whole car washing, the throughput of the car washing would be one car every 25 minutes. However, if one customer is allowed in at the same time in every different service, the throughput would become one car every 5 minutes.

Instruction pipelining is a technique for improving the IPC of the applications that implements a form of parallelism called instruction-level parallelism within a single core of the CPU. The processor needs many cycles in order to complete an instruction but rather than processing each instruction sequentially (finishing one instruction before starting the next), each instruction can be split up into a sequence of steps (similar to the car washing service) so different steps can be executed in parallel and instructions can be processed concurrently, starting the first step of a new instruction before finishing the last step of the previous instruction. The Figure 5.2 represents how an instruction that can be split up into five steps is processed. Each color represents a different instruction. In the clock cycle number 0, all instructions are waiting. In the clock cycle number 1, the first instruction begins being executed but the other four instructions still wait. In the clock cycle number 2, the first instruction jumps to the second step hence the second instruction begins being executed and so on. In the clock cycle number 5, the pipeline is completely full: all instructions are being executed on a different step.

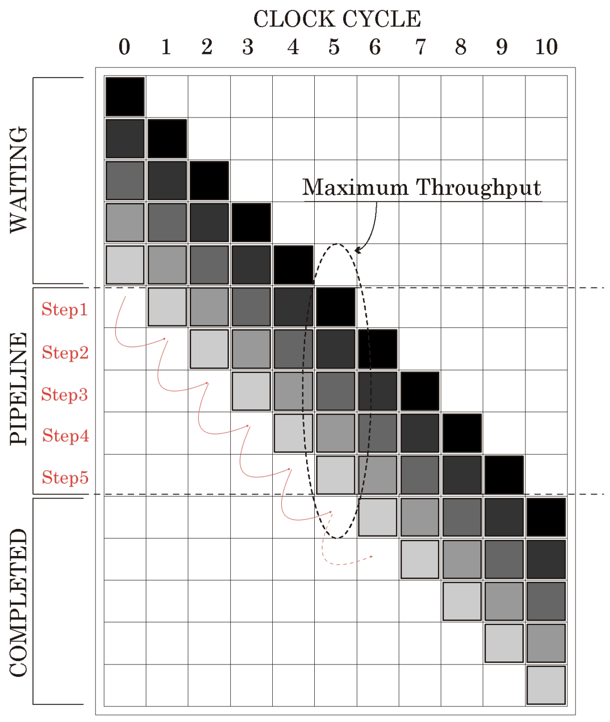


Figure 5.2: Instruction pipelining.

The most common and easy technique to help the processor to pipeline its operations is called loop unrolling. Loop unrolling is a loop transformation technique that aims to optimize a program's execution speed at the expense of its binary size. Loops are re-written as a repeated sequence of similar independent statements. The transformation can be undertaken manually by the programmer or by an optimizing compiler. The goal of loop unrolling is to increase a program's speed by increasing the IPC or reducing (or eliminating) instructions that control the loop such as pointer arithmetic and "end of loop" tests on each iteration. Loop unrolling also reduces branch penalties (improving pipelining) and hides latencies, in particular, the delay in reading data from memory [24].

The number of steps in which an operation can be split up depends on the operation itself. Thus the addition operation can be split up into 3 steps and multiplication into 5 [25]. In other words, to complete the addition takes three processor's cycles and the multiplication takes five. This way, if operations are not pipelined, then IPC_{ADD} is 1/3 and IPC_{MUL} is 1/5. In addition, at least three independent operations are required to saturate the pipeline for the addition and five for the multiplication.

Many algorithms involve a large set of independent operations thus the operations can be pipelined automatically by the compiler (e.g. in the generalized vector addition, the result of one iteration does not depend on the result of the previous one). In contrast, there are some algorithms which involve dependent operations hence its operations are difficult to pipeline and in addition it usually requires the creation of temporary variables (e.g. in the dot product, each iteration calculates the product of two numbers then the result is added to the previous result). If operations are totally pipelined, then both IPC_{ADD} and IPC_{MUL} values are 1.

Below is an exemplification of how to pipeline the operations in order to improve its IPC and thus its specific theoretical maximum performance. The Code 5.1 shows a simple loop which performs N times the addition operation on the same variable hence the operations are dependent and not pipelined.

```
double a = 1.5;
double result = 0;
for (int i=0; i<N; ++i) {
   result = result + a;
}
```

Code 5.1: Study of pipelining. One pipelined addition.

The TMP for the algorithm above is calculated as follows

$$TMP = 4.000GHz \cdot \frac{1}{3} = 1.333GFLOPS, \tag{5.4}$$

where 1/3 is the specific IPC_{ADD} for this application.

The Code 5.2 shows an unrolled loop which performs N times the addition operation distributed over two different temporary variables in order to make the operations independent. In this case two operations are pipelined thanks to the creation of two temporary variables.

```
double a = 1.5;
double result = 0;
double result_tmp[2] = {0};
for (int i=0; i<N/2; ++i) {
   result_tmp[1] = result_tmp[1] + a;
   result_tmp[2] = result_tmp[2] + a;
}
result = result_tmp[0] + result_tmp[1];
```

Code 5.2: Study of pipelining. Two pipelined additions.

The TMP of the algorithm above is calculated as follows

$$TMP = 4.000GHz \cdot \frac{2}{3} = 2.667GFLOPS, \tag{5.5}$$

where 2/3 is the specific IPC_{ADD} for the second application.

The Code 5.3 shows an unrolled loop which performs N times the addition operation distributed over three different temporary variables in order to make the operations independent. In this case three operations are pipelined, then the pipeline is already saturated.

```
double a = 1.5;
double result = 0;
double result_tmp[3] = {0};
for (int i=0; i<N/3; ++i) {
   result_tmp[0] = result_tmp[0] + a;
   result_tmp[1] = result_tmp[1] + a;
   result_tmp[2] = result_tmp[2] + a;
}
result = result_tmp[0] + result_tmp[1] + result_tmp[2];
```

Code 5.3: Study of pipelining. Three pipelined additions.

The TMP of the algorithm above is calculated as follows

$$TMP = 4.000GHz \cdot 1 = 4.000GFLOPS, \tag{5.6}$$

where 1 is the specific IPC_{ADD} for the third application.

It is important to notice that all three codes above are three different implementations of the same application but the TMP of the third application is greater than the second's as well as the TMP of the second application is greater than the first's. However, IPC_{ADD} cannot be further increased by only unrolling more the loop because the third application had already saturated the pipeline.

The reflection above can also be done for the multiplication operation. The Table 5.3 shows the theoretical maximum performance depending on the number of operations pipelined (the acronym POPS is used hereinafter to refer to pipelined operations). As 1 is the maximum value of IPC for this application, the maximum TMP is 4.000 GHz.

In order to evaluate all the concepts mentioned above, one specific benchmark application have been designed (the Code 5.4 shows the kernel of the application). This program preforms a large number of additions or multiplications, distributed over a specific number of independent variables. The application is run N times, increasing n (the number of independent variables) from 1 to N; every time calculates the elapsed time

POPS	TMP_{ADD}	TMP_{MUL}
1	1.333	0.800
2	2.667	1.600
3	4.000	2.400
4		3.200
5		4.000

Table 5.3: Study of pipelining. Theoretical maximum performance.

and the performance (the code is designed in order to allow the compiler to unroll the operating loop in order to evaluate performance vs POPS) measured in GFLOPS. Finally, the program prints the results in a text file and a gnuplot script plots the graphics with the results. Notice that the number of iterations performed, ITER, must be large, otherwise the processor finish all the work quickly without being able to quantify the elapsed time. Since the data size is very small (160 bytes at most), it fits into the processor's registers. Besides variables are read just once, 100,000,000 operations are performed hence the reading time is insignificant. This phenomena is called spatial locality.

```
int N = 10;
double result;
long int ITER = 1e8;
for (int n=1; n<=N; ++n) {
  result = 0.0;
  //creates vectors with 'n' independent elements within
  vector<double> x(n, 1.0);
  vector<double> y(n, 2.0);
  //starts the timer
  timer = STOPTIME ();
  for (long int j=0; j<(ITER/n); ++j) {
    //'n' independent additions performed every iteration
    //this loop can be unrolled by compiler
    for (int i=0; i<n; ++i) {
      x[i] = x[i] + y[i];
    }
  }
  //stops the timer and calculates elapsed time
  timer = STOPTIME () - timer;
  for (int i=0; i<n; ++i) {
      result = result + x[i];
  }
}
```

Code 5.4: Study of pipelining. The benchmark application.

The results obtained from the benchmark are shown in Table 5.4 and in the graphic in Figure 5.3, expressed in GFLOPS. It can be observed from the that the processor is performing the operations as predicted in 5.3. The results for higher POPS values are listed in order to confirm that performance is limited.

POPS	REP_{ADD}	REP_{MUL}
1	1.328	0.796
2	2.656	1.592
3	3.988	2.392
4	3.988	3.192
5	3.988	3.984
6	3.988	3.988
7	3.996	3.988
8	3.988	3.988
9	3.988	3.988
10	3.980	3.972

Table 5.4: Study of pipeline. The benchmark results.

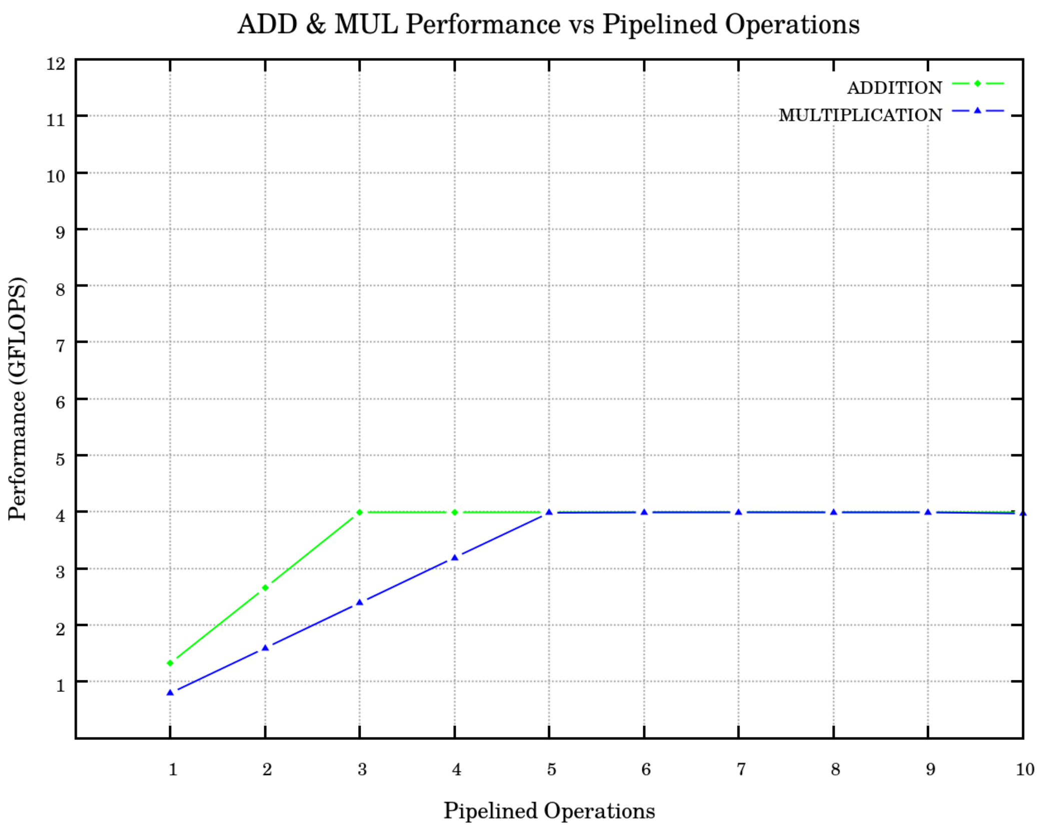


Figure 5.3: Study of pipeline. The benchmark results.

5.3.4 Fused Multiply-Add

The fused multiply-add (FMA) instructions allow the processor to perform both addition and multiplication in the same cycle. In other words, every FMA operation requires a single cycle to perform two FLOP instead of one. Then, if the algorithm requires both addition and multiplication operations to be done simultaneously, the FMA instructions should double the theoretical maximum performance of the application. The number of cycles needed to finalize the FMA operation is five so it can be split up into 5 steps. In other words, to complete the FMA instruction requires 5 processor's cycles. This way, if FMA is not pipelined, then IPC_{FMA} is 2/5. In addition, at least five independent instructions are required to saturate the pipeline for the FMA.

The Code 5.5 shows a simple loop which performs N times the FMA operation on the same variable hence the operations are dependent and not pipelined.

```
double a = 1.5;
double b = 2.0;
double c = 3.0;
double result = 0;
for (int i=0; i<N; ++i) {
  result = a + b*c;
}
```

Code 5.5: Study of FMA. One pipelined operation.

The TMP for this algorithm is calculated as follows

$$TMP = 4.000GHz \cdot \frac{2}{5} = 1.600GFLOPS, \tag{5.7}$$

where 2/5 is the specific IPC_{FMA} for this application.

Similarly to Section 5.3.3, the Code 5.5 can be unrolled in order to improve its specific IPC_{FMA}. The maximum value for IPC_{FMA} in this case is 10/5. The Table 5.5 shows the theoretical maximum performance depending on the number of POPS. As 2 is the maximum value of IPC for this application, the maximum TMP is 8.000 GHz.

POPS	TMP_{FMA}
1	1.600
2	3.200
3	4.800
4	6.400
5	8.000

Table 5.5: Study of FMA. Theoretical maximum performance.

In order to evaluate the effects of the implementation of FMA and confirm that it is actually doubling the performance of the application, one specific benchmark application have been designed (see Code 5.6). This program preforms a large number of FMA instructions, distributed over a specific number of independent variables. The application is run N times, increasing n (the number of independent variables) from 1 to N; every time calculates the elapsed time and the performance (the code is designed in order to allow the compiler to unroll the operating loop in order to evaluate performance vs POPS) measured in GFLOPS. Finally, the program prints the results in a text file and a gnuplot script plots the graphics with the results. Notice that the number of iterations performed, ITER, must be large, otherwise the processor finish all the work quickly without being able to quantify the elapsed time. Since the data size is very small (320 bytes at most), it fits into the processor's registers. Besides variables are read just once, 100,000,000 operations are performed hence the reading time is insignificant. This phenomena is called spatial locality.

```
int N = 10;
double result;
long int ITER = 1e8;
for (int n=1; n<=N; ++n) {
  result = 0.0;
  //creates vectors with 'n' independent elements within
  vector<double> x(n, 1.0);
  vector<double> y(n, 2.0);
  vector<double> z(n, 3.0);
  vector<double> r(n, 0);
  //starts the timer
  timer = STOPTIME ();
  for (long int j=0; j<(ITER/n); ++j) {
    //'n' independent FMA performed every iteration
    //this loop can be unrolled by compiler
    for (int i=0; i<n; ++i) {
      r[i] = x[i] + y[i]*z[i];
    }
  }
  //stops the timer and calculates elapsed time
  timer = STOPTIME () - timer;
  for (int i=0; i<n; ++i) {
      result = result + r[i];
  }
}
```

Code 5.6: Study of FMA. The benchmark application.

The results obtained from the benchmark are shown in Table 5.6 and in the graphic in Figure 5.4, expressed in GFLOPS. The results for higher POPS values are listed in order to confirm that performance is limited. It can be observed from the results that the FMA is doubling the performance of the simple operations as predicted in 5.5. However, it should be noted that the FMA operation is presenting a small overhead when the pipeline is saturated, performing at 7.680 instead of 8.000 GFLOPS (about 96%).

POPS	REP_{FMA}
1	1.596
2	3.192
3	4.800
4	6.360
5	7.560
6	7.680
7	7.680
8	7.680
9	7.680
10	7.680

Table 5.6: Study of FMA. The benchmark results.

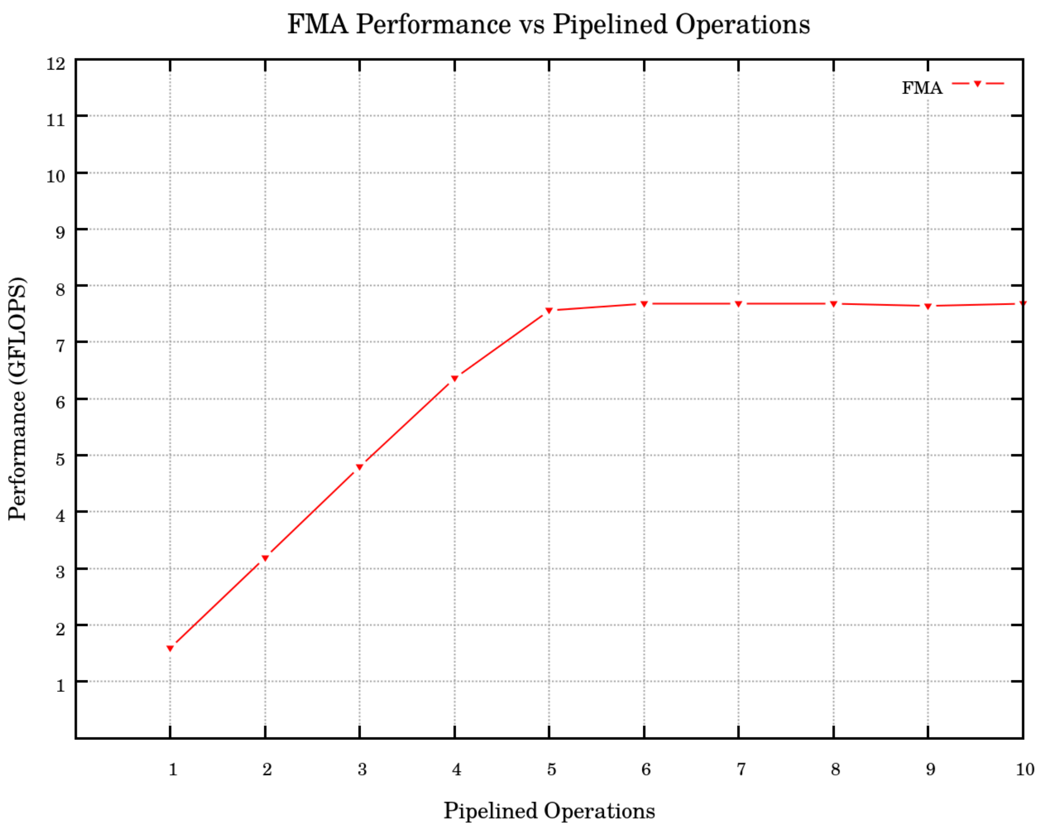


Figure 5.4: Study of FMA. The benchmark results.

5.3.5 Single Instruction Multiple Data

The SIMD (Single Instruction Multiple Data) extensions enables the CPU to perform the same instruction on multiple data using, for example, a 128-bit register in which two 64-bit numbers can be allocated. Then, every clock cycle the instructions are applied not to the single numbers but to the entire register. The Figure 5.5 represents how an instruction is performed on a 128-bit register

Nowadays, the size of SIMD registers is normally 128, 256 or even 512 bits. The streaming SIMD instructions 2 (SSE2) is a particular set of instructions that allows to perform addition and multiplication among many other instructions on 128-bit registers since 2001. These instructions can greatly increase performance when exactly the same in-

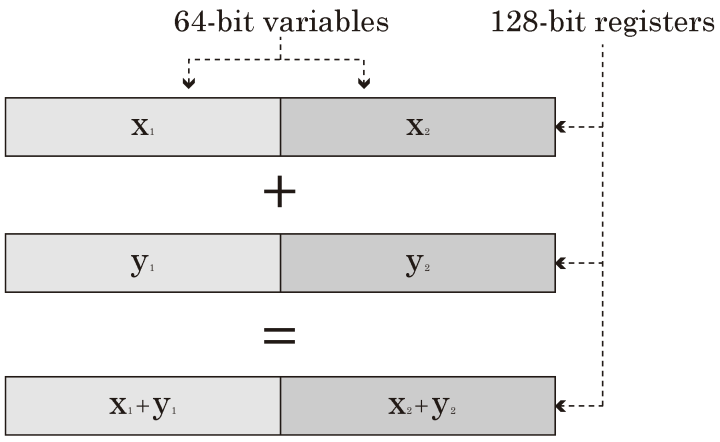


Figure 5.5: Study of SIMD. Scheme of a SSE2 operation.

structions are to be performed on multiple data objects. SSE also aims to allow the processor to manage data faster since the transfers are also done in bigger blocks. There are newer and more advanced SIMD sets such as SSE3 (2004), SSE4 (2006), AVX (2008) and so on. However, in order to make the code portable to older architectures, only SSE2 is used in this project.

The number of cycles needed to finalize the SSE2 addition is three so it can be split up into 3 steps. In a similar way, the number of cycles needed to finalize the SSE2 multiplication is five so it can be split up into 5 steps. In other words, to complete the SSE2 addition instruction requires 3 processor's cycles and the SSE2 multiplication requires 5. This way, if the instructions are not pipelined, then $\text{IPC}_{\text{SSE2ADD}}$ is 2/3 and $\text{IPC}_{\text{SSE2MUL}}$ is 2/5. In addition, at least three independent instructions are required to saturate the pipeline for the SSE2 addition and five for the SSE2 multiplication.

The Code 5.7 shows a simple loop which performs N times the SSE2 addition instruction on the same register hence the operations are dependent and not pipelined.

```
__m128d a;
__m128d b;
a _mm_set_pd (0.0, 0.0);
b _mm_set_pd (1.5, 1.5);
for (int i=0; i<N; ++i) {
  a = _mm_add_pd(a, b);
}
```

Code 5.7: Study of SIMD. One pipelined operation.

The TMP for this algorithm is calculated as follows

$$TMP = 4.000GHz \cdot \frac{2}{3} = 2.667GFLOPS, \tag{5.8}$$

where 2/3 is the specific $\text{IPC}_{\text{SSE2ADD}}$ for this application.

Similarly to Section 5.3.3, the Code 5.7 can be unrolled in order to improve its specific IPC_{SSE2}. The maximum value for both $\text{IPC}_{\text{SSE2ADD}}$ and $\text{IPC}_{\text{SSE2MUL}}$ in this case is 2. The Table 5.7 shows the theoretical maximum performance for both SSE2 addition and multiplication depending on the number of POPS. As 2 is the maximum value of IPC for this application, the maximum TMP is 8.000 GHz.

POPS	$\text{TMP}_{\text{SSE2ADD}}$	$\text{TMP}_{\text{SSE2MUL}}$
1	2.667	1.600
2	5.334	3.200
3	8.000	4.800
4		6.400
5		8.000

Table 5.7: Study of SIMD. Theoretical maximum performance.

In order to evaluate the effects of the implementation of SSE2 instructions and confirm that it is actually doubling the performance of the application, one specific benchmark application have been made (see Code 5.8) by adapting the input data of the application into 128-bit registers and using the specific SSE2 addition and multiplication instructions. The FMA has not been adapted to SSE2 extensions because it is a relatively new feature, available since 2013 with the launch of Intel Hashwell architecture and, as exposed before the code must be portable to older architectures. The program preforms a large number of SSE2 instructions, distributed over a specific number of independent registers. The application is run N times, increasing n (the number of independent registers) from 1 to N; every time calculates the elapsed time and the performance (the code is designed in order to allow the compiler to unroll the operating loop in order to evaluate performance vs POPS) measured in GFLOPS. Finally, the program prints the results in a text file and a gnuplot script plots the graphics with the results. Notice that the number of iterations performed, ITER, must be large, otherwise the processor finish all the work quickly without being able to quantify the elapsed time. Since the data size is very small (320 bytes at most), it fits into the processor's registers. Besides variables are read just once, 100,000,000 operations are performed hence the reading time is insignificant. This phenomena is called spatial locality.

```
int N = 10;
double result;
long int ITER = 1e8;
for (int n=1; n<=N; ++n) {
  result = 0.0;
  //creates vectors with 'n' independent elements within
  __m128d rx[n];
  __m128d ry[n];
  //starts the timer
  //creates vectors with '2*n' independent elements within
  vector<double> x(2*n,1.0);
  vector<double> y(2*n,2.0);
  timer = STOPTIME ();
  for (long int j=0; j<(ITER/n); ++j) {
    //'n' independent FMA performed every iteration
    //this lop can be unrolled by compiler
    for (int i=0; i<n; ++i) {
      rx[i] = _mm_loadu_pd(&x[2*i]);
      ry[i] = _mm_loadu_pd(&y[2*i]);
      rx[i] = _mm_add_pd(rx[i],ry[i]);
    }
  }
  //stops the timer and calculates elapsed time
  timer = STOPTIME () - timer;
  for (int i=0; i<n; ++i) {
    result = result + ((double*)&rx[i])[0] + ((double*)&rx[i])[1];
    }
}
```

Code 5.8: Study of SIMD. The benchmark application.

The results obtained from the benchmark are shown in Table 5.8 and in the graphic in Figure 5.6, expressed in GFLOPS. The results for higher POPS values are listed in order to confirm that performance is limited. It can be observed from the results that the behavior seems random and is different than predicted in 5.7. In one hand, the performance is not depending on the number of POPS. This might be caused by automatic optimizations made by the compiler. On the other hand, both SSE2 addition and multiplication present a small overhead when using SSE2: the maximum throughput obtained is 7.360 instead of 8.000 (about 92%).

POPS	$REP_{SSE2ADD}$	$REP_{SSE2MUL}$
1	7.320	7.080
2	5.960	6.000
3	7.080	7.080
4	7.080	7.080
5	7.240	7.240
6	7.360	7.360
7	7.000	7.000
8	7.080	7.080
9	7.200	7.160
10	4.906	4.960

Table 5.8: Study of SIMD. The benchmark results.

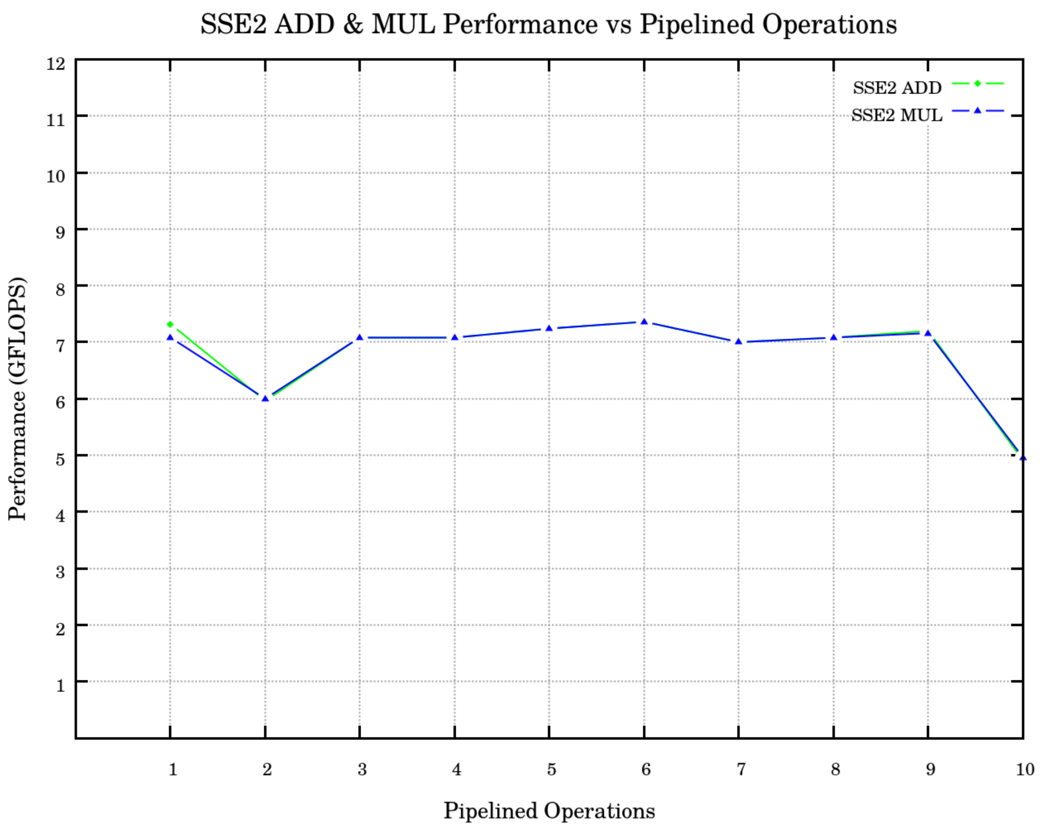


Figure 5.6: Study of SIMD. The benchmark results.

5.3.6 Theoretical Bounded Performance, TBP

The value obtained for the TPP in the Equation 5.2 is very attractive, however it is not possible to reach in most of the cases because it was not taken into account the time required for reading or writing data, which in most applications represents the bottleneck of the performance. In computational science, a bottleneck occurs when the capacity of a computational application or a computing system is severely limited by a single component. A concept that plays an important rol when analyzing performance and bottlenecks is the arithmetic intensity (AI), that is the ratio of total floating-point operations (measured in FLOP) to total data movement (measured in Bytes). The arithmetic intensity depends on the mathematical formulation of the operation. Recall that the intel i5 4670k reaches its TPP when performing eight FMA instructions per cycle. The AI of the FMA operation is

$$AI_{FMA} = \frac{2flop}{16Bytes} = 0.125\frac{flop}{Byte}. \tag{5.9}$$

The expression for calculating the theoretical bounded performance is

$$TBP = Bandwidth \cdot AI. \tag{5.10}$$

A computational application is memory bounded if the computing system's ratio of TPP to memory bandwidth is equal or higher than the algorithm's AI or, in other words if TBP < TMP. Recall from Section 5.2 that there are many different types of memories within a computing system so that there will be many different values for the TBP depending on which type memory needs to be accessed. The Figure 5.7 represents for a generic application the variation of its bounded performance vs the data size. As data size becomes bigger, larger and thus slower memories are needed. The transition from one to another value of TBP is not drastic but progressive because the smaller memories are still in use. Since most of applications in computational science are main-memory bounded, then it might be interesting to compare the TBP_{RAM} of the Intel i5 4670k when performing the same eight FMA operations as in Equation 5.2. For this purpose, the TBP_{RAM} is calculated as follows,

$$TBP_{RAM} = 25.6GB/s \cdot \frac{8flop}{128Byte} = 3.2GFLOPS. \tag{5.11}$$

It is very disappointing that the value obtained is about 1% of the processor's TPP because most of the applications in computational science are main-memory bounded because involve operations and algorithms with very low arithmetic intensity and big data sizes (e.g. the AI of the dot product is 0.125 and the AI of the generalized vector addition is 0.083).

In order to demonstrate the existence of the bottleneck described above, the elapsed time needed for calculating the dot product of two arrays of 1,000,000 elements each is

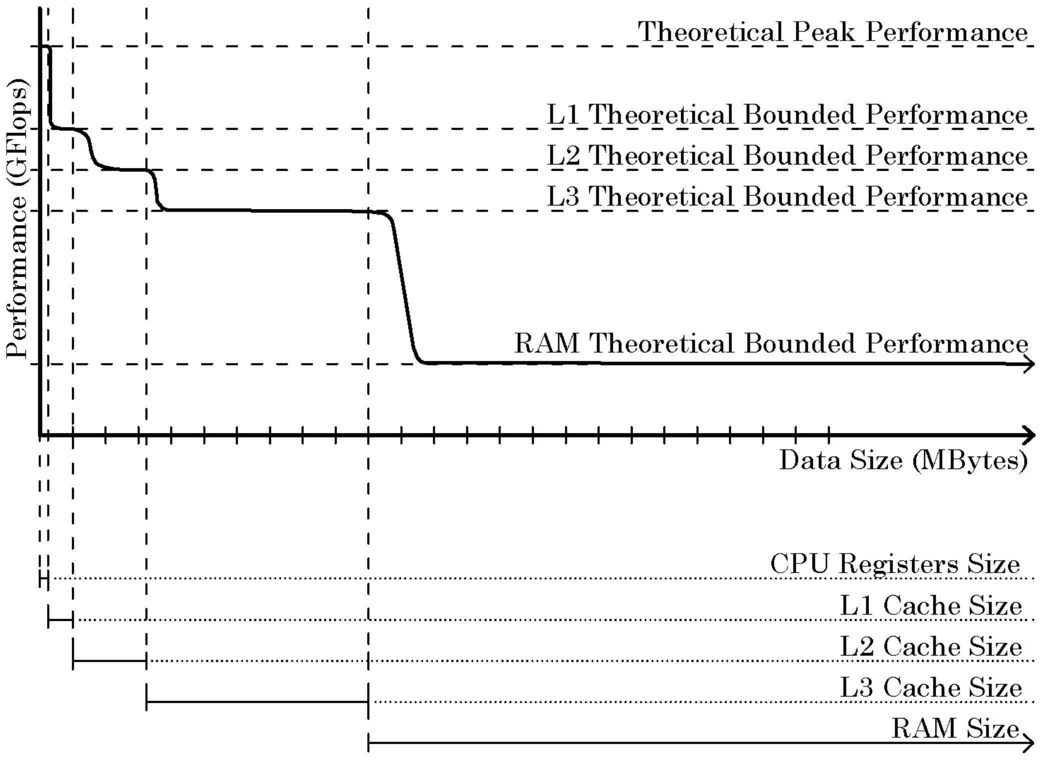

Theoretical Peak Performance
L1 Theoretical Bounded Performance
L2 Theoretical Bounded Performance
L3 Theoretical Bounded Performance
RAM Theoretical Bounded Performance
Performance (GFlops)
Data Size (MBytes)
CPU Registers Size
L1 Cache Size
L2 Cache Size
L3 Cache Size
RAM Size

Figure 5.7: Application's Performance vs Data Size.

going to be both theoretically and experimentally calculated. The data size of two arrays of 1,000,000 elements is calculated as follows,

$$SIZE_{data} = 2 \cdot 8Bytes \cdot 1,000,000 = 16,000,000Bytes \approx 16MBytes. \tag{5.12}$$

16 MB is greater than the capacity of the cache hence the performance is supposed to be main-memory bounded. Then, the TBP$_{\text{RAM}}$ for the dot product can be taken from Equation 5.11 since the AI for both applications is the same. This way, the minimum elapsed time required for performing the 2,000,000 FLOP of the dot product is calculated in Equation 5.13 as follows,

$$t = \frac{2,000,000flop}{3.2GFLOPS} = 0.000625s. \tag{5.13}$$

Finally, a simple C++ application is designed for experimentally timing the execution of the dot product. This application performs 1,000 times the dot product of two arrays of

1,000,000 elements each, then measures the average elapsed time and calculates the real performance or REP and finally prints the result in the *shell* comparing it to the TBP (see Code 5.9).

```
$ TBP: 3.200 GFLOPS, 0.000625 seconds
$ REP: 2.546 GFLOPS, 0.000786 seconds
$ DEVIATION: 25.699%
```

Code 5.9: TBP vs REP for 1,000,000 elements dot product.

The obtained result was even worse than the result predicted in Equation 5.13. The deviation in the result might be caused by the fact that a single core of a multi-core CPU is not able to obtain full advantage of the memory bandwidth. This issue is analyzed more deeply in the Chapter 6.

5.4 Conclusions

Before finishing this chapter, some conclusions obtained from the study above are exposed.

- The AI (arithmetic intensity) is an important parameter for analyzing whether an application is memory bounded or computing bounded.
- The IPC (instructions per cycle) is an important parameter for calculating the TPP (theoretical peak performance) and TMP (theoretical maximum performance).
- It is a must to pipeline the operations in order to optimize the IPC.
- Using FMA doubles the IPC of the application and also its TMP.
- Using SIMD instructions increases the IPC and the TMP of an application. It also allows the compiler to optimize the pipelining of the code.

- The AI of an algorithm depends on the algorithm's mathematical formulation and not on the application.
- Most of the applications in computational science will be main-memory bounded because involve operations and algorithms with very low arithmetic intensity and big data sizes.
- In a main-memory bounded application, the most significant optimization is to minimize as much as possible the data movements. In addition, to design efficient data structures is a must.
- In a main-memory bounded application, optimizing the IPC is not worth it: the REP does not depend on the IPC but on the AI. However, some optimizations may help the processor to read or write data in a more efficient way.
- When a computational application starts running, the data is created and stored into the main memory. If the program iterates many times over the same set of variables, the performance will be bounded by the speed of the memory in which the data fits. If the data size is small (the specific size depends on the processor) it may fit into the processor's caches or even in its registers. Then, the variables are read just once from main memory hence the spatial locality phenomena happens: reading or writing time becomes insignificant.

In conclusion, the major bottleneck in computational science is the speed of reading from and writing to main memory. For this reason, the IPC of the algorithms must be as high as possible and the computational applications must be optimized by minimizing the data movement. The objective is to obtain a REP similar to the TBP, which means that the bottleneck is saturated.

Chapter 6

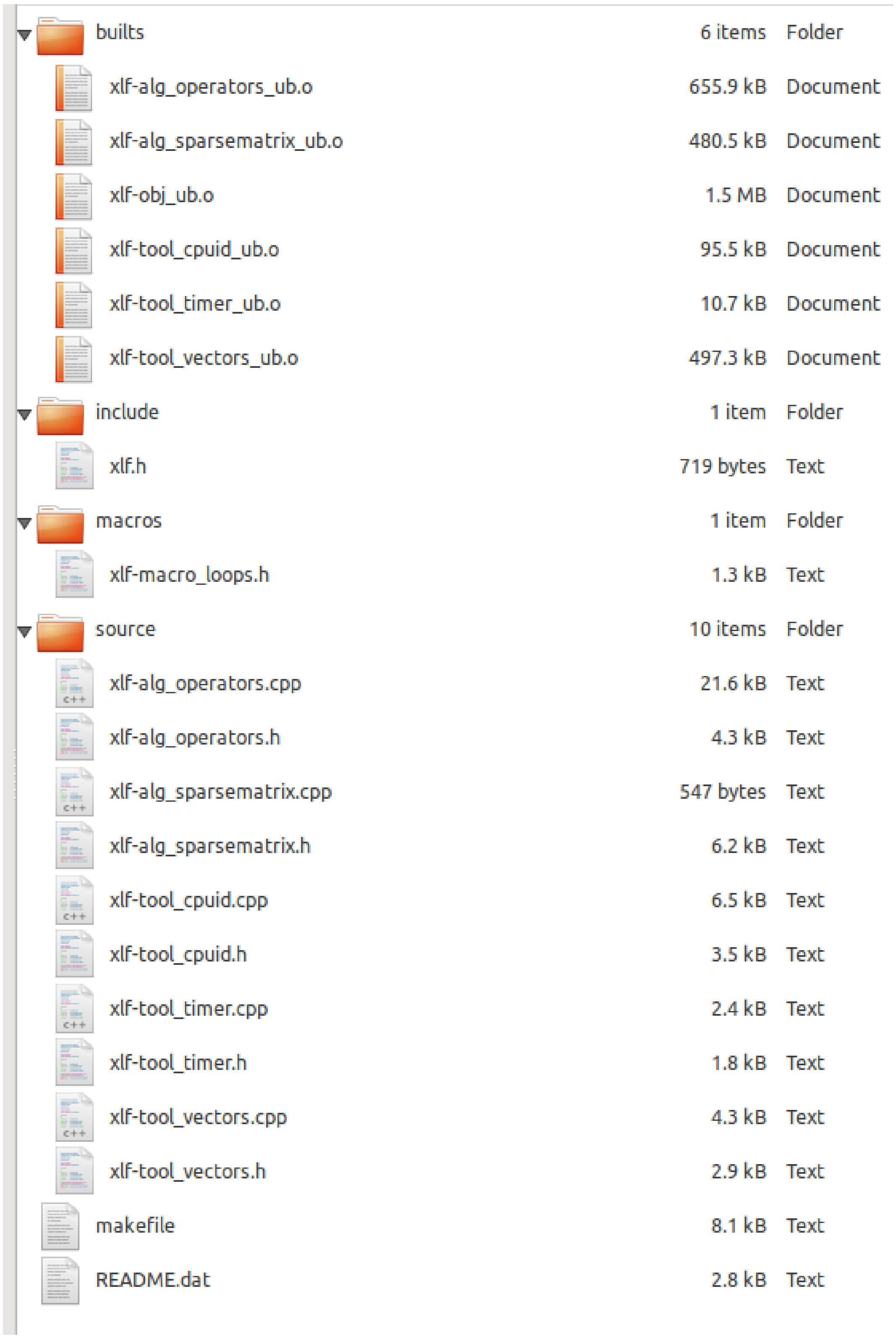
XLF, the Linear Algebra Library

In this chapter, the development process of the algebraic operators of the XLF library is detailed (XLF is how the library designed for this project is named). First of all, the mathematical formulation is introduced. Then, the computational cost of the operators is analyzed. Finally, the performance evaluation is carried for each implementation of the operators.

6.1 The XLF File Structure

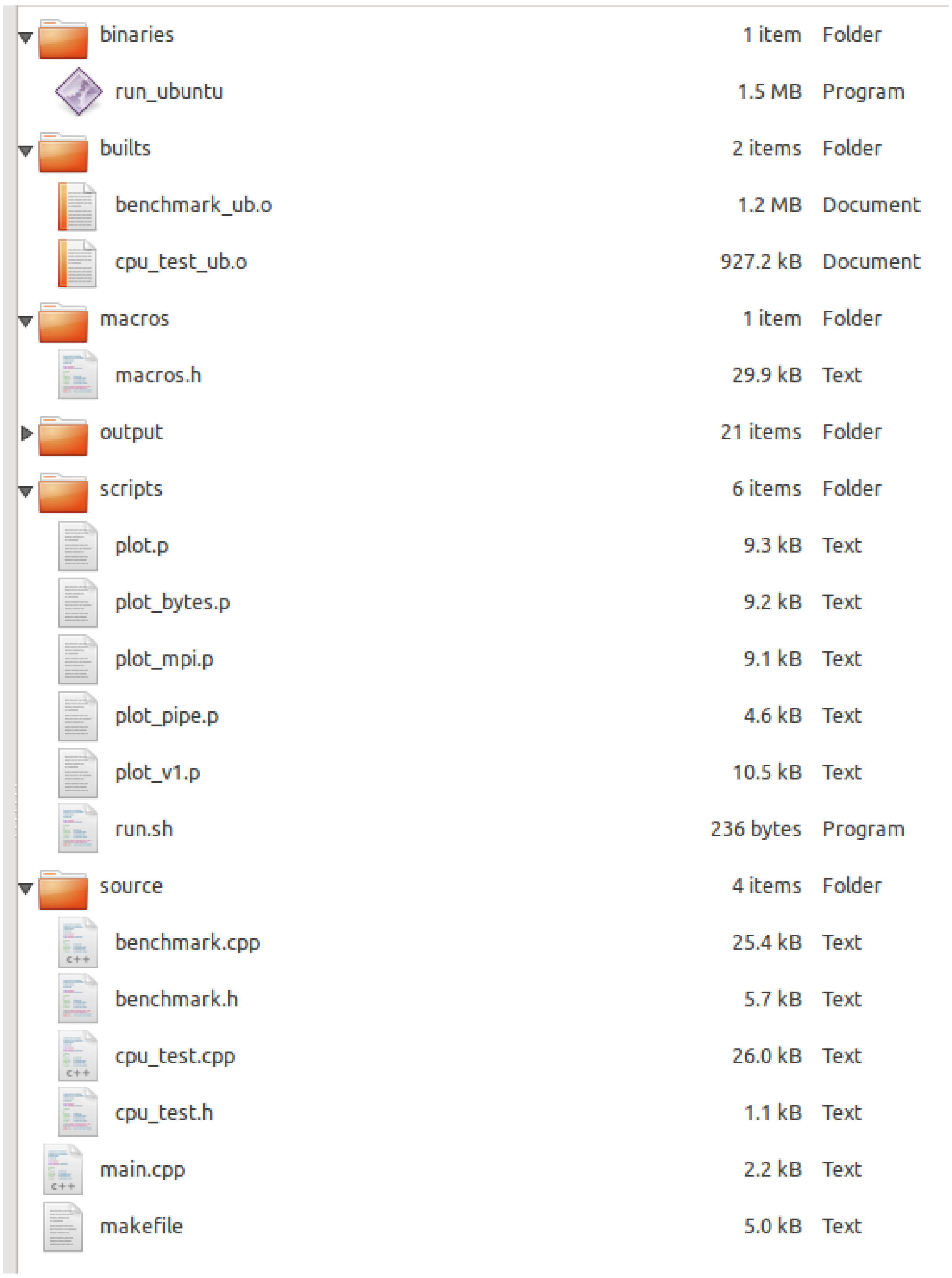
The total number of C++ lines written in this project is larger than 5,000. For this reason, the code has been divided into many different files. The Figure 6.1 shows the file structure of the XLF library, and the Figure 6.2 shows the file structure of the XLF benchmark.

In order to ensure portability to the different computers that have been used, the makefile includes three different settings: Ubuntu + Intel, RedHat + AMD, SuSe11 + Intel.



```
▼ builts                              6 items    Folder
    xlf-alg_operators_ub.o            655.9 kB   Document
    xlf-alg_sparsematrix_ub.o         480.5 kB   Document
    xlf-obj_ub.o                      1.5 MB     Document
    xlf-tool_cpuid_ub.o               95.5 kB    Document
    xlf-tool_timer_ub.o               10.7 kB    Document
    xlf-tool_vectors_ub.o             497.3 kB   Document
▼ include                             1 item     Folder
    xlf.h                             719 bytes  Text
▼ macros                              1 item     Folder
    xlf-macro_loops.h                 1.3 kB     Text
▼ source                              10 items   Folder
    xlf-alg_operators.cpp             21.6 kB    Text
    xlf-alg_operators.h               4.3 kB     Text
    xlf-alg_sparsematrix.cpp          547 bytes  Text
    xlf-alg_sparsematrix.h            6.2 kB     Text
    xlf-tool_cpuid.cpp                6.5 kB     Text
    xlf-tool_cpuid.h                  3.5 kB     Text
    xlf-tool_timer.cpp                2.4 kB     Text
    xlf-tool_timer.h                  1.8 kB     Text
    xlf-tool_vectors.cpp              4.3 kB     Text
    xlf-tool_vectors.h                2.9 kB     Text
  makefile                            8.1 kB     Text
  README.dat                          2.8 kB     Text
```

Figure 6.1: File structure of the XLF library.



```
▼ binaries                     1 item     Folder
    run_ubuntu                 1.5 MB     Program
▼ builts                       2 items    Folder
    benchmark_ub.o             1.2 MB     Document
    cpu_test_ub.o              927.2 kB   Document
▼ macros                       1 item     Folder
    macros.h                   29.9 kB    Text
▶ output                       21 items   Folder
▼ scripts                      6 items    Folder
    plot.p                     9.3 kB     Text
    plot_bytes.p               9.2 kB     Text
    plot_mpi.p                 9.1 kB     Text
    plot_pipe.p                4.6 kB     Text
    plot_v1.p                  10.5 kB    Text
    run.sh                     236 bytes  Program
▼ source                       4 items    Folder
    benchmark.cpp              25.4 kB    Text
    benchmark.h                5.7 kB     Text
    cpu_test.cpp               26.0 kB    Text
    cpu_test.h                 1.1 kB     Text
  main.cpp                     2.2 kB     Text
  makefile                     5.0 kB     Text
```

Figure 6.2: File structure of the XLF benchmark.

6.2 Implementation of Operators

The objective of this section is to traduce the mathematical form of the operators into C++ language so that the computer is able to perform the calculations. For a better understanding of the sparse matrix-vector product, see Section 2.1 in which sparse matrices and its different types of structures are analyzed.

6.2.1 Dot Product

The dot product or scalar product is an algebraic operation that takes two equal-length sequences of numbers and returns a single number, which is the sum of the products of the corresponding entries of the two sequences of vectors. The name dot product is derived from the centered dot (·) that is often used to designate this operation; the alternative name, scalar product, emphasizes that the result is a scalar instead of a vector. The name chosen for programming this operator in the project's library is *dotp*. The dot product of the vectors

$$\mathbf{x} = (x_1, x_2, \cdots, x_n)$$

and

$$\mathbf{y} = (y_1, y_2, \cdots, y_n)$$

is defined as

$$\mathbf{x} \cdot \mathbf{y} = x_1 \cdot y_1 + x_2 \cdot y_2 + \cdots + x_n \cdot y_n \tag{6.1}$$

or

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i \cdot y_i. \tag{6.2}$$

The Code 6.1 shows the most basic implementation of the dot product in C++ language.

```
int N;
int sum = 0;

vector<double> x(N, 1.0);
vector<double> y(N, 2.0);

for (int i=0; i<N; ++i) {
   sum = sum + x[i]*y[i];
}
```

Code 6.1: Algorithm DOTP (Simple).

6.2.2 Generalized Vector Addition

The generalized vector addition or linear combination is an algebraic operation that takes two equal-length sequences of numbers and returns another equal-length sequence of numbers. The returned sequence is the result of multiplying all the numbers of one of the sequences by a scalar then adding the corresponding element of the other sequence. The name chosen for programming this operator in the project library is *axpy*, and it refers to the form of the operation $a \cdot \mathbf{x} + \mathbf{y}$. Then, considering the following set of vectors

$$(\mathbf{e_i})_{i \in n} = ((\delta_{ij})_{j \in n})_{i \in n},$$

$$\mathbf{x} = (x_1, x_2, \cdots, x_n),$$

$$\mathbf{y} = (y_1, y_2, \cdots, y_n)$$

and the real scalar

$$a \in \mathbb{R},$$

the generalized vector addition is defined as

$$a \cdot \mathbf{x} + \mathbf{y} = (a \cdot x_1 + y_1) \cdot \mathbf{e_1} + (a \cdot x_2 + y_2) \cdot \mathbf{e_2} + \cdots + (a \cdot x_n + y_n) \cdot \mathbf{e_n} \qquad (6.3)$$

or

$$a \cdot \mathbf{x} + \mathbf{y} = \sum_{i=1}^{n} (a \cdot x_i + y_i)\mathbf{e_i}. \tag{6.4}$$

The Code 6.2 shows the most basic implementation of the dot product in C++ language.

```
int N;
double a = 3.0;

vector<double> x(N, 1.0);
vector<double> y(N, 2.0);

for (int i=0; i<N; ++i) {
   y[i] = y[i] + a*x[i];
}
```

Code 6.2: Algorithm AXPY (Simple).

6.2.3 Sparse Matrix-Vector Product

The matrix-vector product is an algebraic operation that takes a vector and a matrix; the matrix must have the same number of columns as elements has the vector. The matrix-vector product returns a vector with the same number of elements as rows has the matrix. Each element of the resulting vector is the dot product of every matrix row and the vector. The name chosen for programming this operator in the project library is *smvp*. Then, considering the following matrix

$$\mathbf{A} = \begin{pmatrix} a & b & 0 \\ c & d & e \\ 0 & f & g \end{pmatrix}$$

and the vector

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix},$$

the matrix-vector product is defined as

$$\mathbf{A} \cdot \mathbf{x} = \begin{pmatrix} a \cdot x_1 + b \cdot x_2 \\ c \cdot x_1 + d \cdot x_1 + e \cdot x_2 \\ f \cdot x_1 + g \cdot x_2 \end{pmatrix} \tag{6.5}$$

As exposed in Section 2.1, the coefficient matrix might be very sparse in many engineering problems. The sparse matrix-vector product takes advantage of the matrix sparsity by only calculating the non-zero elements products. The Code 6.3 shows the most basic implementation of the dot product in C++ language using the *sparse_xlf_2d* sparse data structure.

```
int m;       //number of horizontal nodes
int n;       //number of vertical nodes
int N = m*n; //total number of elements in main diagonal
int s, w, p, e, n; //integers for pointer arithmetic

//initialization of the sparse matrix object for an 'm*n' grid
sparse_xlf_2d *a;
a = new sparse_xlf_2d(m,n);

/* vectors are oversized by 2*m in order generalize the
   algorithm. if not, some elements may not exist the (
   e.g south (0-m) or west (0-1) elements)
   the total domain of the vector is (0 to N+2*m)
   the real domain of the vector is (m to m+N)
*/
vector<double> x(m*n+2*m, 1.0);
vector<double> y(m*n+2*m);

for (int i=0; i<N; ++i) {
   s = m+i-m;
   w = m+i-1;
```

```
    p = m+i;
    e = m+i+1;
    n = m+i+m;
    y[p] = a->as[i]*x[s] + a->aw[i]*x[w] + a->ap[i]*x[p] + a->ae[i]*x[e] +
         a->an[i]*x[n];
}
```

Code 6.3: Algorithm SMVP (Simple).

6.3 Operator's Computational Cost

In this section is analyzed the computational cost of the different operators. In order to further evaluate the performance of the code, it is necessary to find how many operations and data transfers are needed for running each operator.

In some algebraic operators there is data reuse (i.e some components of the vector or entries of the matrix that are used for more than one floating point operation e.g. in a dense matrix-vector product, each component of the vector is multiplied n_{row} times). In the sparse matrix-vector product defined in Section 6.2.3 each vector component is multiplied five times, corresponding to the central, east, west, north and south couplings that involve each unknown (with some differences at the boundaries). In computational science, this situation is known as spatial locality and to take advantage of it the data being reused is kept, if possible, on the cache memory to minimize data transfer costs. This is known as cache reuse.

The cache memory is managed by the processor and the achievement of the maximum cache reuse will depend on the size of the problem and on the size of the cache memory, as well as on the data structures used to implement the algorithm. For instance, in the *smvp* kernel defined in Section 6.2.3, every component of the vector is used during three consecutive iterations as east, central and west unknown respectively as shown in Figure 6.3. In this case, it is easily kept on the cache by the processor. However, the same component may have been used n_{row} iterations before and after, to account for the south and north couplings. In this cases, for large enough values of n_{row} the reuse is not possible since the cache memory is filled with data required on the intermediate iterations. Thus, the theoretical necessary data in order to operate the *smvp* is 7N: 5N for the five diagonals plus 1N for the input vector plus 1N for the output vector. This value could be incremented up to 11N in the absence of cache reuse.

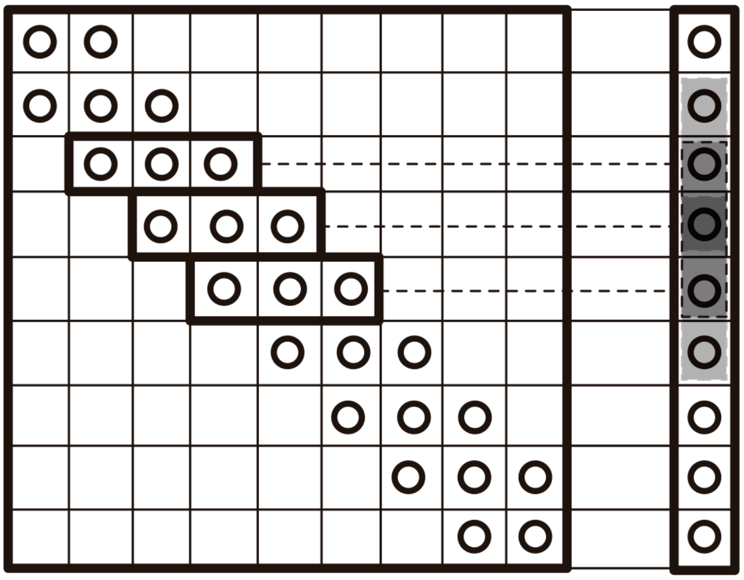


Figure 6.3: Cache reuse in sparse matrix-vector product.

The values for different parameters relative to the computational cost of the operators are listed in Table 6.1. All parameters in this table are representative of the operator's mathematical definition and do not depend on the implementation.

	DOTP	AXPY	SMVP
FLOP	$2N$	$2N$	$9N$
Bytes	$8 \cdot 2N$	$8 \cdot 3N$	$8 \cdot 7N$
Arithmetic Intensity	0.125	0.083	0.161
TBP_{RAM}	3.200	2.125	4.122

Table 6.1: Computational cost of the operators.

6.4 Developement and Performance Evaluation

In this part of the chapter, the evaluation of performance is carried out using a benchmark specially designed for this task. Some aspects are considered during the evaluation process.

- It is not possible to get a 100% efficiency. There are always some loses while performing operations: latencies, pointer arithmetic or interferences with system background.
- Many cores are needed in order to saturate the bandwidth in a multi-core processor.
- Two timings of the same function in the same conditions can be slightly different due to random computer behaviors or other interferences. For this reason, the operators are run 1,000 times and the results are the average.
- For big enough problem sizes, the *smvp* implementation of this project is able to reuse only 2N elements of the vector instead of 4N. Because of this fact, the obtained REP is not able to be similar to the TBP_{RAM}.

The tasks carried out by the benchmark are:

1. Creating the necessary matrix and vector objects for a N vector size.
2. Initializing the objects with a previously known set of numbers. The reason not to use random elements for the objects is no other than knowing the result of the operators and thus being able to detect possible application's errors.
3. Running the operators 1,000 times saving the elapsed time for each execution.
4. Calculating the average time and also the average amount of operations per second, measured in GFLOPS.

5. If desired, incrementing the N value and start over all the process.
6. Printing all the results into a .dat file.
7. Executing a gnuplot script to create the graphics of the results.

The specifications of the system where the *Stage 01* and *Stage 02* of the benchmark is run are detailed in Section 5.1 and Section 5.2.

The *Stage 03* of the benchmark aims to evaluate the performance for distributed memory systems. For this reason, it has been run in the MareNostrum III supercomputer. Although MareNostrum III has a total of 48,896 Intel SandyBridge-EP E5-2670 cores at 2.600 GHz and 51.200 GB/s of main-memory bandwidth (3,056 compute nodes with two processors per node), only from 1 to 8 nodes (16 to 128 cores) are used in *Stage 03* for scalability evaluations due to queue limitations. See [26] for further information about MareNostrum III.

6.4.1 Stage 01: Sequential Optimization

The objective of this section is to optimize the sequential operators in order to use them later in the multi-threading implementation. If there is a set of optimized sequential operators available, the parallelization task becomes very simple: every thread within the parallel region must only run the optimized sequential operators.

Sequential optimization aims to make the code perform as fast as possible using a single core. For this purpose, many different implementations of the three basic operators are included in the library and evaluated by the benchmark. The Table 6.2 shows, for each implementation, its optimizing features as well as its specific IPC, TMP, AI and TBP_{RAM} (see Section 5.3 for further information about those terms); Recall that the last column refers to the theoretical bounded performance for large problem sizes which need to be stored in the main memory (RAM). The corresponding sequential operators of the CBLAS library are evaluated in order to compare its results to the XLF performance. However, since the sparse matrix-vector product is highly dependent on the sparse data structure used and the *xlf_2d_sparse* is specifically customized for this project, it has not been compared with any reference.

Version	Features	IPC	TMP	AI	TBP$_{\text{RAM}}$
DOTP					
DOTP-01-A	FMA	2/5	1.600	0.125	3.200
DOTP-01-B	FMA + Unrolling	10/5	8.000	0.125	3.200
DOTP-01-C	SSE2	2/5	1.600	0.125	3.200
DOTP-01-D	SSE2 + Unrolling	10/5	8.000	0.125	3.200
AXPY					
AXPY-01-A	FMA	10/5	8.000	0.083	2.125
AXPY-01-B	FMA + Unrolling	10/5	8.000	0.083	2.125
AXPY-01-C	SSE2	10/5	8.000	0.083	2.125
AXPY-01-D	SSE2 + Unrolling	10/5	8.000	0.083	2.125
SMVP					
SMVP-01-A	FMA	10/5	8.000	0.161	4.122
SMVP-01-B	FMA + Unrolling	10/5	8.000	0.161	4.122

Table 6.2: Stage 01. Theoretical parameters of the operators.

The results within gray colored cells represent the application's theoretical limit in performance for large data sizes. Notice that SSE2 instructions are not used in the *smvp* implementations because of the way how SIMD registers organize the data which makes it not possible to manage for the cache reuse. Hence, the data efficiency would be reduced in order to increase the IPC. As exposed in the previous chapter's conclusions, it is more important to minimize data movement rather than increase IPC.

Results of DOTP

The results obtained from the benchmark by running the sequential *dotp* operators are listed in the Table 6.3 and plotted in Figure 6.4. The second column of the table refers to the maximum real performance (REP) obtained for each operator and the third column refers to the average performance obtained when the data is larger than 32MB, that is when it is supposed to be main-memory bounded (RAM bounded). Recall that most of the applications in computational science are main-memory bounded. Hence, the results of interest for this project are these obtained from large data sizes (32MB are considered large enough for the i5 4670k since its cache size is 6MB).

Version	REP_{MAX}	REP_{RAM}	REP_{RAM}/TBP_{RAM}
CBLAS	2.900	1.950	0.609
DOTP-01-A	1.680	1.520	0.475
DOTP-01-B	5.577	2.300	0.719
DOTP-01-C	6.972	2.354	0.736
DOTP-01-D	10.290	2.381	0.744

Table 6.3: Stage 01. Results of the dot product.

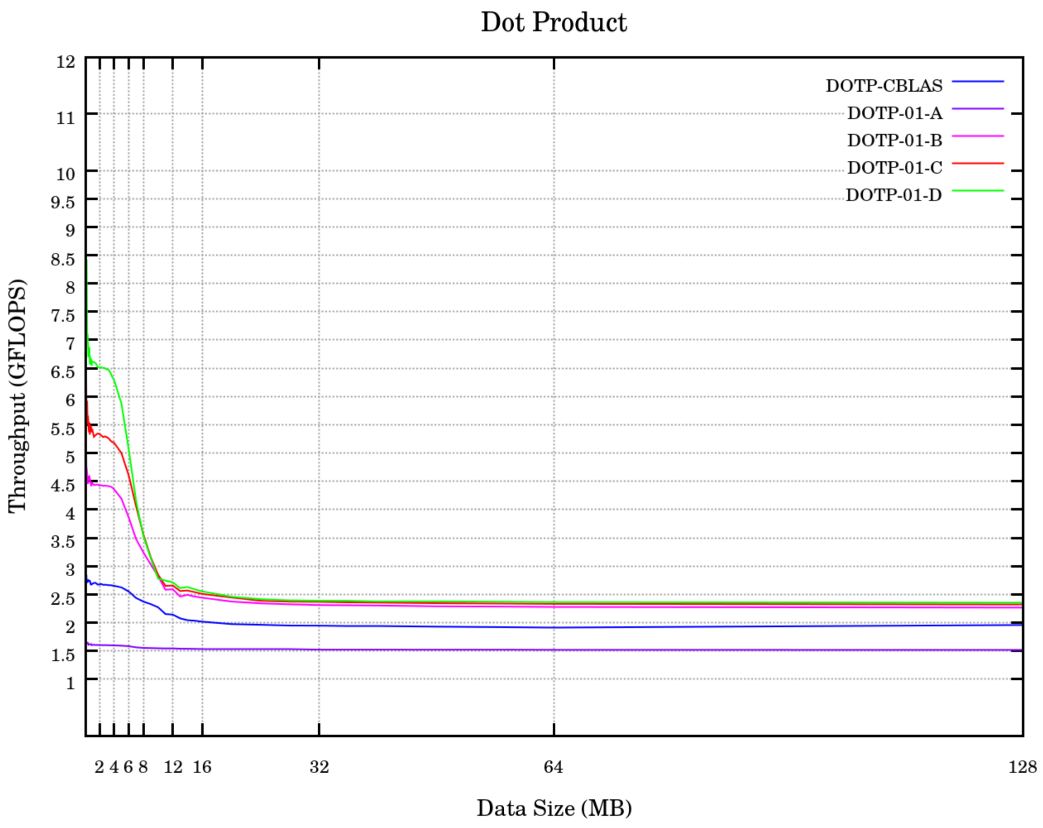


Figure 6.4: Stage 01. Results of the dot product.

From the theoretical values in Table 6.2, both DOTP-01-A and DOTP-01-C should be computing bounded due to its low IPC value. However, it can be observed from the results that only the implementation DOTP-01-A is actually bounded by the TMP. The REP_{RAM} obtained from the implementation DOTP-01-C is greater than its specific TMP but still lower than its TBP_{RAM}. This is because when applying SSE2 instructions the IPC can be easily optimized by the compiler or the processor as deduced in Section 5.3.5. Also the DOTP-01-D presents an unexpected behavior since its REP_{MAX} is greater than its TMP. Nevertheless, the results of interest are those obtained for large data sizes and, under these conditions the best implementation is the DOTP-01-D.

The DOTP-01-D implementation designed for this project is performing greater than the corresponding operator within CBLAS library. However, it is not able to saturate the main-memory bandwidth since many cores are needed in order to saturate the bandwidth in a multi-core processor.

Results of AXPY

The results obtained from the benchmark by running the sequential *axpy* operators are listed in the Table 6.4 and plotted in Figure 6.5. The second column of the table refers to the maximum real performance (REP) obtained for each operator and the third column refers to the average performance obtained when the data is larger than 32MB, that is when it is supposed to be main-memory bounded (RAM bounded). Recall that most of the applications in computational science are main-memory bounded. Hence, the results of interest for this project are these obtained from large data sizes (32MB are considered large enough for the i5 4670k since its cache size is 6MB).

Version	REP_{MAX}	REP_{RAM}	$\text{REP}_{\text{RAM}}/\text{TBP}_{\text{RAM}}$
CBLAS	4.295	1.644	0.774
AXPY-01-A	10.559	1.793	0.844
AXPY-01-B	10.559	1.805	0.850
AXPY-01-C	8.590	1.759	0.828
AXPY-01-D	9.843	1.786	0.841

Table 6.4: Stage 01. Results of the generalized vector addition.

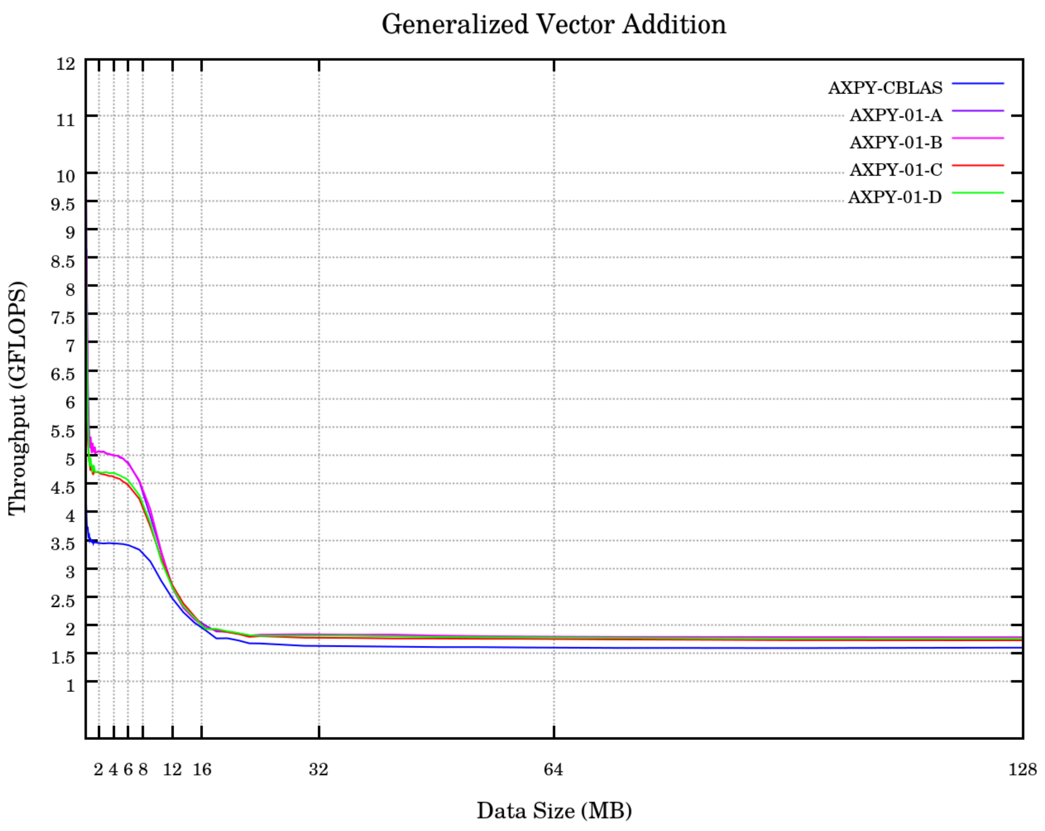


Figure 6.5: Stage 01. Results of the generalized vector addition.

From the results it can be observed that the REP_{MAX} is greater than the theoretical values of the TMP in Table 6.2 for all the implementations. As deduced previously, some automatic optimizations might have been done to the IPC. Nevertheless, the results of interest are those obtained for large data sizes and under these conditions the best implementation is the AXPY-01-B. In addition, the ratio $\text{REP}_{\text{RAM}}/\text{TBP}_{\text{RAM}}$ obtained from the *axpy* results are better than those obtained with the *dotp*. In other words, the AXPY-01-B is closer to the TBP than the DOTP-01-D.

Results of SMVP

The results obtained from the benchmark by running the sequential *smvp* operators are listed in the Table 6.5 and plotted in Figure 6.6. The second column of the table refers to the maximum real performance (REP) obtained for each operator and the third column refers to the average performance obtained when the data is larger than 32MB, that is when it is supposed to be main-memory bounded (RAM bounded). Recall that most of the applications in computational science are main-memory bounded. Hence, the results of interest for this project are these obtained from large data sizes (32MB are considered large enough for the i5 4670k since its cache size is 6MB).

Version	REP_{MAX}	REP_{RAM}	$\text{REP}_{\text{RAM}}/\text{TBP}_{\text{RAM}}$
SMVP-01-A	3.456	2.286	0.555
SMVP-01-B	3.524	2.207	0.535

Table 6.5: Stage 01. Results of the sparse matrix-vector product.

From results it can be observed that the ratio the ratio $\text{REP}_{\text{RAM}}/\text{TBP}_{\text{RAM}}$ of the *smvp* is the lowest among the evaluated operators. In other words, the *smvp* is the least efficient of the operators. This is because both implementations of the *smvp* are not able to take full advantage of the cache reuse. Hence, the entries of the vector are read three times instead of one (this behavior is predicted in Section 6.3).

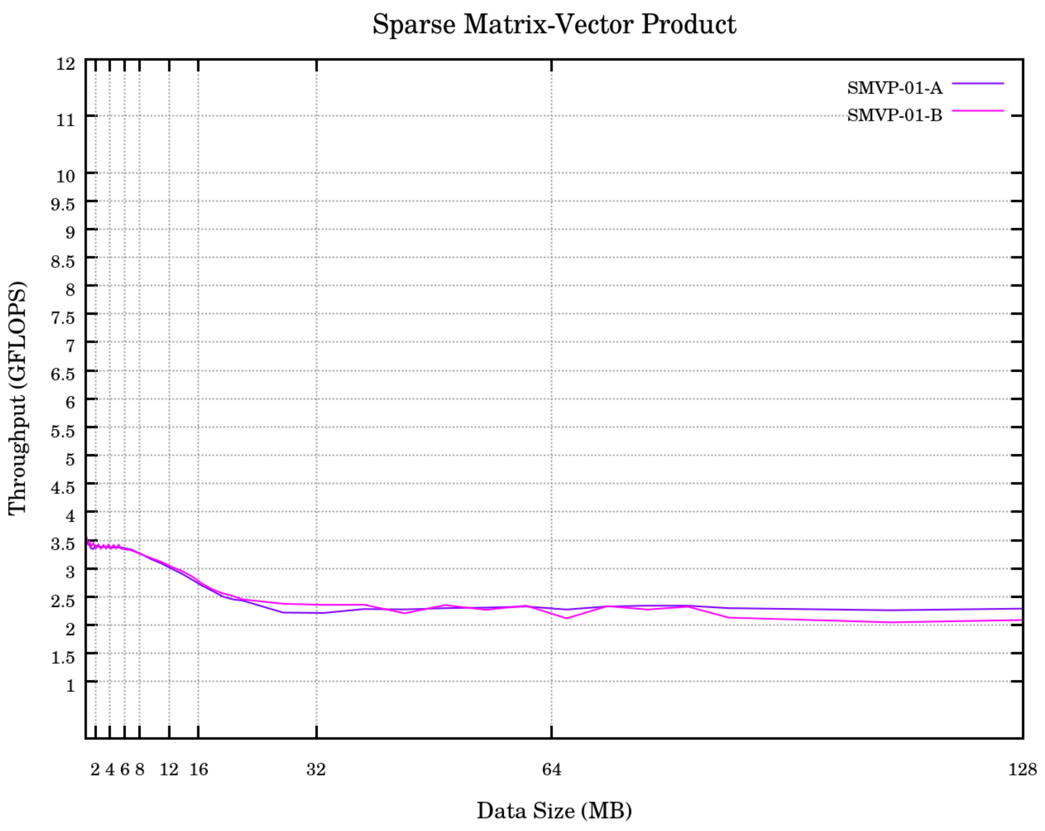


Figure 6.6: Stage 01. Results of the sparse matrix-vector product.

Conclusions

The conclusions obtained from the first stage evaluation are:

- It is not possible to saturate the main-memory bandwidth using a single core of a multi-core processor.
- The sequential operators implemented in the XLF are performing very good. All the optimized implementations of the *dotp* and *axpy* operators yield a better result than their corresponding reference found in CBLAS.
- It may be difficult to implement the algorithms in a computational application and take full advantage of the cache reuse (e.g. the sparse matrix-vector relative

performance is the worst among all the operators when it is memory bounded because it had to read three times the entries of the vector instead of one).

- It is necessary to optimize the IPC of the applications in order to be main-memory bounded instead of computing bounded for large problem sizes (i.e. the DOTP-01-A is computing bounded by its TMP instead of its TBP).
- The SIMD extensions are a great optimization for applications that fits within the cache memory. However, for main-memory bounded applications, using SIMD results in a limited improvement (e.g. the *dotp* performed about 3% better when using SIMD) or even in a performance reduction (e.g. the *axpy* performed about 1% worse when using SIMD).
- It has not been found any pattern that relates the improvement in performance to the optimization applied (e.g. the *dotp* improved with all the optimizations, the *axpy* only improved with the loop unrolling and finally the *smvp* did not improve even with loop unrolling). Thus, the behavior of performance versus type of optimization also depends on the algorithm itself.

6.4.2 Stage 02: Shared Memory Parallelization

The objective of this section is to develop a set of algebraic kernels able to take full advantage of the main-memory bandwidth. Although the number of cores is multiplied by 4 in this case, the objective is to saturate the memory bandwidth and get a performance close to the TBP_{RAM}, that is improving by about 20% the performance obtained from the operators in *Stage 01* instead of multiplying it by 4. Once the effectiveness of this operators is validated, a distributed parallel code can be designed by running locally in every node the parallel operators designed in this section.

Parallelization for shared memory systems aims to make the application to use all the cores available in the same node in an efficient way, that is sharing the work and also the data stored within the main memory in order to avoid unnecessary data movement.

For this purpose, different options have been implemented for each operator using the OpenMP standards (see Section 2.2 for instance). The Table 6.6 shows, for each implementation, its optimizing features as well as its specific IPC, TMP, AI and TBP_{RAM} (see Section 5.3 for further information about those terms); Recall that the last column refers to the theoretical bounded performance for large problem sizes which need to be stored in the main memory (RAM).

Two different implementations of the *dotp* and the *axpy* are parallelized in this section because from the results in the previous section the advantage in performance is not clear enough. In addition, the operators have been parallelized in both manual and automatic ways with the objective of analyzing if there is an overhead related to the automatic load balancing.

Version	Features	IPC	TMP	AI	TBP_{RAM}
DOTP					
DOTP-02-A	OMP Manual + DOTP-01-A	8/5	6.400	0.125	3.200
DOTP-02-B	OMP Automatic + DOTP-01-A	8/5	32.000	0.125	3.200
DOTP-02-C	OMP Manual + DOTP-01-D	40/5	6.400	0.125	3.200
DOTP-02-D	OMP Automatic + DOTP-01-D	40/5	32.000	0.125	3.200
AXPY					
AXPY-02-A	OMP Manual + AXPY-01-A	40/5	32.000	0.083	2.125
AXPY-02-B	OMP Automatic + AXPY-01-A	40/5	32.000	0.083	2.125
AXPY-02-C	OMP Manual + AXPY-01-B	40/5	32.000	0.083	2.125
AXPY-02-D	OMP Automatic + AXPY-01-B	40/5	32.000	0.083	2.125
SMVP					
SMVP-02-A	OMP Manual + SMVP-01-A	40/5	32.000	0.161	4.122
SMVP-02-B	OMP Automatic + SMVP-01-A	40/5	32.000	0.161	4.122

Table 6.6: Stage 02. Theoretical parameters of the operators.

The results within gray colored cells represent the application's theoretical limit in performance for large data sizes. Notice that this time all the implementations are supposed to be memory bounded.

Results of DOTP

The results obtained from the benchmark by running the parallelized *dotp* operators are listed in the Table 6.7 and plotted in Figure 6.7. The second column of the table refers to the maximum real performance (REP) obtained for each operator and the third column refers to the average performance obtained when the data is larger than 32MB, that is when it is supposed to be main-memory bounded (RAM bounded). Recall that most of the applications in computational science are main-memory bounded. Hence, the results of interest for this project are these obtained from large data sizes (32MB are considered large enough for the i5 4670k since its cache size is 6MB).

Version	REP_{MAX}	REP_{RAM}	REP_{RAM}/TBP_{RAM}
DOTP-01-D	10.290	2.381	0.744
DOTP-02-A	6.443	2.959	0.925
DOTP-02-B	30.065	3.054	0.954
DOTP-02-C	6.924	2.956	0.924
DOTP-02-D	30.315	3.050	0.953

Table 6.7: Stage 02. Results of the dot product.

The results for the *dotp* in the second stage's evaluation are consistent with the values in Table 6.6. The best bounded performance for large problem sizes has been obtained by the automatic and optimized implementation, the DOTP-02-B. However, the difference between the DOTP-02-B and the DOTP-02-D is very small. In addition, the maximum REP has been achieved by the DOTP-02-D implementation. Nevertheless, since only the bounded performance is important, the DOTP-02-B is considered the best option.

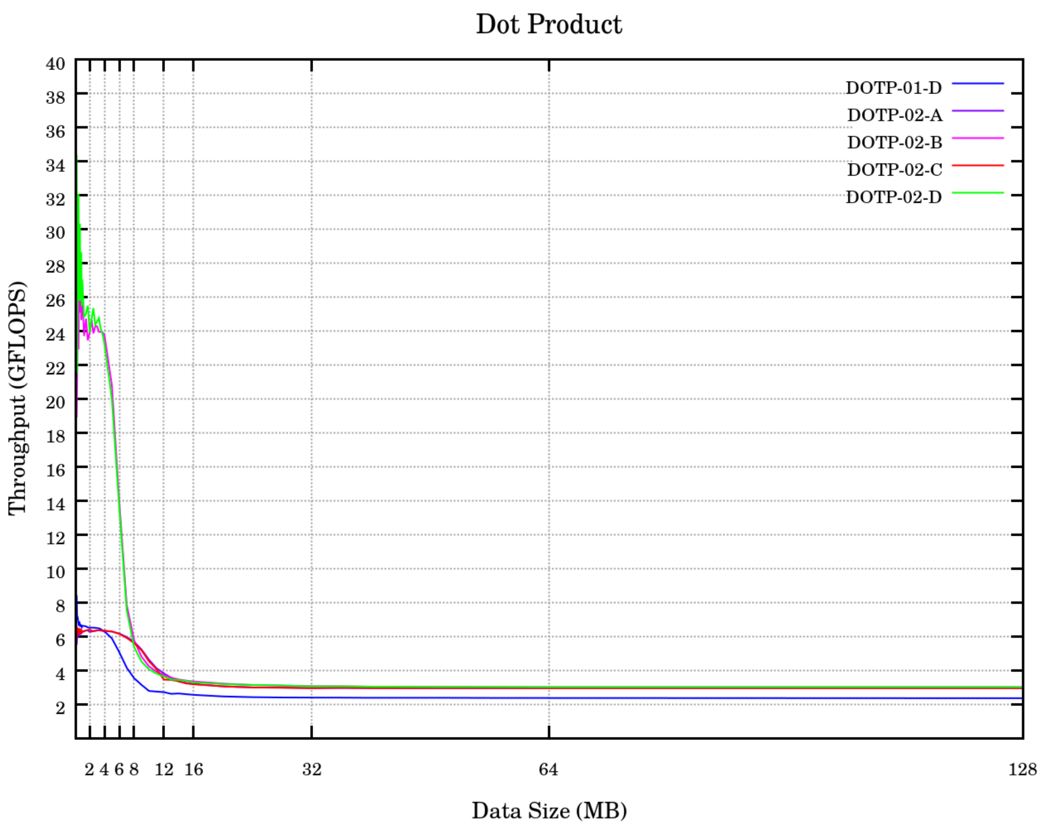


Figure 6.7: Stage 02. Results of the dot product.

Results of AXPY

The results obtained from the benchmark by running the parallelized *axpy* operators are listed in the Table 6.8 and plotted in Figure 6.8. The second column of the table refers to the maximum real performance (REP) obtained for each operator and the third column refers to the average performance obtained when the data is larger than 32MB, that is when it is supposed to be main-memory bounded (RAM bounded). Recall that most of the applications in computational science are main-memory bounded. Hence, the results of interest for this project are these obtained from large data sizes (32MB are considered large enough for the i5 4670k since its cache size is 6MB).

The results for the *axpy* in the second stage's evaluation are consistent with the values

Version	REP_{MAX}	REP_{RAM}	REP_{RAM}/TBP_{RAM}
AXPY-01-B	10.559	1.805	0.850
AXPY-02-A	34.360	2.008	0.945
AXPY-02-B	32.451	2.019	0.950
AXPY-02-C	34.360	2.021	0.951
AXPY-02-D	34.360	2.001	0.942

Table 6.8: Stage 02. Results of the generalized vector addition.

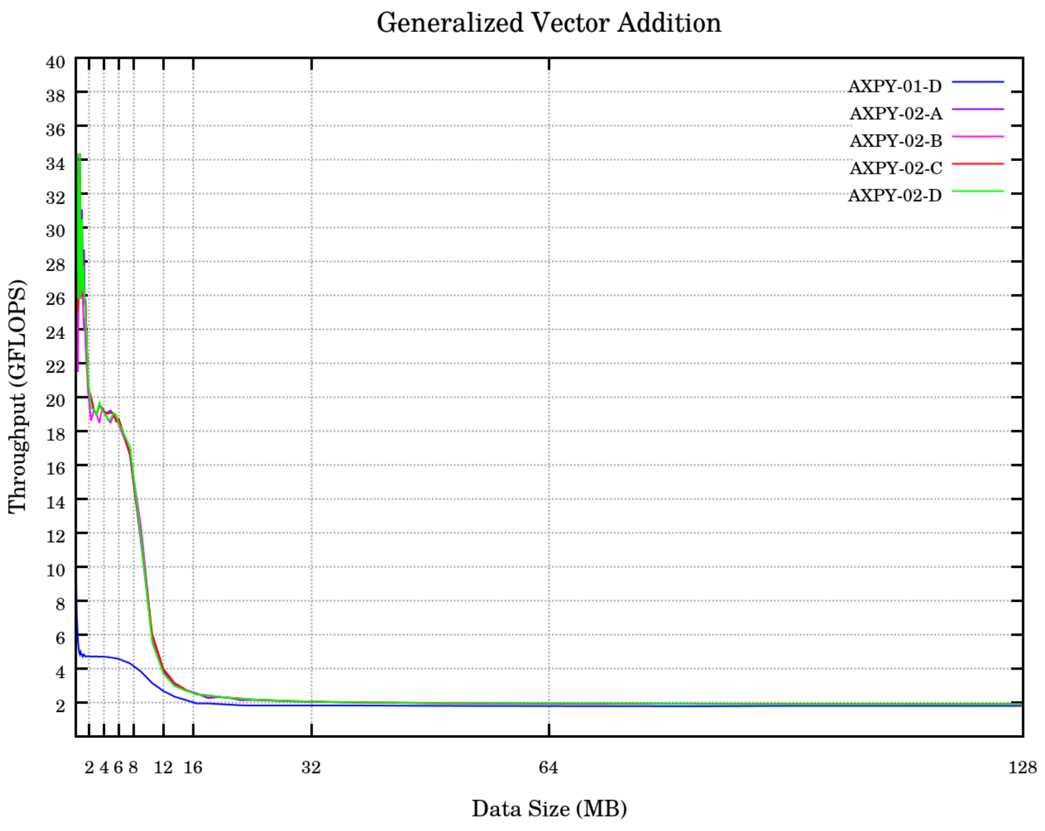


Figure 6.8: Stage 02. Results of the generalized vector addition.

in Table 6.6. In contrast to the parallel dot product, the maximum performance for large problem sizes has been obtained by the manual and simple implementation, the AXPY-02-C. However, the difference between the AXPY-02-C and the other versions is very small.

Results of SMVP

The results obtained from the benchmark by running the parallelized *smvp* operators are listed in the Table 6.9 and plotted in Figure 6.9. The second column of the table refers to the maximum real performance (REP) obtained for each operator and the third column refers to the average performance obtained when the data is larger than 32MB, that is when it is supposed to be main-memory bounded (RAM bounded). Recall that most of the applications in computational science are main-memory bounded. Hence, the results of interest for this project are these obtained from large data sizes (32MB are considered large enough for the i5 4670k since its cache size is 6MB).

Version	REP_{MAX}	REP_{RAM}	REP_{RAM}/TBP_{RAM}
SMVP-01-A	3.456	2.286	0.555
SMVP-02-A	1.471	1.213	0.294
SMVP-02-B	11.514	3.216	0.780

Table 6.9: Stage 02. Results of the sparse matrix-vector product.

The results for the *smvp* in the second stage's evaluation are totally clear: the maximum performance is obtained with the manual implementation. In addition, the REP_{RAM} obtained with the SMVP-02-B is greater than 3.200 GFLOPS, which would be the TBP_{RAM} if considering only the 2N value of cache reuse related to the east and west couplings. Thus this result implies that a small cache reuse related to the north and south couplings is still possible for large problem sizes.

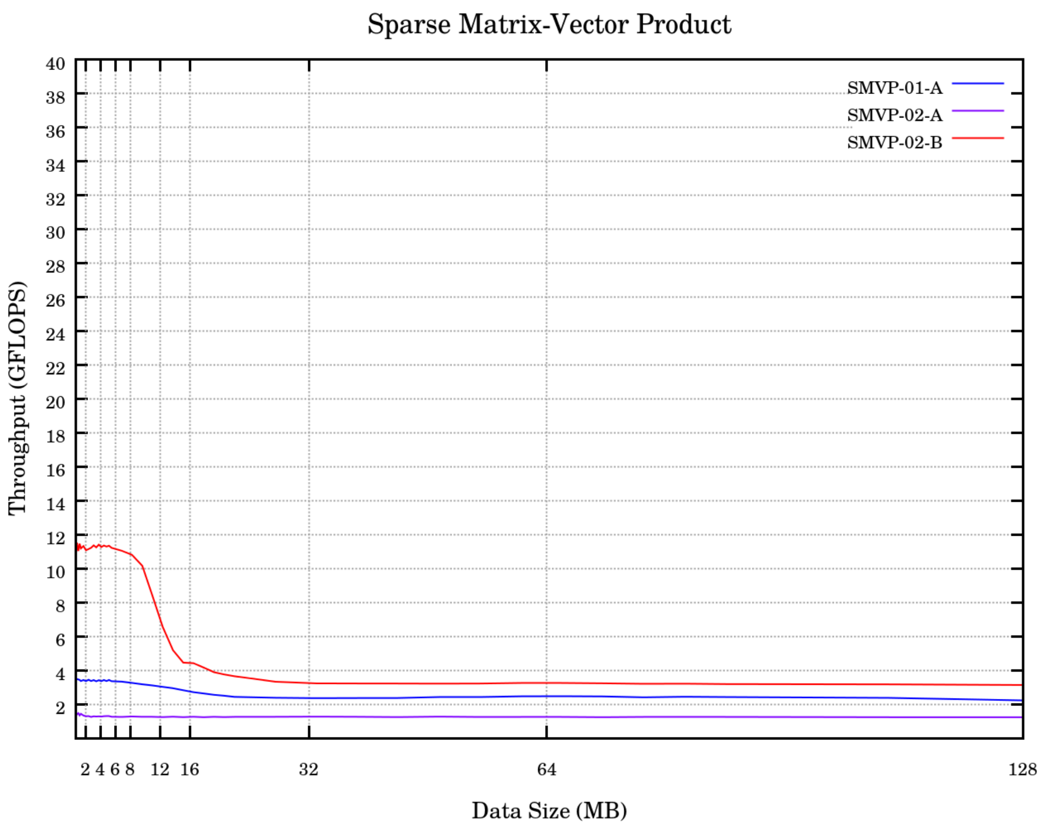


Figure 6.9: Stage 02. Results of the sparse matrix-vector product.

Conclusions

The conclusions obtained from the second stage evaluation are:

- In a main-memory bounded application, the objective of multi-threading is to saturate its memory bandwidth instead of multiplying the performance by the number of cores used. This is due to the bottleneck.
- Both *dotp* and *axpy* exceeded the 95% of the bandwidth. Since the pointer arithmetic has not been taken into account when calculating the number of FLOP, neither the interferences with the system's background nor the multi-threading management, the 95% can be considered a great result. Thus, the main-memory bandwidth can be considered saturated.

- The sparse matrix-vector product is still limited by the small cache reuse which is about 2N instead of 4N.
- Any pattern that relates the improvement in performance with the parallelization applied has been found (i.e. the *dotp* improved slightly with the automatic mode, the *axpy* improved slightly with the manual mode and finally the *smvp* sharply decreased with the automatic mode). Thus the behavior of performance versus multi-threading options also depends on the algorithm itself. Nevertheless, the differences in performance due to different OMP options are small.

6.4.3 Stage 03: Distributed Memory Parallelization

The objective of this section is to develop a set of algebraic kernels able to properly scale in a supercomputer in order to solve efficiently large problems or operations. The complexity of the communications increases with the number of nodes. Hence, the performance of the operators may be reduced due to the cost of the MPI communications.

Parallelization for distributed memory systems aims to make the application to distribute the work load among many nodes in an efficient way that is minimizing the communications and data transfers between nodes. For this purpose, one MPI version has been implemented for both *dotp* and *smvp* (see Section 2.3 for instance) to execute concurrently multiple instances of the optimal implementations of the operators from *Stage 02*. Notice that each MPI process will run a OpenMP application. Hence, the parallelization model is hybrid of MPI + OpenMP.

The Table 6.10 shows, for each operator, its theoretical main-memory bounded performance depending on the number of nodes. Recall that each node in MareNostrum III has two Intel Xeon E5-2670, then the total main-memory bandwidth per node is 102.400 GB/s. This way the available bandwidth increases with the number of nodes.

	DOTP-03	SMVP-03
Number of Nodes	TBP_{RAM}	TBP_{RAM}
1	12.800	16.486
2	25.600	36.972
3	38.400	49.458
4	51.200	65.944
5	64.000	82.430
6	76.800	98.916
7	89.600	115.402
8	102.400	131.888

Table 6.10: Stage 03. Theoretical parameters of the operators.

The MPI version of the *axpy* operator does not require communications because all operations are totally independent from the rest. For this reason, even though the hybrid version of the generalized vector addition has been implemented in the XLF, it has not beeen tested in this stage.

Results of DOTP

The results obtained from the benchmark by running the hybrid *dotp* operator are listed in the Table 6.11 and plotted in Figure 6.10. The third column refers to the average performance obtained when the data is larger than 120MB per node, that is when it is supposed to be main-memory bounded (RAM bounded). Recall that most of the applications in computational science are main-memory bounded. Hence, the results of interest for this project are these obtained from large data sizes (120MB are considered large enough for the nodes of MareNostrum III since its total cache is 40MB).

The results for the *dotp* in the third stage's evaluation are consistent with the values in Table 6.10. The performance starts from about 90% and decreases in about 20% as the number of nodes increases. This behavior is predicted and is due to the cost of the communications, which are more complex when the number of nodes is increased.

Number of Nodes	REP_{RAM}	REP_{RAM}/TBP_{RAM}
1	11.246	0.876
2	21.792	0.851
3	31.737	0.827
4	43.159	0.843
5	45.518	0.711
6	58.447	0.761
7	68.877	0.769
8	81.653	0.797

Table 6.11: Stage 03. Results of the dot product.

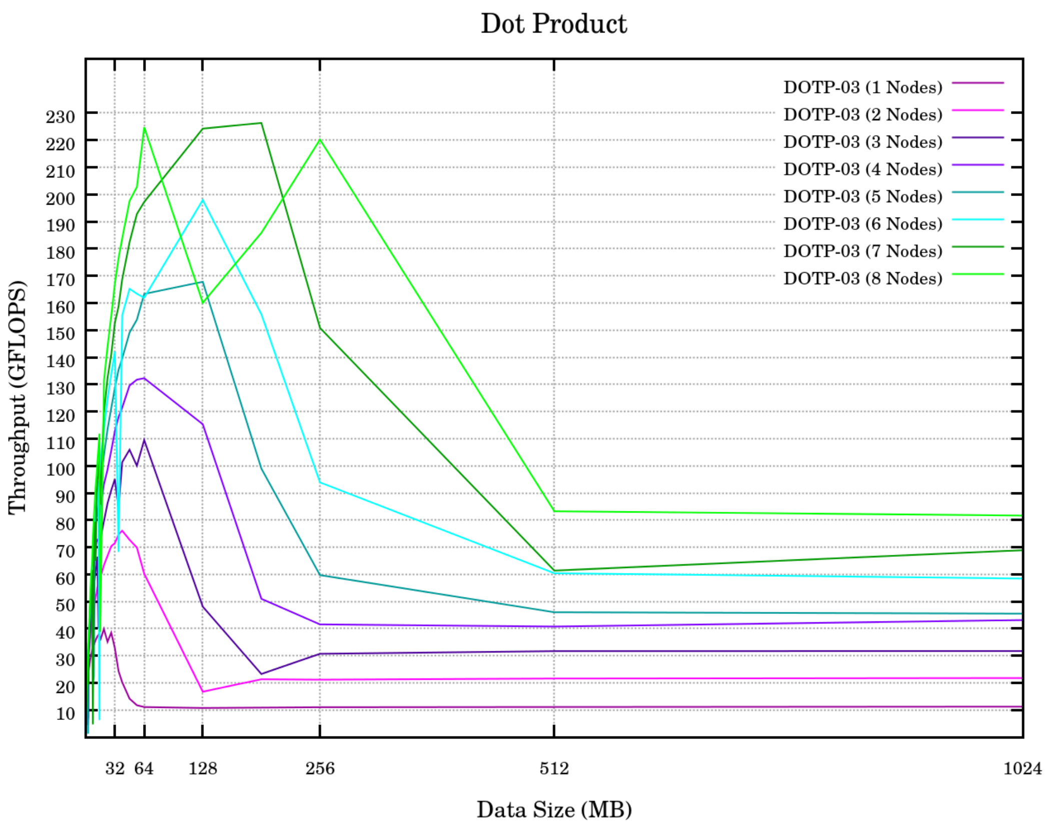


Figure 6.10: Stage 03. Results of the dot product.

Results of SMVP

The results obtained from the benchmark by running the hybrid *smvp* operator are listed in the Table 6.12 and plotted in Figure 6.11. The third column refers to the average performance obtained when the data is larger than 120MB per node, that is when it is supposed to be main-memory bounded (RAM bounded). Recall that most of the applications in computational science are main-memory bounded. Hence, the results of interest for this project are these obtained from large data sizes (120MB are considered large enough for the nodes of MareNostrum III since its total cache is 40MB).

Number of Nodes	REP_{RAM}	REP_{RAM}/TBP_{RAM}
1	11.321	0.687
2	22.373	0.605
3	32.253	0.652
4	41.460	0.629
5	47.330	0.574
6	55.052	0.557
7	72.939	0.632
8	83.559	0.634

Table 6.12: Stage 03. Results of the sparse matrix-vector product.

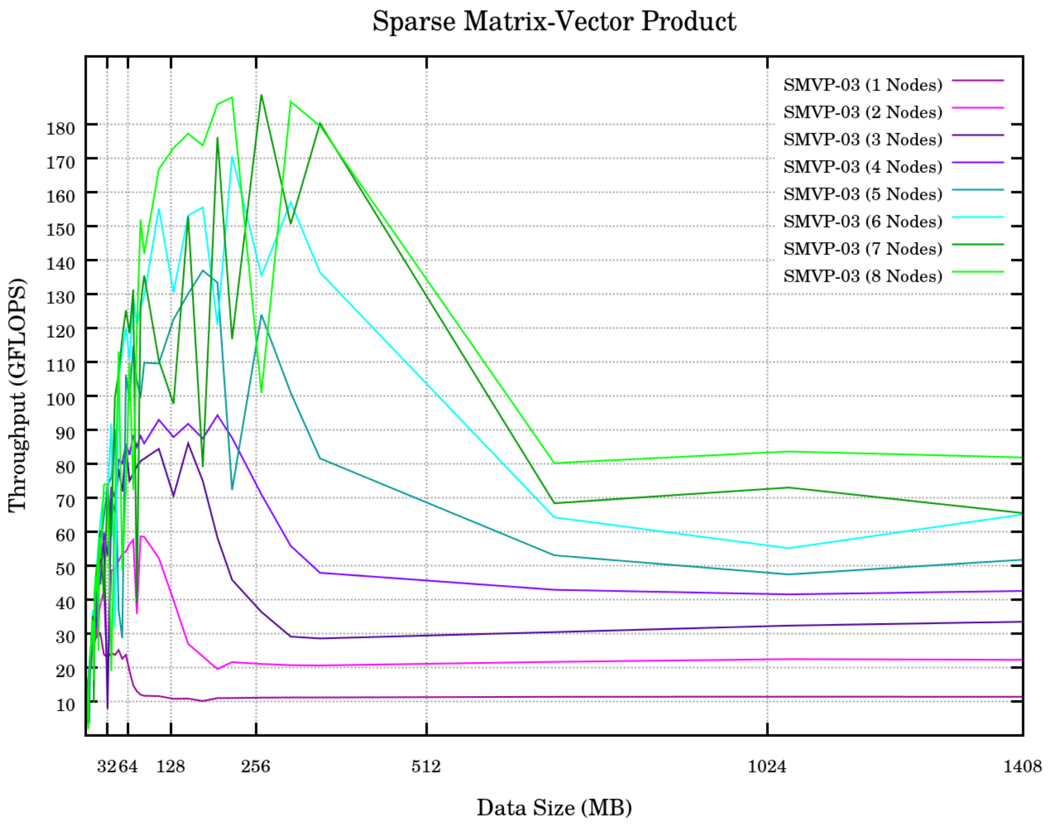


Figure 6.11: Stage 03. Results of the sparse matrix-vector product.

The results for the *smvp* in the third stage's evaluation are consistent with the values in Table 6.10. This time, the performance starts from about 70% and decreases in about 10% as the number of nodes increases.

Study of the Scalability

In computational science, scalability is the capability of a system to increase its total throughput under an increased load when resources (typically hardware) are added. Since the performance of the operators in *Stage 03* is evaluated by incrementing the number of cores or nodes used for running the operators, the throughput is expected to increase in the same way. However, the MPI communications involve a computational cost which is going to be evaluated below.

There are two common methods to evaluate the performance of a distributed application. In one hand, strong scaling studies the speed-up versus the number of nodes for a constant work-load. The ideal result should be a straight line such as $y = x$. On the other hand, weak scaling studies the performance versus the number of nodes while increasing the work-load the same as the computing power. The ideal result should be a constant straight line such as $y = C$.

The Figure 6.12 represents the strong scaling study, and the Figure 6.13 represents the weak scaling study.

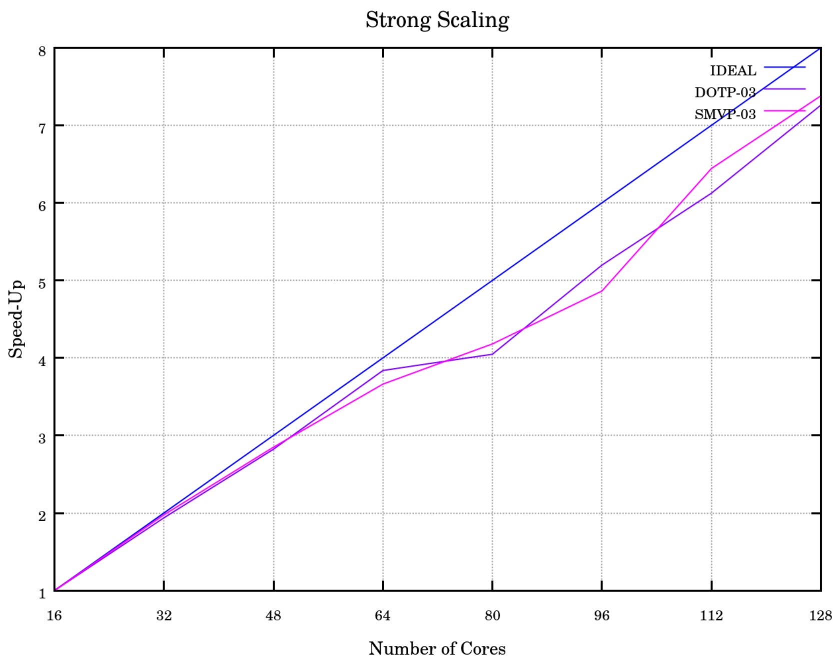


Figure 6.12: Stage 03. Study of the Strong Scaling behavior.

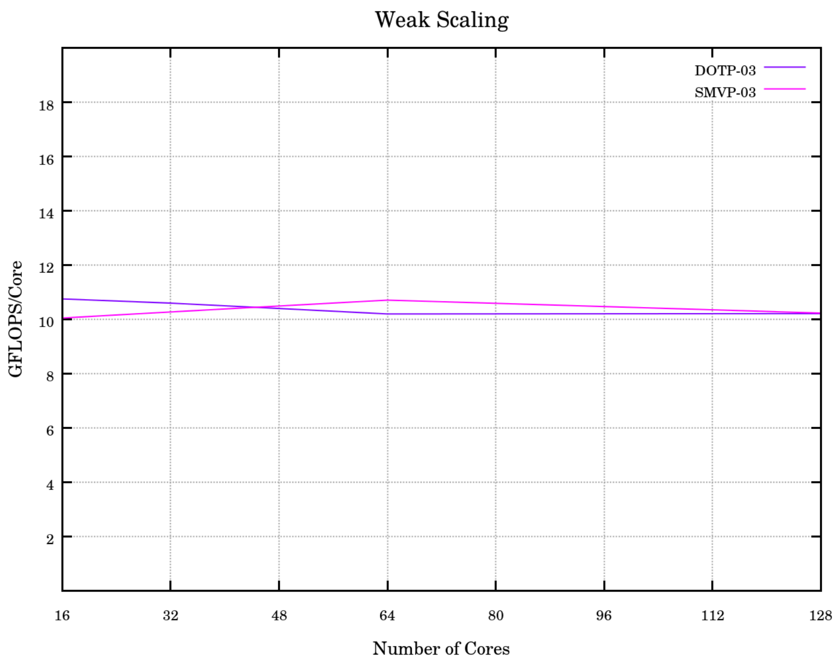


Figure 6.13: Stage 03. Study of the Weak Scaling behavior.

Conclusions

The conclusions obtained from the third stage evaluation are:

- The hybrid operators implemented in the XLF scale correctly when running in up to 8 nodes (that is 128 cores).
- The XLF library can be used to solve CFD problems efficiently in large computer clusters.
- The hybrid parallel concept is very useful in order to take full advantage of every shared-memory systems within a distributed-memory system. In addition, it is very confortable to implement a hybrid code.
- The MPI communications present an overhead that makes the performance of the operators to decrease as the number of nodes or cores increase.

Chapter 7

Environmental Impact Analysis

In order to identify and, if possible, minimize all the negative effects that the project may cause to the environment, it is necessary to carry out an environmental impact analysis. This analysis takes into account all project phases, from the preliminary study to decommissioning in order to be as environmentally friendly as possible.

Recall that the objective of this project is to develop an efficient parallel tool in order to reduce the power consumption of many computational applications. For this reason, the project is already environmentally friendly.

7.1 Preliminary Study and Draft Project

As this section deals with the documentation and study prior to code development, the environmental impact associated with the design phase has been minimal. Only the office supplies that may have been used as well as the energy consumption of the different electronic devices required for the drafting phase such as lighting or computers is considered.

During this phase HVAC systems have been used responsibly by setting a maximum of 20℃in winter and a minimum of 27℃in summer. The use of lighting has been necessary but always using low-energy fluorescent. Most of the information and books have been consulted via Internet in order to minimize the use of paper. A laptop with low power requirements has been used for gathering, organizing and reading information. Recycled paper has been used for printing documents when it has been required. Finally, the only office supplies which have been consumed are recycled paper and two pens.

7.2 Memory Writing

This section deals with the environmental impact related to write the memory and creating all the documentation required. This way, the impact associated with this phase is the same as in Section 7.1 since it has been carried out in the same working environment.

7.3 Construction Phase

This section deals with the environmental impact related to the construction stage. This project does not require any material construction because only a computational application has been developed. Although the code has been developed in the same working environment and thus the environmental impact is the mostly the same as considered in Section 7.1, two additional computers with different architectures have been required during this task. In order to minimize the energy consumption, only one computer has been used at the same time thus the other two have been off.

7.4 Operational Phase

As this section deals with the execution of the computational applications, only the environmental impact associated with the power consumption of the computers is considered since the execution requires no supervision. Furthermore, the object of this project is to implement an efficient code hence the environmental impact associated with this phase is minimized.

7.5 Decommissioning

As this project is about developing a computational application, all the elements are computer files hence to eliminate the project only requires to delete the computer files.

Chapter 8

Budget Summary

In this section the cost estimate of the project is attached. It takes into account the costs of hardware, research and development and the energy consumption. Consult the **Cost Estimate** document attached to the project for further information of each item.

8.1 Cost Estimate

The total cost estimate is attached in Table 8.1. All the items are detailed in the sections above.

Item	Cost [€]
Hardware	718.80
Research and development	24,408.00
Power consumption	219.05
Cost Estimate (before IVA)	25,345.85
IVA (21%)	5,322.63
Cost Estimate	30,668.48

Table 8.1: Cost Estimate.

Thirty thousand six hundred sixty-eight euro and forty-eight cent.

Chapter 9

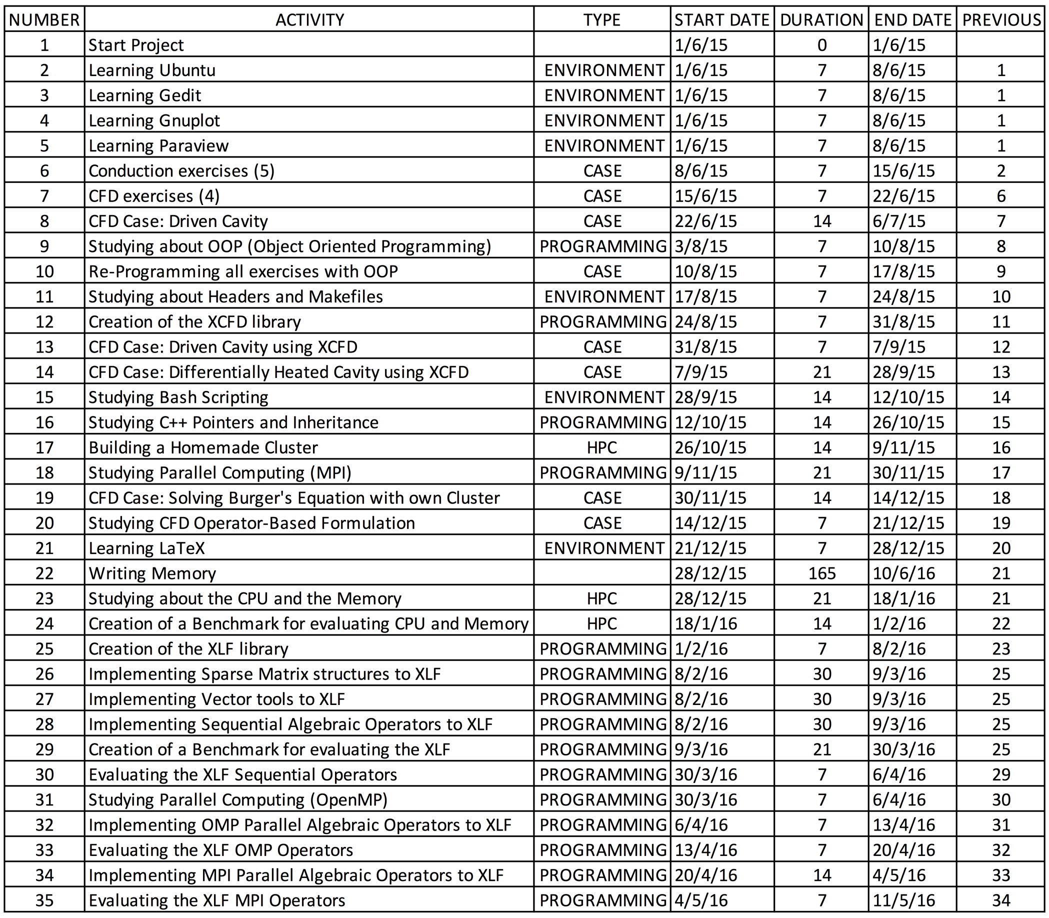
Project Planning

The present project was developed between the 1st of June of 2015 and the 10th of June of 2016. Many tasks have been carried out during this period. This chapter aims to give the reader an idea about how the tasks have been planned and organized.

9.1 List of Tasks

In the table attached in Figure 9.1 all the tasks realized during the development of the project are listed. Below is a short summary of each task.

1. **Start Project**.
2. **Learning Ubuntu**: Ubuntu is a Debian-based Linux operative system very useful for programmers and developers.
3. **Learning Gedit**: Gedit is the default text editor of the GNOME desktop environment. Designed as a general-purpose text editor, gedit emphasizes simplicity and



NUMBER	ACTIVITY	TYPE	START DATE	DURATION	END DATE	PREVIOUS
1	Start Project		1/6/15	0	1/6/15	
2	Learning Ubuntu	ENVIRONMENT	1/6/15	7	8/6/15	1
3	Learning Gedit	ENVIRONMENT	1/6/15	7	8/6/15	1
4	Learning Gnuplot	ENVIRONMENT	1/6/15	7	8/6/15	1
5	Learning Paraview	ENVIRONMENT	1/6/15	7	8/6/15	1
6	Conduction exercises (5)	CASE	8/6/15	7	15/6/15	2
7	CFD exercises (4)	CASE	15/6/15	7	22/6/15	6
8	CFD Case: Driven Cavity	CASE	22/6/15	14	6/7/15	7
9	Studying about OOP (Object Oriented Programming)	PROGRAMMING	3/8/15	7	10/8/15	8
10	Re-Programming all exercises with OOP	CASE	10/8/15	7	17/8/15	9
11	Studying about Headers and Makefiles	ENVIRONMENT	17/8/15	7	24/8/15	10
12	Creation of the XCFD library	PROGRAMMING	24/8/15	7	31/8/15	11
13	CFD Case: Driven Cavity using XCFD	CASE	31/8/15	7	7/9/15	12
14	CFD Case: Differentially Heated Cavity using XCFD	CASE	7/9/15	21	28/9/15	13
15	Studying Bash Scripting	ENVIRONMENT	28/9/15	14	12/10/15	14
16	Studying C++ Pointers and Inheritance	PROGRAMMING	12/10/15	14	26/10/15	15
17	Building a Homemade Cluster	HPC	26/10/15	14	9/11/15	16
18	Studying Parallel Computing (MPI)	PROGRAMMING	9/11/15	21	30/11/15	17
19	CFD Case: Solving Burger's Equation with own Cluster	CASE	30/11/15	14	14/12/15	18
20	Studying CFD Operator-Based Formulation	CASE	14/12/15	7	21/12/15	19
21	Learning LaTeX	ENVIRONMENT	21/12/15	7	28/12/15	20
22	Writing Memory		28/12/15	165	10/6/16	21
23	Studying about the CPU and the Memory	HPC	28/12/15	21	18/1/16	21
24	Creation of a Benchmark for evaluating CPU and Memory	HPC	18/1/16	14	1/2/16	22
25	Creation of the XLF library	PROGRAMMING	1/2/16	7	8/2/16	23
26	Implementing Sparse Matrix structures to XLF	PROGRAMMING	8/2/16	30	9/3/16	25
27	Implementing Vector tools to XLF	PROGRAMMING	8/2/16	30	9/3/16	25
28	Implementing Sequential Algebraic Operators to XLF	PROGRAMMING	8/2/16	30	9/3/16	25
29	Creation of a Benchmark for evaluating the XLF	PROGRAMMING	9/3/16	21	30/3/16	25
30	Evaluating the XLF Sequential Operators	PROGRAMMING	30/3/16	7	6/4/16	29
31	Studying Parallel Computing (OpenMP)	PROGRAMMING	30/3/16	7	6/4/16	30
32	Implementing OMP Parallel Algebraic Operators to XLF	PROGRAMMING	6/4/16	7	13/4/16	31
33	Evaluating the XLF OMP Operators	PROGRAMMING	13/4/16	7	20/4/16	32
34	Implementing MPI Parallel Algebraic Operators to XLF	PROGRAMMING	20/4/16	14	4/5/16	33
35	Evaluating the XLF MPI Operators	PROGRAMMING	4/5/16	7	11/5/16	34

Figure 9.1: List of Tasks.

ease of use. It includes tools for editing source code and structured text such as markup languages.

4. **Learning Gnuplot**: Gnuplot is a portable command-line driven graphing utility available in Ubuntu and many other platforms. The source code is copyrighted but freely distributed (i.e., you don't have to pay for it). It was originally created to allow scientists and students to visualize mathematical functions and data interactively.

5. **Learning ParaView**: ParaView is an open-source, multi-platform data analysis and visualization application. ParaView users can quickly build visualizations to analyze their data using qualitative and quantitative techniques.

6. **Conduction Exercises**: During this task a set of five conduction exercises (from

one-dimensional steady problems to two-dimensional transient problems) was realized.

7. **CFD Exercises**: During this task a set of four CFD exercises (from one-dimension steady problems to two-dimensional transient problems) was realized.
8. **CFD Case: Driven Cavity**: During this task the Driven Cavity exercise (two-dimensional transient laminar problem) was solved. The results were compared to the available bibliography and also an animation was produced with ParaView.
9. **Studying about OOP**: Object-oriented programming is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.
10. **Re-Programming all exercises with OOP**: During this task all the problems that were solved previously were re-programmed in order to practice the OOP concept.
11. **Studying about Headers and Makefiles**: In order to create big computational applications, the code may be split up into a set of files. This also increases the complexity of compiling the code. The makefiles allows the user to automatically compile and link a big set of files.
12. **Creation of the XCFD library**: A library was designed for solving CFD problems in a generic way during this task. The library included many objects and utilities for solving two-dimensional CFD laminar problems.
13. **CFD Case: Driven Cavity using XCFD**: During this task the Driven Cavity was re-programmed using the functions, tools and utilities included in the XCFD.
14. **CFD Case: Differentially Heated Cavity using XCFD**: During this task the Differentially Heated Cavity (two-dimensional CFD laminar problem with natural convection) was solved using the functions, tools and utilities included in the XCFD.
15. **Studying Bash Scripting**: Bash is a command processor that typically runs in a text window, where the user types commands that cause actions. Bash can also

read commands from a file, called a script. It is very useful to automate repetitive actions such as running a program with different inputs for getting different results.

16. **Studying Pointers and Inheritance**: During this task the programming skills were improved in order to be ready to face new problems.

17. **Building a Homemade Cluster**: During this task the concepts related to networks and cluster building were studied. In addition, a small cluster was built with a total of 4 nodes.

18. **Studying Parallel Computing (MPI)**: MPI is a parallel interface useful for parallelizing codes that are going to be run within a distributed memory system such as a cluster or a supercomputer.

19. **CFD Case: Solving Burger's Equation with own Cluster**: The Burger's Equation is a typical CFD problem for starting to study the turbulent flow. This problem was parallelized using MPI and executed within the homemade cluster.

20. **Studying CFD Operator-Based Formulation**: The governing equations in CFD can be discretized in an operator-based formulation. This way most of the CFD problems are reduced to three basic algebraic operations (*dotp*, *axpy* and *smvp*) which are the object of this project.

21. **Learning LaTeX**: LaTeX is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is the *de facto* standard for the communication and publication of scientific documents.

22. **Writing Memory**: This task is about writing the project's memory.

23. **Studying about the CPU and the Memory**: In order to evaluate the performance of the algebraic kernel implemented in this project, it is necessary to previously learn how hardware works and what are the bottlenecks.

24. **Creation of a Benchmark for evaluating CPU and Memory**: In order to learn deeper the CPU and Memory behavior, this task is carried out. A system bench-

mark was developed in order to experimentally understand the behavior of the CPU and the Memory.

25. **Creation of the XLF library**: During this task the file-system of the XLF was designed.

26. **Implementing Sparse Matrix structures to XLF**: This task involves both implementing and evaluating the sparse matrix data structures as well as tools and utilities like printing, transposing and many others.

27. **Implementing Vector tools to XLF**: This task involves both implementing and evaluating vector tools and utilities such as functions for calculating the norm of a vector, the absolute and relative error of two vectors and many others.

28. **Implementing Sequential Algebraic Operators to XLF**: This task involves implementing the algebraic kernels and also check whether the function operate correctly. Many different implementations of each operator were included in order to compare the effectiveness of different optimizations.

29. **Creation of a Benchmark for evaluating the XLF**: During this task a benchmark was designed for evaluating the performance of the XLF methods. The benchmark creates and initializes the input data necessary for running the algebraic kernels. In addition, many tools and utilities for calculating time and performance were developed.

30. **Evaluating the XLF Sequential Operators**: During this task the all the implementations of the sequential operators were evaluated. This task also involves the post-processing of the data with gnuplot scripts.

31. **Studying Parallel Computing (OpenMP)**: Before parallelizing the operators for shared memory system, it was necessary to study the basics of OpenMP.

32. **Implementing OMP Parallel Operators to XLF**: Many different implementations were included in order to compare the effectiveness of different OpenMP options.

33. **Evaluating the XLF OMP Operators**: During this task all the implementations of the OMP parallel operators were evaluated. This task also involves the post-processing of the data with gnuplot scripts.

34. **Implementing Hybrid Parallel Operators to XLF**: The OMP parallel operators are optimized for running in an entire node. For this reason, only one MPI version is implemented for each operator. The MPI only has to distribute the work-load among the available MPI processes, then the OMP operators are executed.

35. **Evaluating the XLF MPI Operators**: During this task the scalability of the Hybrid parallel operators was evaluated. This task also involves the post-processing of the data with gnuplot scripts.

9.2 Gantt Chart

A Gantt chart is a type of bar chart, adapted by Karol Adamiecki in 1896 and independently by Henry Gantt in the 1910s, that illustrates a project schedule. Gantt charts illustrate the start and finish dates of the terminal elements and summary elements of a project.

The Gantt chart of the tasks listed above is attached in Figure 9.2.

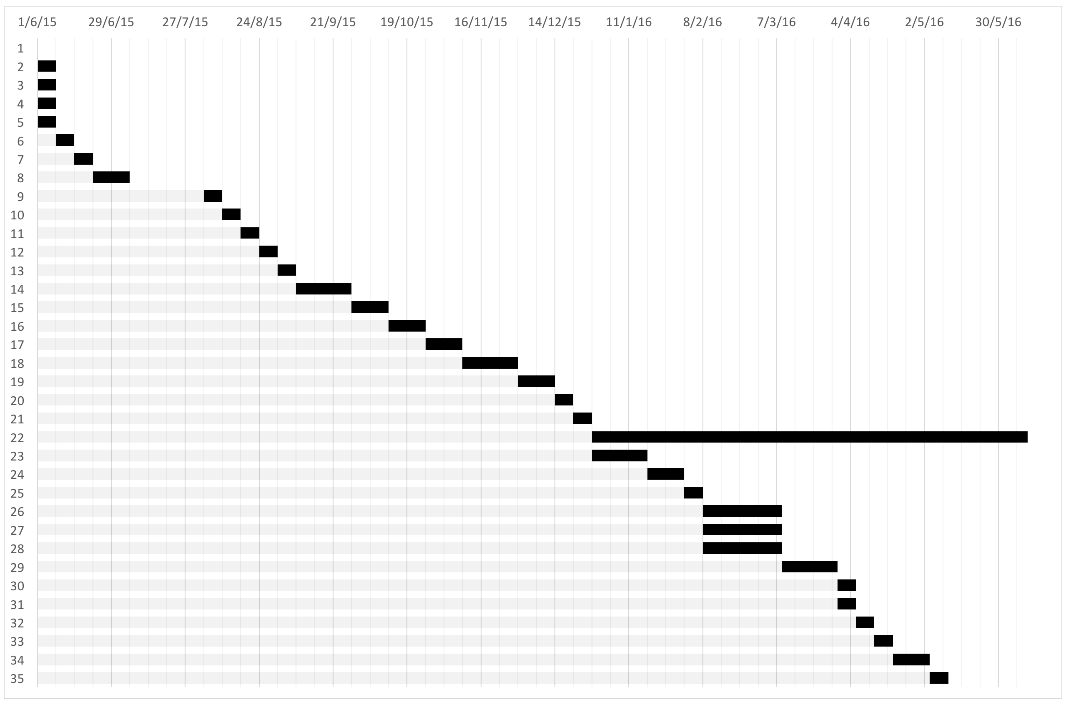


Figure 9.2: Gantt Chart.

Chapter 10

Conclusions

In this chapter, the conclusions after the accomplishment of the present project are written.

10.1 General Conclusions

The object of this project is considered accomplished. A powerful efficient, parallel computational application able to perform three algebraic operations has been developed and evaluated obtaining great results of performance and scalability. In addition, the relation between CFD algorithms and algebraic operators has been mathematically introduced.

The general conclusions about the results are:

- The numerical methods are very useful for solving the Navier-Stokes system of equations since its analytical solution is only known for very simple cases such as one-dimensional steady flow.

- In order to solve the Navier-Stokes equations using a numerical method, computers are needed.
- Although the hardware industry is in constant development and the computing elements are becoming more efficient (i.e. the ratio of power consumption to operations per second is increasing), the total power consumption of clusters and supercomputer is very high. For this reason it is a must to develop efficient computational tools in order to minimize the energy consumption.
- It is very important to also develop computational tools able to evaluate the performance of another computational application and compare the throughput with the bottlenecks (e.g. the benchmark developed in this project is able to do it).
- To design an efficient, portable algebra library is a good deal. In one hand, the CFD code is easily written using the algebraic operators. On the other hand, to improve or implement the CFD code in a specific computational architecture becomes simple since only the operators need to be changed. In addition, the algebra library is suitable not only for CFD codes but for every application that may involve solving differential equations.

This project is intended to be an introduction to research and development. For this reason, during the realization of the project, its author has deepened in many fields of science and engineering:

- mathematical formulation,
- numerical methods,
- computational fluid dynamics,
- programming in C++,
- computational science,

- Linux OS.

10.2 Future Work

The computing science is in constant development. For this reason, this project is considered just the tip of a large iceberg. The tasks that are going to be carried out from now are:

- Deepen in mathematical formulation, specifically in operator-based formulation.
- Study the features of the new computational architectures.
- Evaluate the limit in scalability for the actual XLF (i.e. the maximum number of nodes for which the performance increases).
- Implement the operators in new architectures (e.g. GPU or Intel XeonPhi).
- Implement the operators using new programming paradigms (e.g. OpenCL or CUDA).
- Deepen in mathematical formulation in order to be able to solve more generic CFD cases (i.e. eliminate hypotheses such as Boussinesq assumption, incompressible flow and so on).
- Generalize the data structures and algebraic operators in order to adapt the code to different numerical methods (e.g. unstructured grids, three-dimensional problems or collocated meshes).

Appendices

Appendix A

Solved Cases

During the realization of this project, many CFD cases have been solved:

- One-dimension, steady state conduction problem.
- Two-dimension, steady state conduction problem.
- One-dimension, unsteady conduction problem.
- Two-dimension, unsteady conduction problem.
- One-dimension, steady state convection-diffusion problem.
- Two-dimension, steady state convection-diffusion problem.
- One-dimension, unsteady convection-diffusion problem.
- Smith-Hutton problem.
- Driven cavity problem.
- Differentially heated cavity problem.
- Burguers Equation.

Below is a short summary of some of the most relevant cases that have been solved during.

A.1 Smith-Hutton Case

The objective of this case is to obtain the steady state solution of the Smith-Hutton problem, described in [27]. To do so, a two-dimensional convection-diffusion equation must be solved numerically in a rectangular domain (see Figure A.1) with the prescribed velocity field given by

$$u(x, y) = +2y(1 - x^2) \tag{A.1}$$

$$v(x, y) = -2y(1 - y^2) \tag{A.2}$$

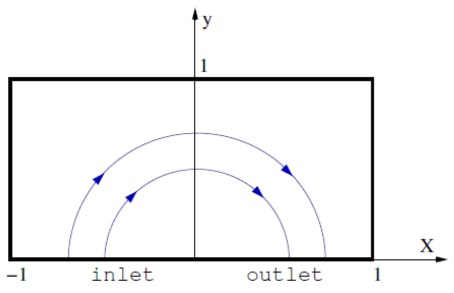
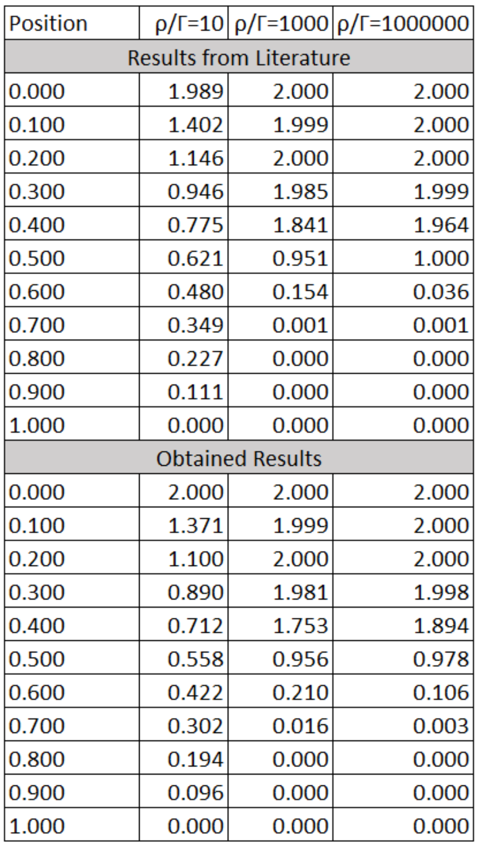


Figure A.1: General schema of the Smith-Hutton problem.

The table of the results obtained from the Smith-Hutton is attached in the Figure A.2. The results from the literature are also listed. Finally, the plots with the results for different mesh sizes are shown in Figures A.3, A.4, and A.5.



Position	ρ/Γ=10	ρ/Γ=1000	ρ/Γ=1000000
Results from Literature			
0.000	1.989	2.000	2.000
0.100	1.402	1.999	2.000
0.200	1.146	2.000	2.000
0.300	0.946	1.985	1.999
0.400	0.775	1.841	1.964
0.500	0.621	0.951	1.000
0.600	0.480	0.154	0.036
0.700	0.349	0.001	0.001
0.800	0.227	0.000	0.000
0.900	0.111	0.000	0.000
1.000	0.000	0.000	0.000
Obtained Results			
0.000	2.000	2.000	2.000
0.100	1.371	1.999	2.000
0.200	1.100	2.000	2.000
0.300	0.890	1.981	1.998
0.400	0.712	1.753	1.894
0.500	0.558	0.956	0.978
0.600	0.422	0.210	0.106
0.700	0.302	0.016	0.003
0.800	0.194	0.000	0.000
0.900	0.096	0.000	0.000
1.000	0.000	0.000	0.000

Figure A.2: Results for the Smith-Hutton problem.

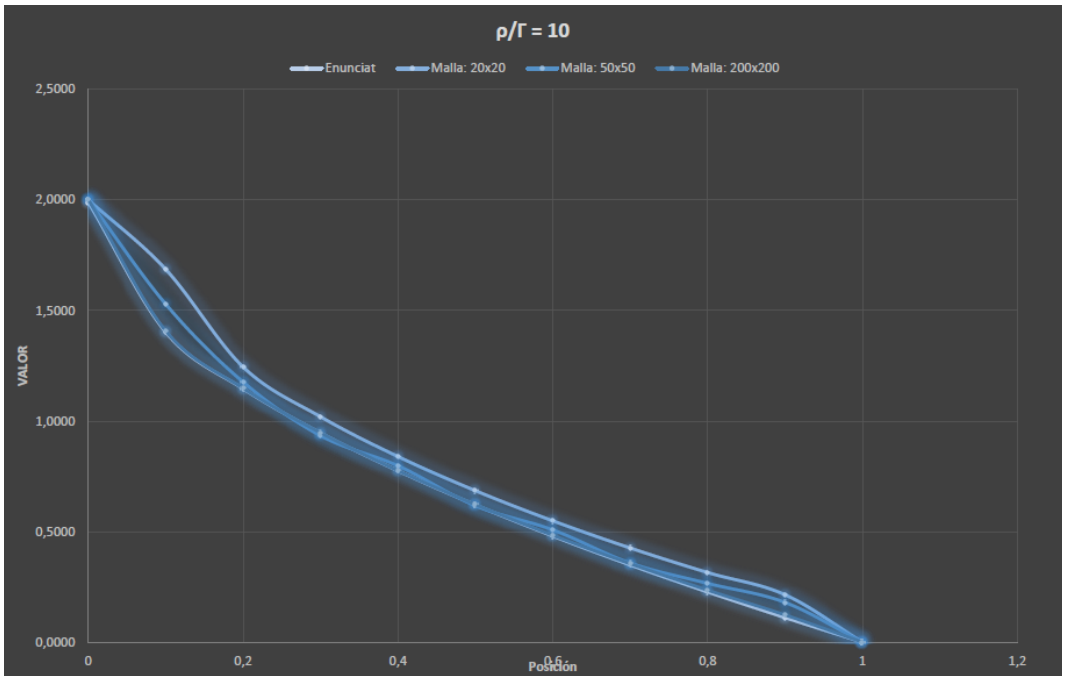


Figure A.3: Smith-Hutton. Plot for $\rho/\Gamma = 10e1$.

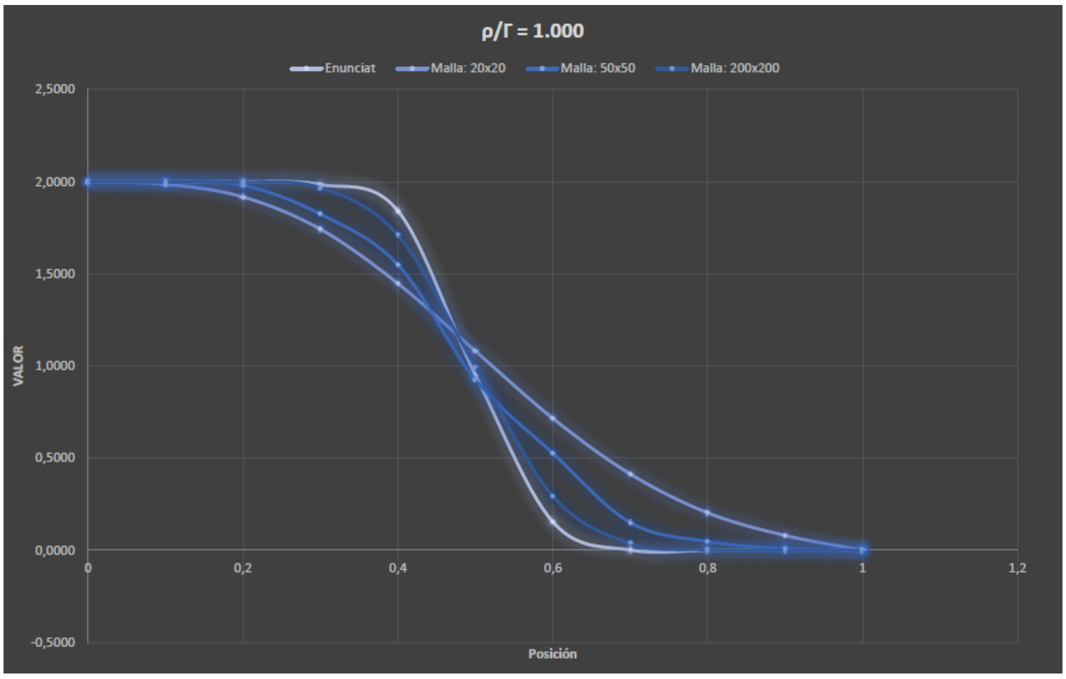


Figure A.4: Smith-Hutton. Plot for $\rho/\Gamma = 10e3$.

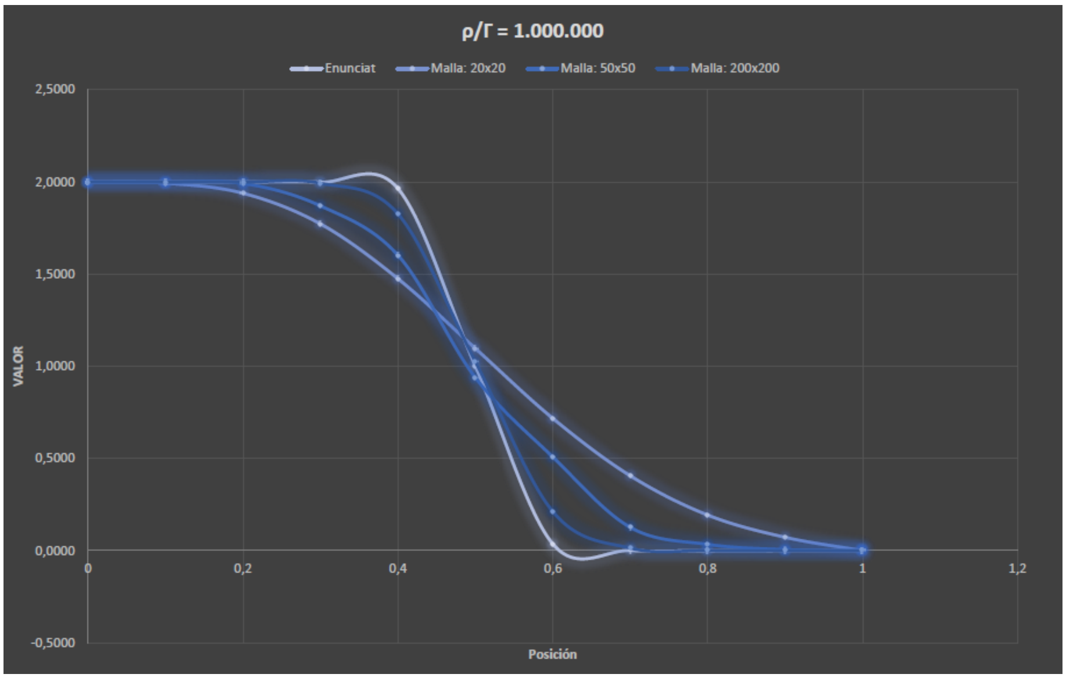


Figure A.5: Smith-Hutton. Plot for $\rho/\Gamma = 10e6$.

A.2 Driven Cavity Case

The objective of this case is to obtain the unsteady solution of the driven cavity problem, described in [28]. To do so, the two-dimensional Navier-Stokes system of equations must be solved numerically in a square domain (see Figure A.6).

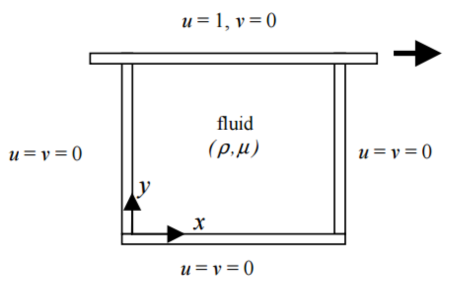


Figure A.6: General schema of the driven cavity problem.

The results have been obtained for different Reynolds numbers: 100, 400, 1,000, 3,200, 7,500 and 10,000. The plots are attached in the Figures A.7, A.8, A.9, A.10, A.11 and A.12. Notice that the green line is the result of reference, found in the literature. The blue line is the result obtained using a 25x25 grid and a CDS numerical scheme. The red line is the result obtained using a 40x40 grid and a QUICK numerical scheme.

In addition, one animation has been made for each Reynolds number using ParaView. The Figure A.13 shows the motion of the fluid for an instant of time for $Re = 10000$.

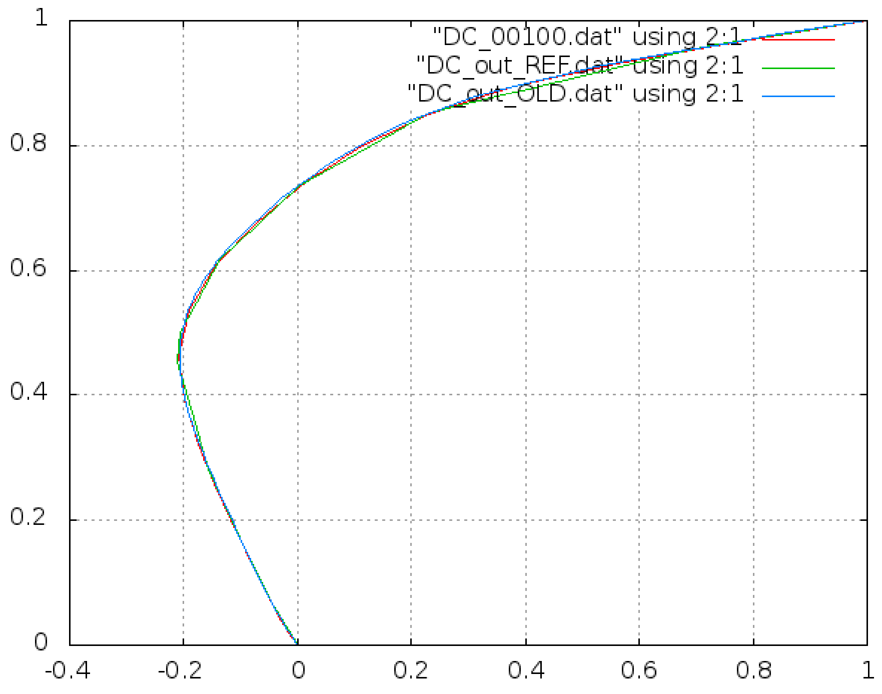


Figure A.7: Driven cavity. Plot for $Re = 100$.

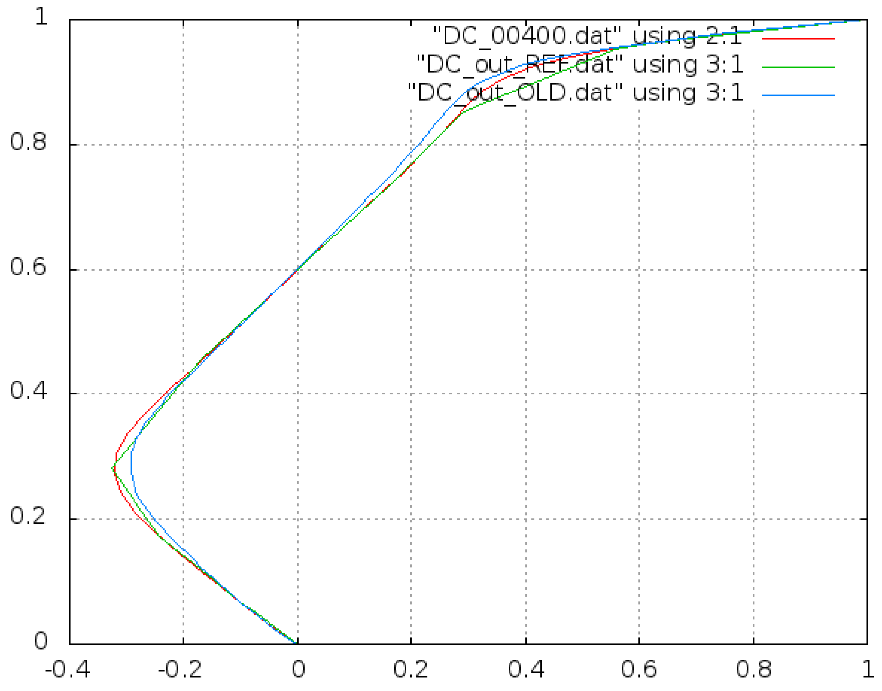


Figure A.8: Driven cavity. Plot for $Re = 400$.

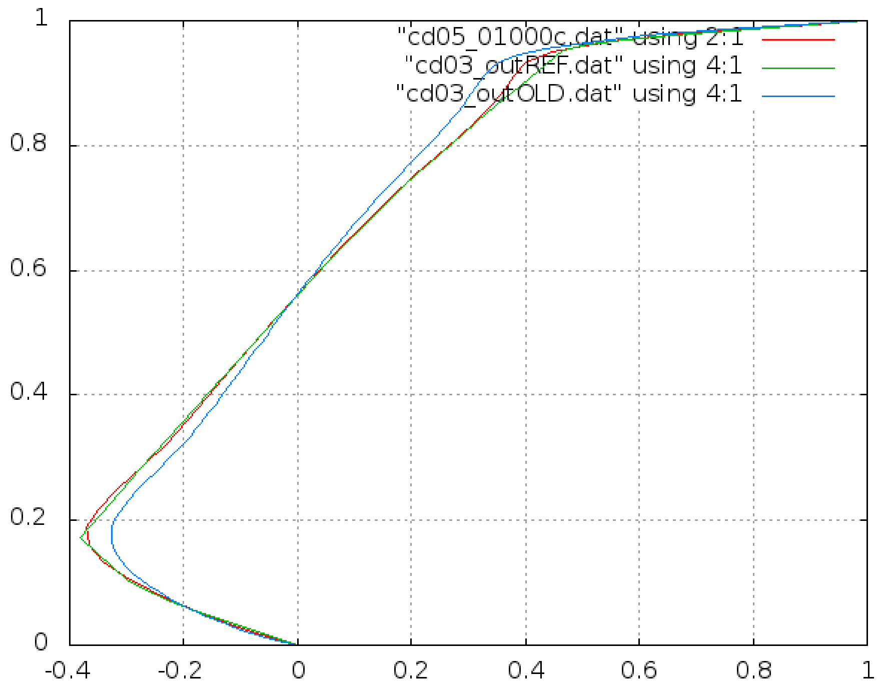


Figure A.9: Driven cavity. Plot for $Re = 1000$.

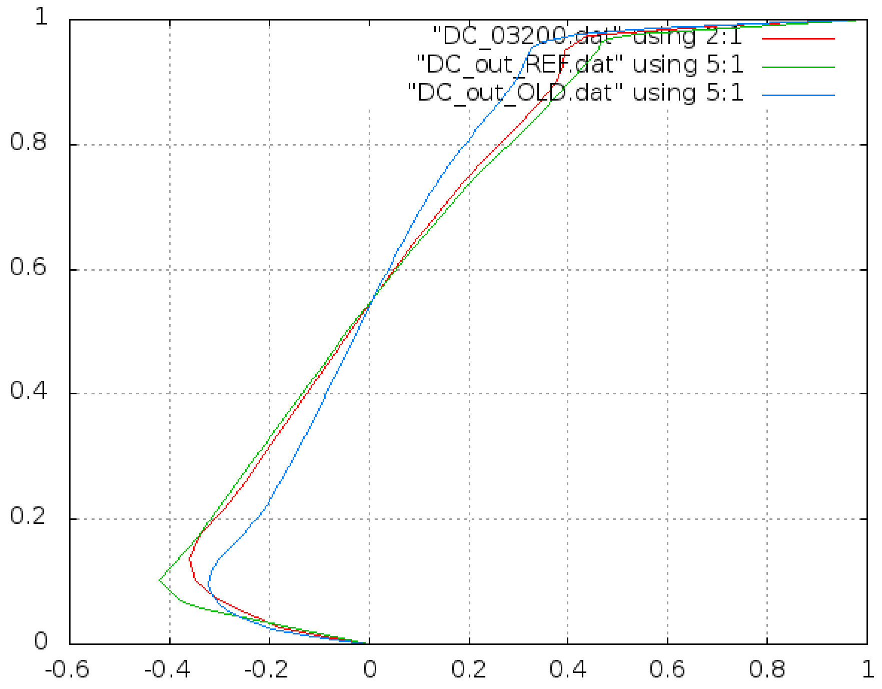


Figure A.10: Driven cavity. Plot for $Re = 3200$.

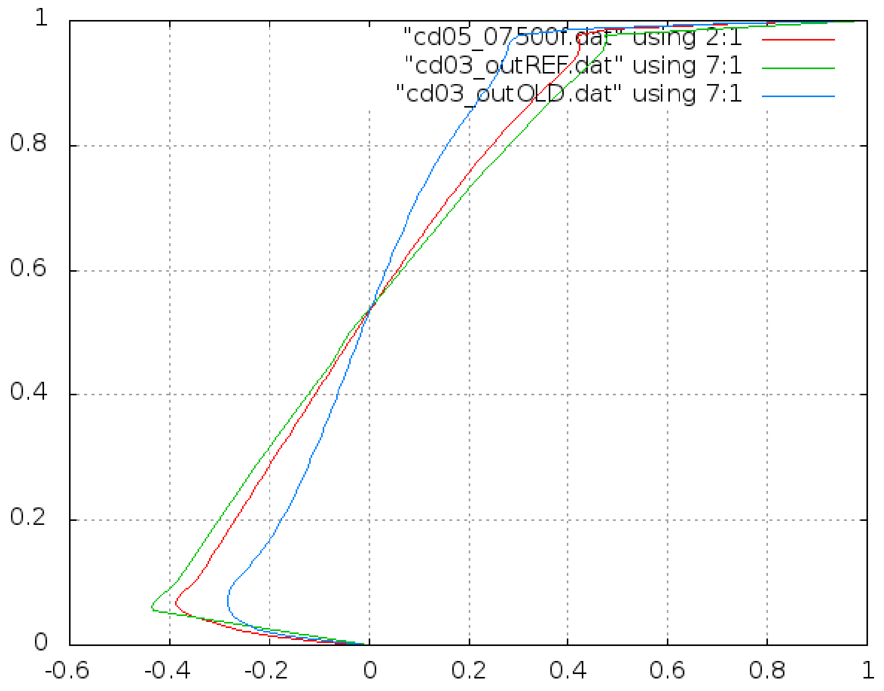


Figure A.11: Driven cavity. Plot for $Re = 7500$.

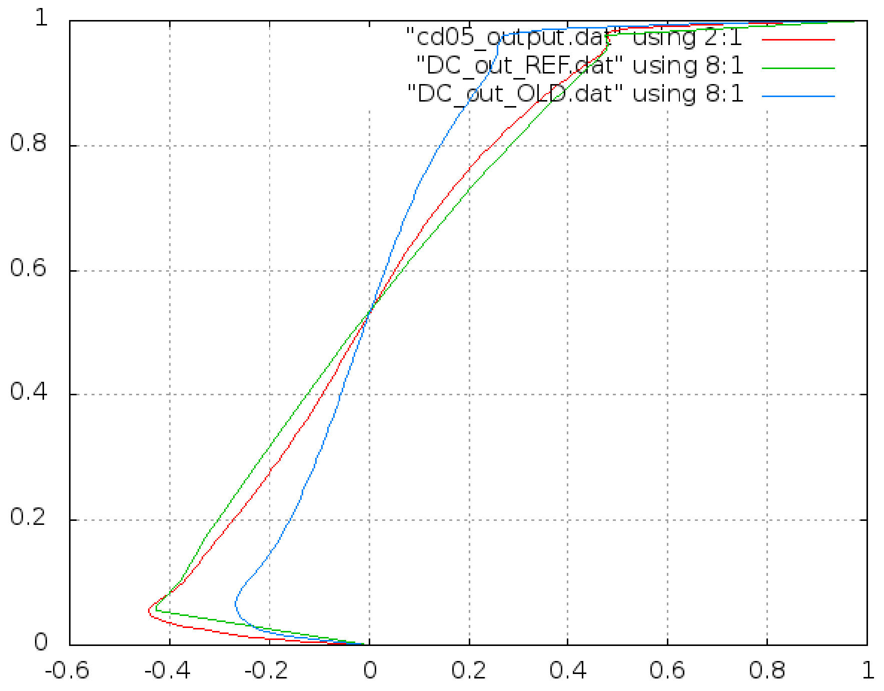


Figure A.12: Driven cavity. Plot for $Re = 10000$.

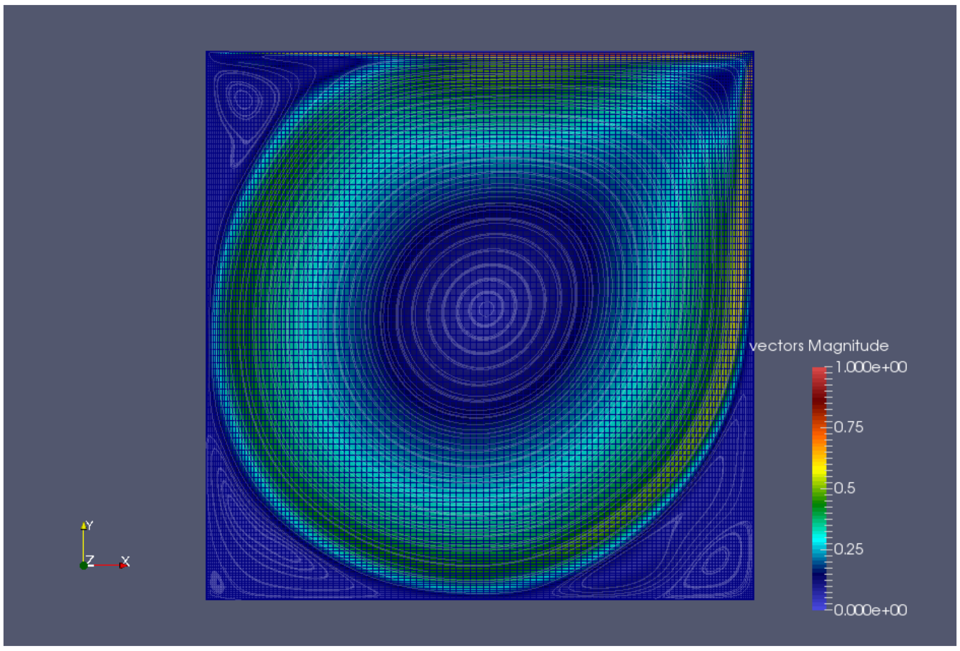


Figure A.13: Driven cavity. Capture in ParaView for $Re = 10000$.

A.3 Differentially Heated Cavity Case

The objective of this case is to obtain the steady state solution of the driven cavity problem, described in [29]. To do so, the two-dimensional Navier-Stokes system of equations must be solved numerically in a square domain (see Figure A.14). In this case, the Boussinesq assumption must be considered in order to simulate the natural convection in the fluid (buoyancy).

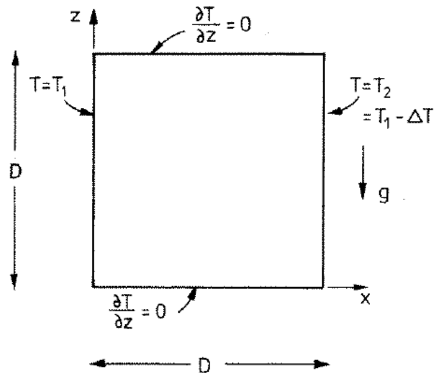

z
$\frac{\partial T}{\partial z} = 0$
$T = T_1$
$T = T_2$
$= T_1 - \Delta T$
D
g
$\frac{\partial T}{\partial z} = 0$
x
D

Figure A.14: General schema of the differentially heated cavity problem.

The results have been obtained for different Rayleigh numbers: 1,000, 10,000, 100,000, and 1,000,000. The plots of average Nusselt number vs number of nodes and relative error (compared to the article of reference) vs number of nodes are attached in the Figure A.15. Notice that the calculations have been carried out using many different numerical schemes.

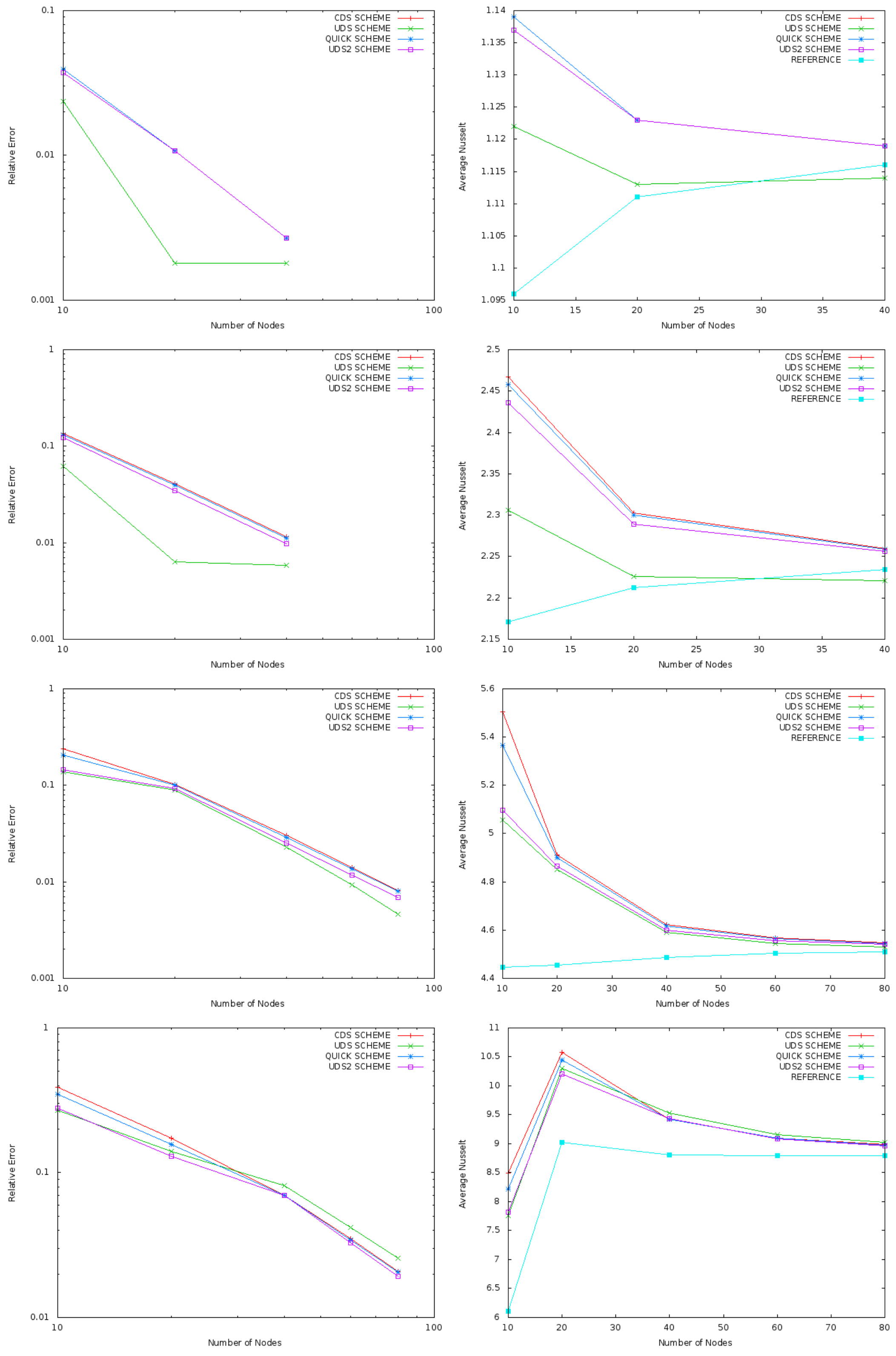


Figure A.15: Differentially heated cavity plots.

Bibliography

[1] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, pages 114–117, April 1965.

[2] The Scipy Community. Dictionary Of Keys based sparse matrix., May 2016.

[3] The Scipy Community. Compressed Sparse Row matrix., May 2016.

[4] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP*. Massachusetts Institute of Technology, 2008.

[5] OpenMP. About OpenMP, June 2012.

[6] A. Silberschatz, P. Baer Galvin, and G. Gaine. *Operating System Concepts*. New Haven, 2013.

[7] OpenMP. OpenMP Application Programming Interface, November 2015.

[8] R. W. Green. OpenMP Loop Scheduling, September 2012.

[9] MPICH. Web pages for MPI and MPE.

[10] Mathematics and Computer Science Division. *MPICH User's Guide.*, 2015.

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard. Version 3.1.*, 06 2015.

[12] W. Kendall. MPI Tutorial.

[13] Bruce R. Munson. *Fundamentals of Fluid Mechanics, 7th.* John Wiley and Sons, 2013.

[14] E. R. G. Eckert and R. M. Drake Jr. *Analysis of Heat and Mass Transfer*. McGraw-Hill, 1972.

[15] W. M. Rohsenow, J. P. Hartnett, and E. N. Ganic. *Handbook of Heat and Mass Transfer Fundamentals*. McGraw-Hill, 1985.

[16] L. Virto. *Mecànica de fluids. Fonaments I.* Edicions UPC, 1993.

[17] F. M. White. *Mecánica de fluidos, 6 ed.* Mc Graw Hill, 2008.

[18] F. X. Trias, M. Soria, A. Oliva, and C. D. Pérez-Segarra. Direct numerical simulations of two- and three-dimensional turbulent natural convection flows in a differentially heated cavity of aspect ratio 4. *Journal of Fluid Mechanics*, 586:259–293, 2007.

[19] University of Kentucky. *Lectures in Computational Fluid Dynamics*, 2007.

[20] R. Courant, K. Friedrichs, and H. Lewy. Uber die differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100:32–74, 1928.

[21] A. J. Chorin. Numerical solution of the navier-stokes equations. *Mathematics of Computation*, 22:745–762, 1968.

[22] F. X. Trias, O. Lehmkuhl, A. Oliva, C.D. Pérez-Segarra, and R.W.C.P. Verstappen. Symmetry-preserving discretization of Navier-Stokes equations on collocated unstructured meshes. *Journal of Computational Physics*, 258:246–267, 2014.

[23] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 01 2016.

[24] W. Petersen and P. Arbenz. *Introduction to Parallel Computing.* Oxford University Press, 2004.

[25] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs. An optimization guide for assembly programmers and compiler makers.* Technical University of Denmark, January 2016.

[26] Barcelona Supercomputing Center. *MareNostrum III User's Guide*, 04 2016.

[27] R. M. Smith and A. G. Hutton. The numerical treatment of advection: a performance comparison of current methods. *Numerical Heat Transfer*, 5:439–461, 1982.

[28] M. Kawaguti. Numerical solution of the navier-stokes equations for the flow in a two-dimensional cavity. *Journal of the Physical Society of Japan*, 16:2307–2315, 1961.

[29] G. de Vahl Davis. Natural convection in a square cavity: a comparison exercise. *International Journal for Numerical Methods in Fluids*, 3:227–248, 1983.