UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Computer Systems and Computation

CPU-GPU parallelization of an ODE solver for the chemical integration of reaction mechanisms of aviation fuels

Master's Thesis

Master's Degree in Cloud and High-Performance Computing

AUTHOR: Moure Sabaté, Álvaro

Tutor: Román Moltó, José Enrique

ACADEMIC YEAR: 2022/2023

Acknowledgements

Me gustaría agradecer a mis tutores, Jose y Daniel, por brindarme la oportunidad de participar en un proyecto tan desafiante como emocionante. Gracias a ti también Anurag, por el apoyo en los momentos de bloqueo y tus consejos cuando más lo necesitaba. Durante todo este periodo he podido aprender un sinfín de tecnologías y conceptos nuevos que han hecho que me termine de enamorar de este ámbito de investigación. Como dice el dicho, "Elige un trabajo que te guste y no tendrás que trabajar ni un día de tu vida", creo que aquí he encontrado eso que andaba buscando.

Diego, gran amigo, qué buenos dos años hemos pasado juntos. Gracias por las llamadas diarias para tomar el café, por las panzadas que me pego contigo y por haber estado a mi lado en todos los momentos complicados. Por supuesto, agradecer a toda mi banda de villanos, Josu, Javi, Ibai, Rafa, Tonet, Dani, Marcos y los imprescindibles pichicas, Calde, Pedro, Gonzi y Carlos, por haber hecho que este último periodo universitario (o no) haya sido inolvidable.

Maria y Pedro, gracias por abrirme la puerta al mundo de la mecánica de fluidos computacional. Sin duda, sois los responsables de que quiera seguir estudiando y aprendiendo el resto de mi vida. Gracias por dejarme robaros horas y horas de vuestro limitado tiempo en ayudarme a aclarar las ideas. No tengo dudas de que de una forma u otra nuestros caminos se volverán a cruzar.

Gerson y Varea, pese a que nuestros caminos se van separando, como no puede ser de otra manera, no puedo estar más agradecido de poder seguir contando con vosotros para todo. A los verdaderos amigos no hace falta verlos todos los días. Sé que se os avecina un futuro brillante, y ahí estaré para veros triunfar.

Marshall, gracias por darme la motivación de levantarme cada mañana para ir a cuidar una de las cosas más importantes que tenemos, el cuerpo. Gracias a ti, me vuelvo a sentir sano, e incluso curado de la espalda. Eres mi referente en lo que disciplina y la constancia se refiere. No sé qué haré sin ti en Barcelona, espero que aparezcas pronto por ahí.

A mi nueva banda, HPFFICE, quién iba a decir que mientras desembalábamos las cajas de la nueva oficina se estaban forjando grandes amistades. Desde luego que lo mejor que me llevo de mi paso por HP sois vosotros.

A ti Paula, a mi compañera de vida, gracias por tu apoyo y tu cariño incondicional. Por enseñarme a ser cada día mejor persona y por aguantarme en los días en los que no lo soy tanto. Por seguir viéndonos cumplir objetivos, lo mejor todavía está por llegar.

Y por último, a mi familia, gracias por alentarme a seguir para adelante, por enseñarme a luchar por lo que quiero, que todo esfuerzo tiene su recompensa. Esto es más vuestro que mío.

Esto no hubiese sido posible sin el apoyo y la compañía de todos vosotros, ¡gracias!

Abstract

In the effort to reduce the carbon footprint generated by air transportation, the exhaustive study of the combustion processes, including traditional, SAF and hydrogen fuels, is necessary to develop advanced methodologies and computational models that can assist the aeronautical industry with the decarbonization objectives of the aviation sector.

In this context, the Ordinary Differential Equations (ODEs) involved in the resolution of chemical kinetics must be solved with an implicit numerical method, known as stiff solver, in order to maximize the time step of the integration interval. This stiff solver includes several numerical methods inside. Typically, a Newton iteration is needed to reach the solution of the nonlinear equation system. Even further, the Newton iteration requires the evaluation of the Jacobian matrix as well as a linear solver execution. All these methods together are usually computationally expensive.

This is where modern computer architectures and Graphic Processor Units (GPUs) can be useful to speed up chemical integration. Multi-threading handling technologies, such as OpenMP and CUDA can be used to parallelize the stiff solver at different levels of granularity. Coarse-grained parallelism, with OpenMP threads, solving chemical integration in multiple mesh points simultaneously. Fine-grained parallelism, launching CUDA kernels to GPU devices, for floating point matrix operations.

In this work an Ordinary Differential Equations integrator library has been developed with the objective of minimizing the integration time. This library exploits the parallel capability of CPU-GPU heterogeneous architectures through shared-memory parallel programming models.

Resumen

En el esfuerzo de reducir la huella de carbono generada por el transporte aéreo, un estudio exhaustivo de los procesos de combustión, tanto para los combustibles tradicionales como para los novedosos combustibles sostenibles SAF e hidrógeno, es necesario para entender y desarrollar nuevas metodologías y modelos computacionales que puedan ayudar a la industria aeronáutica a los objetivos de descarbonización de la aviación.

En este contexto, el sistema de Ecuaciones Diferenciales Ordinarias (EDOs), formado por las reacciones acopladas de los numerosos compuestos que integran el combustible, ha de ser resuelto con un método numérico implícito, conocido en la literatura como "stiff solver", para poder maximizar el paso de integración del intervalo temporal a resolver. Normalmente, para resolver el sistema de ecuaciones no lineal se utiliza el método de Newton o alguna de sus variantes. A su vez, este último método necesita de la evaluación de la matriz Jacobiana, así como, de la resolución de un sistema lineal de ecuaciones algebraicas. La resolución de todos métodos numéricos anidados supone un coste computacional elevado.

Aquí es donde las arquitecturas de computadores modernas y las unidades de procesamiento gráfico (GPUs) pueden aprovecharse para acelerar la integración química. Tecnologías para la programación con sistemas multi-hilo, como OpenMP y CUDA pueden usarse para paralelizar los distintos métodos numéricos a diferentes niveles de granularidad. Paralelización de grano grueso, usando hilos de OpenMP, para resolver la integración química en múltiples puntos del dominio computacional de forma simultánea. Paralelismo de grano fino, invocando "kernels" de CUDA en las GPUs, para realizar operaciones matriciales con aritmética en punto flotante.

En este trabajo se ha desarrollado un librería que integra sistemas de EDOs con el objetivo de minimizar los tiempos de integración. Esta libraría explota las capacidades de paralelización de las arquitecturas heterogéneas CPU-GPU a través de los modelos de programación paralela en memoria compartida.

Resum

En l'esforç de reduir l'empremta de carboni generada pel transport aeri, un estudi exhaustiu dels processos de combustió, tant per als combustibles tradicionals com per als nous combustibles sostenibles SAF i hidrogen, és necessari per a entendre i desenvolupar noves metodologies i models computacionals que puguen ajudar a la indústria aeronàutica als objectius de descarbonització de l'aviació.

En aquest context, el sistema d'Equacions Diferencials Ordinàries (EDOs), format per les reaccions acoblades dels nombrosos compostos que integren el combustible, ha de ser resolt amb un mètode numèric implícit, conegut en la literatura com a "stiff solver", per a poder maximitzar el pas d'integració de l'interval temporal a resoldre. Normalment, per a resoldre el sistema d'equacions no lineal s'utilitza el mètode de Newton o alguna de les seues variants. A la vegada, aquest últim mètode necessita de l'avaluació de la matriu Jacobiana, així com de la resolució d'un sistema lineal d'equacions algebraiques. La resolució de tots els mètodes numèrics anidats suposa un cost computacional elevat.

Ací és on les arquitectures de computadors modernes i les unitats de processament gràfic (GPUs) poden aprofitar-se per a accelerar la integració química. Tecnologies per a la programació amb sistemes multi-fil, com OpenMP i CUDA, poden ser utilitzades per a paral·lelitzar els diferents mètodes numèrics a diferents nivells de granularitat. Paral·lelització de gran gruix, utilitzant fils d'OpenMP, per a resoldre la integració química en múltiples punts del domini computacional de forma simultània. Paral·lelisme de granulació fina, invocant "kernels" de CUDA en les GPUs, per a realitzar operacions matricials amb aritmètica en punt flotant.

En aquest treball s'ha desenvolupat una llibreria que integra sistemes d'EDOs amb l'objectiu de minimitzar els temps d'integració. Aquesta llibreria explota les capacitats de paral·lelització de les arquitectures heterogènies CPU-GPU a través dels models de programació paral·lela en memòria compartida.

Contents

List of Figures

List of Tables

Listings

Acronyms

API Application Programming Interface.

AVX Advanced Vector Extensions.

BDF Backward Differentiation Formula.

BSC Barcelona Supercomputing Center.

CAF Conventional Aviation Fuel.

CFD Computational Fluid Dynamics.

CoEC Center of Excellence in Combustion.

CPU Central Processor Unit.

CSV Comma-Separated Values.

DCM Direct Connection Machine.

DNS Direct Numerical Simulation.

EU European Union.

FIFO First-In, First-Out.

FPGA Field-Programmable Gate Array.

GHG GreenHouse Gases.

GPU Graphic Processor Unit.

HPC High Performance Computing.

IATA International Air Transport Association.

ICAO International Civil Aviation Organization.

IVP Initial-Value Problem.

LES Large Eddy Simulation.

MPI Message Passing Interface.

NSE Navier Stokes Equations.

ODE Ordinary Differential Equation.

OOP Object-Oriented Programming.

PDE Partial Differential Equation.

PRAM Parallel Random Access Machine.

RAM Random Access Memory.

RANS Reynolds Averaged Navier-Stokes.

SAF Sustainable Aviation Fuel.

SIMD Single Instruction Multiple Data.

SIMT Single Instruction Multiple Thread.

SQL Structured Query Language.

TBB Threading Building Blocks.

UML Unified Modeling Language.

UPV Universitat Politècnica de València.

Chapter 1

Introduction

1.1 Motivation

One of today's global challenges is the global warming, that is the temperature increase of the planet, mainly caused by the emission to the atmosphere of GreenHouse Gases (GHG), such as CO2, H2O, CH4, etc. In particular, this is a very hot topic in the transportation sector, whose emission contributions are growing every year, in comparison with industrial and civil sectors that are showing a decreasing trend.

Zooming in the transportation sector, the aviation is presenting a steadily growing evolution over the years, with a prediction of doubling air traffic every 15 years [1]. The reason why this can be a major problem is because nowadays, the way an aircraft is generating the force to move forward and fly is the result of a combustion process that occurs inside its propulsion engines. Traditionally, the fuel that feed those engines is fossil-based. To give the reader a very brief and simple illustration of the chemical composition of an aeroengine exhaust gases, in the Eq.(1.1) can be seen a perfect combustion reaction of methane with oxygen, as the reader can appreciate, the reaction's products are CO2 and H2O.

$$CH_4 + 2O_2 \longrightarrow CO_2 + 2H_2O \tag{1.1}$$

In a real scenario, additional products are generated as a result of an incomplete combustion, hazardous emissions such as NOx, soot, CO, among others, are expelled to the atmosphere. These last molecules have dangerous effects on human lives as well as harmful consequences for the environment.

According to the International Civil Aviation Organization (ICAO), the aviation is responsible for the 2-3% of the total CO2 emitted, and 13% in the transportation sector per year [2]. For many decades, scientists and engineers have been trying to properly understand and model all the complex phenomena that comes with a combustion reaction, with the objective to enhance the current combustion techniques that use Conventional Aviation Fuel (CAF) (mainly kerosene) and develop more environmental-friendly aeroengines. In parallel, other research lines start to make noise as it will be seen in the

following section.

In order to reach the goals of climate change of the Paris Agreement [3], in 2021 a resolution approved by the International Air Transport Association (IATA) set a new horizon of net-zero carbon emissions for the global air transport industry by 2050 [4].

Given the 2050 deadline for the complete decarbonization of the aviation, research and development of several potential net-zero technologies must be accelerated. Two main lines of research for the replacement of aviation's fossil-based fuel are very active currently:

- Sustainable Aviation Fuel (SAF).
- Hydrogen

1.1.1 Sustainable Aviation Fuels and Hydrogen

SAFs are known as fuels that have less GHG and less pollutant emissions when burned. Consequently, they are a good potential replacement of fossil-base fuel in the path to reduce the carbon footprint. Furthermore, the industry has committed to using it as one of the main solutions to decarbonize aviation where other technologies, such as electric propulsion systems, are not competitive for medium and long-range travelling. There are two main different SAFs regarding its way of creation:

- Synthetic fuels, which are chemically produced from hydrocarbon derivatives, feedstocks such as coal and natural gas.
- Bio-renewable fuels, produced from agricultural crops and bio-wastes, using feedstocks like lipids, fats and oils.

The great advantage that SAFs have versus other technologies is that they are referred to as *drop-in* fuels, which means that the way to evaluate the specifications and fuel requirements is the same that CAFs have, i.e, if a SAF is approved it can be directly used as a fuel for a current operational aircraft. There is no need to change the propulsion plant, nor the design of the aircraft, nor infrastructure of the airport, in contrast with technologies such as hydrogen as it will be explained shortly.

Most of the current approved SAFs present lower specific heat than CAFs, so to reach the demands of power that aircraft need, a blending of CAF and SAF is commonly used. Depending on the needs, blending ratios ranges from 10 to 50% [1]. The achievement of 100% SAF will drive to the net-zero carbon goals, as carbon emissions will be neutral because is presumed to be absorbed in the same creation process.

Hydrogen is a non-carbon fuel (see Eq.(1.2)) that has a high specific heat. It has been used by the aerospace industry, as a reducing agent in liquid propellant rockets, for years [5], so it is a tentative option not to reduce, but to eliminate the carbon footprint in the aviation industry. However, to embrace hydrogen as a solution to decarbonize the aviation, many problems have to be solved before make it real. Using hydrogen as a jet fuels comes with a redesign of both aircraft and engines, its low density compared with the kerosene requires huge volumes tanks (even in liquid state) and a reinforced fuselage that can handle the loads generated from these tanks [6].

$$2H_2 + O_2 \longrightarrow 2H_2O \tag{1.2}$$

Even more, the amount of energy both, for generate hydrogen and cryogenic storage, is far from negligible, thus, in order to be considered as carbon-free fuel the energy invested in those process has to come from a clean and renewable sources, such as wind, ocean, solar and so on. Current studies show that operational costs for H2-powered aircraft scenario can increase by 77-112% compared with kerosene-powered aircraft [7], manifesting that still a lot of research focus on H2 and its related infrastructure is needed to prove that H2 can really be an affordable option.

1.1.2 Numerical computation

Scientific computation is one of today's three pillars in all research activities. Before the era of the computer, the traditional way in which scientific advances occurred was through dialogue between theory and experimentation. Experimentation has different limits depending on the area of research, in flow mechanics, set up a laboratory configuration that reproduces the same flow condition that the target phenomena is usually really expensive or impossible, even more, assuming that the experiment is affordable, measuring the flow properties, such as temperature, pressure or velocity is far from trivial. Computational simulations help in the understanding of complex physics phenomena and, complementing laboratory experiments, is a tool for exploring science and technology. However, due to the demanding computation that reactive flow simulations requires, the time needed to get results can be prohibitive.

High Performance Computing (HPC) is a branch of computer science that involves specific software, networking and hardware devices that aims to get the maximum performance of a supercomputer. Flow and combustion simulations relies heavily on it for its development, but, at the same time, its high computational demand is one of the greater precursors of HPC. The programmer that is in charge of coding the computer program that will run the simulations has a key role in order to achieve the maximum performance that a supercomputer can offer. A strong knowledge of parallel computing, software architecture details and the underlying hardware is needed to write efficient code, orders of magnitude of difference in simulation time can be the impact of having an optimized code.

In the Chapter 4 is discussed more in detail the typical parallel architecture of a supercomputer, how this parallelism occurs at different levels, how HPC deals with heterogeneous hardware and the multiple memory models that are present in a parallel computer.

1.2 Objectives

This Master's Thesis is in the frame of the Center of Excellence in Combustion (CoEC), a program funded by the European Union (EU) whose objective is to promote and develop advance simulation software that can contribute to the decarbonization of the energy and transportation sectors [8]. The thesis has taken place in the *Departamento de Sistemas*

Informáticos y Computación (DSIC) of the Universitat Politècnica de València (UPV) in collaboration with the Barcelona Supercomputing Center (BSC).

The purpose of the thesis is to accelerate the computation of the chemical integration stage when solving reacting flow simulations with detailed chemistry. That process gives rise to non-linear Ordinary Differential Equation (ODE) systems that are stiff. Given the scenario, the project can be broken down into the following points:

- Study of the available libraries that are developed to solve stiff ODE systems.
- Configuration and validation of a base case that is computed in a single processor unit (in serial).
- Code analysis to speed up computation using parallelization in a shared memory model.
- Development of a standalone portable library that supports multiple integrators using different levels of parallelism.
- Parametric study of the acceleration achieved using different configurations.

Chapter 2

High-fidelity combustion simulations

2.1 Combustion

Combustion is an exothermic reaction that converts chemical energy contained in fuel molecules into heat that can be used in engines to produce work or thrust. In the literature the reader will find that the flame phenomena, when isolated, can be classified as [9]:

- Premixed flame.
- Non-premixed or diffusion flame.

A premixed flame is found when fresh gases (mixture of fuel and oxidizer) and burnt gases are separated by a reaction zone. On the other hand, a diffusion flame occurs when fuel and oxidizer come from different sources thus they found and mix each other as a result of mass diffusion transport, the reaction happens at the same place of the mixture. Although, an exothermic reaction occurs in both scenarios, the behavior is completely different. On the left side of the Figure 2.1, a pure diffusion flame is seen and gradually from left to right premixed oxygen is added until the stoichiometric mixture is reached, leaving a premixed flame. Not only in the visual, both flames have different flame speed, pollutant formation mechanisms, stability and more. On top of that, turbulence has a strong influence, modifying its behavior, compared with laminar regimes. So, depending on the Reynolds number[1] (see Eq.(2.1)) different combustion regimes are identified.

$$Re = \frac{\rho u L}{\mu} \tag{2.1}$$

In real world combustion applications, such as the ones that occur inside the aeroengines, many of the regimes commented above live simultaneously in the same combustion chamber. Therefore, a proper understanding of all phenomena is needed to advance science towards a cleaner future aviation. From the point of view of numerical modelling, combustion involves complex chemical and physical phenomena that bring challenges to multiple disciplines. The reacting flow that goes through the combustion chamber is characterized for having high temperature, high pressure and high Reynolds number, which indicates

[1] Reynolds number is a dimensionless number that is used to compare the convection forces of the flow with the viscosity.

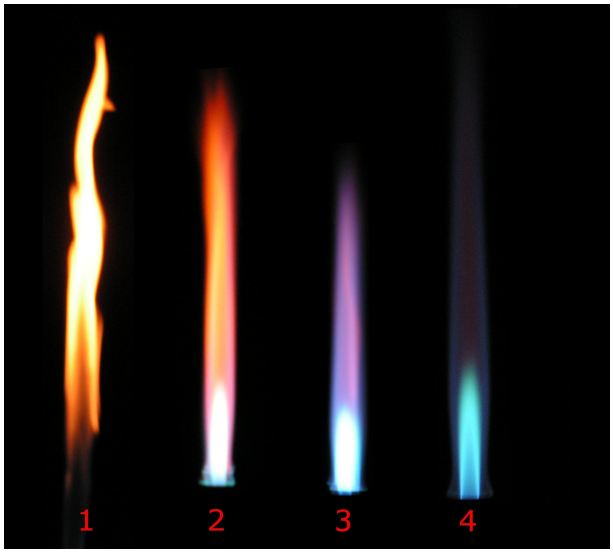


Figure 2.1: Different flame types of a Bunsen burner depend on oxygen supply [10].

that the flow is predominantly turbulent. The injected fuel, usually in liquid state, experiments an atomization phase, which is the process where the liquid jet breaks down into ligaments and droplets, that later on are evaporated and mixed with the surrounding air. In the following section it is discussed what is turbulent combustion modeling and why it is a relevant field of study.

2.1.1 Turbulent combustion modeling

As explained before, a wide range of coupled problems are involved in turbulent flames, then, in order to design new approaches/strategies of combustion, using numerical computation, the following physical and chemical phenomena must be modeled:

- The knowledge of flow properties is crucial to understand the air-fuel mixing process, as well as the heat transfer and mass transport. Computational Fluid Dynamics (CFD) is an extensive research area, which is focused on solving numerically the Navier Stokes Equations (NSE) [11] to determine the thermofluid dynamic state of a given flow.
- Chemical reaction mechanisms predict the fuel oxidation, the heat released and the species that allow to know the combustion products and pollutant. But to achieve that, a precise mechanism that can model all the real chemistry that occurs on the system is needed.
- Multiphase modeling plays an essential role in liquid fuel injection systems where atomization models drive the evaporation rate and air-fuel mixing for combustion.
- Models for radiative heat transfer emitted by some species and radicals. This is even more important when simulating carbon-based fuels due to the radiative heat emission from soot formation.

In order to solve numerically all of these, a high spatial and time resolution of the compu-

tational domain is needed[2]. In CFD, solving all the turbulence scales is known as a Direct Numerical Simulation (DNS). Nowadays, practical combustion applications are computationally prohibitive, even using cutting edge supercomputers. In contrast, an averaging approach of the equations are necessary. Two major techniques are used widely when modeling turbulence:

- Reynolds Averaged Navier-Stokes (RANS). This technique lead to model all the turbulent scales that appear in the simulation.
- Large Eddy Simulation (LES) consists in solving the integral spatial and temporal scales of the turbulence, and using a filter for modeling small scales (the ones that are smaller than the mesh resolution).

When dealing with reacting flows, LES simulations are the most widely used approach due to its trade-off between accuracy and computational cost, as it possesses great properties. Large turbulent structures are very dependent on the geometry of the domain, in contrast, smaller scales present some independence of the domain as they share universal behavior. Thus, in terms of accuracy, modelling only the smaller scales should have less penalty. Furthermore, as large structures are numerically solved, and they predominate the air-fuel mixing process and flame/turbulence interaction, LES seems to be a better approach than RANS in order to achieve high-fidelity combustion simulations. The reader can find more about turbulent combustion modeling in [13].

2.2 Governing equations in reacting flows

Fluid dynamics are based on three fundamental principles:

1. Mass is conserved.
2. Newton's second law, $F = m \cdot a$.
3. Energy is conserved.

These principles lead to the mathematical formulation of the Navier Stokes Equations[3]. In combustion simulations NSE are modified to add the conservation of the species. The continuity and momentum equations are not affected by the chemical reaction, but the effects of mass diffusion must be added to the conservation of energy, additionally, to describe the chemical state of the flow, one transport equation for each species that participate in the chemical mechanism must be added to the system. In summary, for the simulation of reacting flows, a system of Partial Differential Equation (PDE)s composed by the NSE and species conservation must be solved. The formulation of the NSE is outside the scope of the current Master's Thesis, but if the reader is interested, it is recommended to consult Chapter 17 of [14]. However, a special attention will be paid to the transport equation of the species:

$$\frac{\partial \left(\rho Y_k\right)}{\partial t} + \nabla \cdot \left(\rho u Y_k\right) = -\nabla \cdot \left(\rho V_k Y_k\right) + \dot{\omega}_k \ , \tag{2.2}$$

[2]The spatial and temporal scales of the turbulence present in the real reacting flow applications span from the integral length scales to the Kolmogorov microscales [12].

[3]When assuming a Newtonian, unsteady and viscous flow.

where ρ is the flow density, u is the velocity (3 dimensional vector) and Y_k the mass fraction of the kth species. Eq.(2.2) was derived on the basis of an infinitesimally small element fixed in space, which means that it is displayed in its conservation form (see Fig.2.2). The first term of the left-hand side of the equation is the time rate of change of the species mass inside the control volume, i.e., the species mass that appears or disappears during time, for instance, as a result of a chemical reaction. Next to it is the convection term that represent the change of species mass due to mass fluxes entering and leaving the control volume. On the right-hand side, the first term describes the diffusion of species caused by the gradient of species concentration. Finally, $\dot{\omega}_k$, is the chemical source term of the kth species, which is, the target term of the current thesis. The value of this term is the result of the chemical mechanism evaluation involving the kth species and, usually, is the most computational demanding calculation when simulating reacting flows [15].

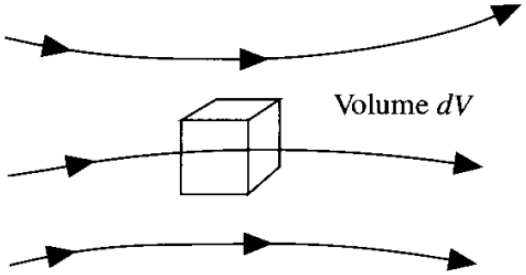


Figure 2.2: Infinitesimal fluid element fixed in space with the fluid moving through it [16].

In the next section, the numerical approach followed to solve the PDE system formed by Eq.(2.2) is described.

2.3 Splitting schemes for chemical integration

In order to solve Eq.(2.2) numerically, a transformation from the partial differential equation to an approximate solution, commonly known as difference equation[4], is needed. On the other hand, when dealing with non-linear initial value problems in three space variables, a first order discretization usually drives to a high mesh resolution (both in time and space) looking to get enough accuracy, or higher order schemes that comes with complicated and expensive differencing, both with a non-negligible computational demand. A popular approach to tackle this problem is using an alternating direction scheme or operator splitting scheme proposed by Strang in [17]. The numerical scheme of Strang, suggest splitting the governing equations into decoupled sub-equations that deals with only a portion of the physics, which means that the terms of the Eq.(2.2) are divided into sub-equations. Strang proved a second order accuracy when taking sufficiently small time steps. With this method, each sub-equation is integrated separately and sequentially in time for each time step.

To demonstrate Strang method, let's assume one-dimensional premixed flame, in an adiabatic and isobaric system. Moreover, taking into account convection, diffusion and reaction processes, the transport equations for the thermochemical state of the flow can be

[4]Difference equation's name comes from the theory of numerical differentiation of derivatives terms.

expressed as follows:

$$\frac{\partial(\rho\phi)}{\partial t} = C(\phi) + D(\phi) + S(\phi) \ , \tag{2.3}$$

where $\phi \equiv \{Y, T\}$ is the thermochemical composition vector and $C(\phi)$, $D(\phi)$, $S(\phi)$ are the convection, diffusion and chemical reaction terms [18]. For simplicity, Eq.(2.3) can be discretized in time using a constant time step:

$$\Delta t = \frac{t_f - t_0}{N} \ , \tag{2.4}$$

where t_0 is the starting calculation time, t_f the final time, and N the number of time steps covered by the calculation. Applying the operator splitting method to the Eq.(2.3), the sub-equations to solve in a Δt are described as:

$$\frac{\partial(\rho^{(1)}\phi^{(1)})}{\partial t} = S(\phi^{(1)}), \quad \phi^{(1)}(x,0) = \phi^n \quad \text{on } [0, \Delta t/2] \tag{2.5}$$

$$\frac{\partial(\rho^{(2)}\phi^{(2)})}{\partial t} = C(\phi^{(2)}) + D(\phi^{(2)}), \quad \phi^{(2)}(x,0) = \phi^{(1)}(x, \Delta t/2) \quad \text{on } [0, \Delta t] \tag{2.6}$$

$$\frac{\partial(\rho^{(3)}\phi^{(3)})}{\partial t} = S(\phi^{(3)}), \quad \phi^{(3)}(x,\Delta t/2) = \phi^{(2)}(x, \Delta t) \quad \text{on } [\Delta t/2, \Delta t] \tag{2.7}$$

where $\phi^n = \phi(x, t_n = t_0 + n\Delta t)$ and ϕ^{n+1} will be the result of the calculation of $\phi^{(3)}(x, \Delta t)$. Eqs.(2.5) and (2.7) correspond to chemical reaction substeps and Eq.(2.6) to a convection-diffusion substep. This sequence is known as $\frac{1}{2}R - T - \frac{1}{2}R$, which is referred as one reaction evaluation over half-time step follow by a transport evaluation (convection + diffusion) and another reaction evaluation over half-time step. Other sequences can be found in the literature, such as $\frac{1}{2}T - R - \frac{1}{2}T$.

The sub-equations concerning this thesis are Eqs.(2.5) and (2.7), which correspond to stiff, non-linear Ordinary Differential Equation system. Due to the large range of time scales involved in the chemical mechanism and considering that the time step is generally dictated by the flow field, which is usually greater than the lowest chemical reaction timescale, a stiff ODE solver is mandatory for accuracy and stability of the solution.

In the next Chapter, a further explanation will be given on how to solve numerically Eq.(2.5) and all the different mathematical problems that arise with it, including a special section dedicated to stiffness.

Chapter 3

The mathematical problem

3.1 Initial Value Problems for Ordinary Differential Equations

The Equation (2.5) presented in the previous chapter is an Initial-Value Problem (IVP), where an ODE or a system of ODEs are given with an initial condition. In most calculus books, the problem is presented as

$$\frac{dy}{dt} = f(t, y), \quad \text{for } a \leq t \leq b, \tag{3.1}$$

with an initial condition $y(a) = \alpha$. When dealing not with a single but with a coupled first-order differential equation system, the formulation is as follows

$$\begin{aligned}
\frac{dy_1}{dt} &= f_1(t, y_1, y_2, \ldots, y_n) \\
\frac{dy_2}{dt} &= f_2(t, y_1, y_2, \ldots, y_n) \\
&\vdots \\
\frac{dy_n}{dt} &= f_n(t, y_1, y_2, \ldots, y_n)
\end{aligned}$$

for $a \leq t \leq b$, given the initial conditions

$$y_1(a) = \alpha_1, \quad y_2(a) = \alpha_2, \quad \ldots, \quad y_n(a) = \alpha_n$$

This last system formulation correspond to the Eq.(2.5) since $\phi = \{T, Y_k\}$, where there is one first-order differential equation for each species and one extra for the temperature. Actually, it is possible to solve all mass fraction species minus one because the last one can be calculated from the mass conservation law, as seen in Eq.(3.2).

$$Y_n = 1 - \sum_{i=1}^{n-1} Y_i \tag{3.2}$$

Eq.(3.1) and the system of ODEs just presented can be treated equally in terms of numerical methods, so in the rest of the chapter, the described techniques may be applied indistinctly to them, as appreciated in the vector notation in Eq.(3.3).

$$y = (y_1, y_2, \ldots, y_n)^T \qquad f(t, y) = (f_1, f_2, \ldots, f_n)^T \tag{3.3}$$

3.1.1 Numerical methods for solving IVP

Eq.(3.1) must satisfy two conditions in order to be a *well-posed problem*:

- $y(t)$ has a unique solution, i.e., it satisfies Lipschitz condition [19].
- Small perturbations in the statement of the problems introduce small changes in the solution. When using numerical methods for solving ODEs, round-off errors are always introduced. Thus, Eq.(3.4), known as the associated *perturbed problem*, is the approximation to Eq.(3.1).

$$\frac{dz}{dt} = f(t, z) + \delta(t), \quad a \leq t \leq b, \qquad z(a) = \alpha + \delta_0, \qquad |z(t) - y(t)| < k\varepsilon \tag{3.4}$$

where $\varepsilon > 0,\ k > 0$ and $|\delta(t)| < \varepsilon$.

In this section, we will present many of the most widespread numerical methods to solve ODEs. Formula derivation for all the methods is outside the frame of this thesis, however, to illustrate the reader with some of the techniques used in more advance methods, the derivation of Euler's Method will be briefly explained. For more detailed explanation of Euler's Method and the other techniques presented in this section, in Chapter 5 of [20] the reader will find an extended explanation of numerical methods for solving ODEs.

Euler's Method

Euler's Method is widely known for its simplicity, and despite being one of the bases for the rest of the methods, it is rarely used in practice due to its low efficiency when solving real engineering problems.

Given a temporal discretization of the Eq.(3.1) with a constant step size, i.e., $h = (b - a)/N = t_{i+1} - t_i$, the value of a function in a point with respect to another nearby point can be described according to Taylor's Theorem

$$y(t_{i+1}) = y(t_i) + (t_{i+1} - t_i)y'(t_i) + \frac{(t_{i+1} - t_i)^2}{2}y''(t_i) + \ldots$$

Assuming a second-order local error, $\mathcal{O}(h^2)$ and $y'(t_i) = f(t_i, y(t_i))$

$$y(t_{i+1}) \approx y(t_i) + hf(t_i, y(t_i)) \tag{3.5}$$

Finally, when using Eq.(3.5) in a computer algorithm with finite precision arithmetic (round-off errors), the expression of Euler's Method is

$$\boxed{y_{i+1} = y_i + hf(t_i, y_i)} \tag{3.6}$$

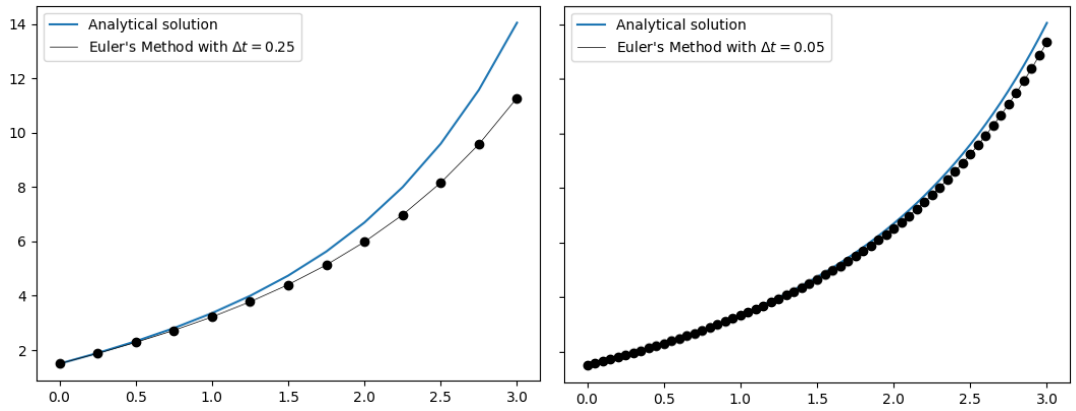
As an illustration of the use of Euler's Method, let's consider the following IVP

$$y'(t) = f(t, y) = y - t, \quad 0 \leq t \leq 3, \quad y(0) = 1.5 \tag{3.7}$$

The analytical result is

$$y(t) = 1 + \frac{1}{2}e^t + t$$

Figure 3.1 shows the solutions of the analytical equation and the numerical Euler's



(a) $h = 0.25$ (b) $h = 0.05$

Figure 3.1: Analytical vs numerical solution of the Eq.(3.7), using different step sizes.

Method, using $h = 0.25$ and $h = 0.05$. It can be appreciated that the error propagation or global error is very dependent of the step size. In order to have high accuracy with this technique the step size should be very small, which is inefficient and computational demanding when integrating large ranges of the independent variable. The remaining of the section will be dedicated to present other methods.

One-step methods

One-step methods use the result of the previous step (y_i) to calculate the solution that follows (y_{i+1}). Among one-step methods, they can be classified as **explicit** or **implicit**. Explicit means that the value to compute, in this case y_{i+1}, is isolated (or can be) in one side of the equation, whereas implicit methods have the variable to compute in both sides of the equation. The latter cannot be directly solved, thus, additional numerical methods are needed, as it will be seen in future sections. Some of the most common one-step methods are

- **Euler's Method**. The just explained method of Eq.(3.6).
- **Higher-Order Taylor Methods**. Just as it has been done with the derivation of Euler's Method, expanding Taylor's series, i.e., using more terms, a greater accuracy (lower local truncation error) is achieved with the cost of an increased computational cost due to the evaluation of higher order derivatives. Eq.(3.8) shows the Taylor method of order n.

$$y_{i+1} = y_i + h\left(f(t_i, y_i) + \frac{h}{2}f'(t_i, y_i) + \cdots + \frac{h^{n-1}}{n!}f^{(n-1)}(t_i, y_i)\right) \tag{3.8}$$

- **Runge-Kutta Methods**. These techniques try to solve the penalty that high-order Taylor Methods have, due to the evaluation of the derivatives of $f(t, y)$, maintaining high-order truncation errors. The most widely used is the **explicit Runge-Kutta Order Four**:

$$\begin{aligned} k_1 &= f(t_i, y_i) \\ k_2 &= f(t_i + \frac{h}{2}, y_i + \frac{1}{2}k_1) \\ k_3 &= f(t_i + \frac{h}{2}, y_i + \frac{1}{2}k_2) \\ k_4 &= f(t_{i+1}, y_i + k_3) \\ y_{i+1} &= y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{3.9}$$

  There are **implicit Runge-Kutta methods** too, with the form

$$\begin{aligned} k_l &= f\left(t_n + c_l h, y_n + h\sum_{j=1}^{s} a_{l,j}k_j\right), \quad l = 1, \ldots, s, \\ y_{n+1} &= y_n + h\sum_{l=1}^{s} b_l k_l \end{aligned} \tag{3.10}$$

  that derive to first-order **Implicit Euler's Method** or second-order **Implicit Trapezoidal method**, depending on the selected order.

Multistep methods

Unlike one-step methods, multistep methods use two or more values computed in previous iterations to calculate the value y_{i+1}. As well as before, there are two types, explicit and implicit methods. The general formula for ***m*-step multistep method** is as follows

$$\begin{aligned} y_{i+1} = \; & a_{m-1}y_i + a_{m-2}y_{i-1} + \cdots + a_0 y_{i+1-m} \\ & + h[b_m f(t_{i+1}, y_{i+1}) + b_{m-1}f(t_i, y_i) \\ & + \cdots + b_0 f(t_{i+1-m}, y_{i+1-m})] \end{aligned} \tag{3.11}$$

When $b_m = 0$ the equation is explicit, otherwise it is implicit. The idea behind these methods is to benefit from previous calculated solutions to have a more accurate calculation of the following one. To do so, $f(t, y(t))$ is approximated by a polynomial $P(t)$ that contains the information of these previous points/solutions (see Eq.(3.12)).

$$y(t_{n+1}) - y(t_n) = \int_{t_n}^{t_{n+1}} f(t, y(t))\, dt \approx \int_{t_n}^{t_{n+1}} P(t)\, dt \tag{3.12}$$

Adams methods use the Lagrangian interpolating polynomial [21] to calculate $b_k,\ k = 0, \ldots, m$ coefficients. Notice that in the Eq.(3.12), coefficients $a_k,\ k = 0, \ldots, m-1$ from Eq.(3.11) are all zero apart from $a_{m-1} = 1$.

Adams-Bashforth explicit methods are constructed taking $b_m = 0$, they range from two-steps to five-steps or more. Eq.(3.13) refers to **Adams-Bashforth Two-Step Explicit Method**.

$$\begin{aligned} y_0 &= \alpha, \quad y_1 = \alpha_1, \\ y_{i+1} &= y_i + \frac{h}{2}[3f(t_i, y_i) - f(t_{i-1}, y_{i-1})] \end{aligned} \tag{3.13}$$

Adams-Moulton implicit methods add $(t_{i+1}, f(t_{i+1}, y(t_{i+1})))$ as an additional point to the interpolating polynomial. As well as Adams-Bashforth methods, in the literature the reader will find Adams-Moulton methods ranging many steps, from two to even twelve. To compare against Two-Step Adams-Bashforth, Eq.(3.14) describes **Adams-Moulton Two-Step Implicit Method**.

$$\begin{aligned} y_0 &= \alpha, \quad y_1 = \alpha_1, \\ y_{i+1} &= y_i + \frac{h}{12}[5f(t_{i+1}, y_{i+1}) + 8f(t_i, y_i) - f(t_{i-1}, y_{i-1})] \end{aligned} \tag{3.14}$$

As the reader may appreciate in Eqs.(3.13) and (3.14), the m first step solutions cannot be calculated with Adams methods. Therefore, the first m solutions are calculated using one-step methods, such as Euler, Runge-Kutta or any of the methods mentioned before.

Error control and variable step size

The methods presented until now have been derived assuming equally-spaced solution nodes (constant step size), however, this approach is inefficient when the problem presents simultaneously intervals of low variations (small derivatives values) and intervals with large variations (larger derivatives values). Small step sizes are needed to capture the latter intervals, but using the same step size for intervals where the function is constant, or its rate of change is very low, is, from the point of view of computation, a waste of time. To solve this efficiency problem, there are adaptive methods that use error control procedures to modify the step size in order to minimize the computational cost. The technique varies a little among one-step and multistep methods, but the idea is basically the same.

Given an error tolerance $\varepsilon > 0$, the objective is to minimize the number of steps while keeping the global error $|y(t_i) - y_i|$ less or equal than ε. For one-step methods, this can be done by using methods of different local order of accuracy[1]. Subtracting local errors of both order methods and doing some mathematical manipulations, the expression to control global error is as follows

$$q \leq \left(\frac{\varepsilon h}{|\tilde{y}_{i+1} - y_{i+1}|} \right)^{1/n}, \tag{3.15}$$

where ε is the tolerance, y_{i+1} the solution of the low order method, $\tilde{y}_{i+1}$ the solution of the method that is used to control the error (one order of accuracy higher than the previous method), h the step-size and n the order of accuracy of the low order method. Eq.(3.15) will drive to two scenarios:

[1]Global error and local truncation error in IVP for ODEs are closely related with the convergence of the solution, for more information read the Section 5.10 of [20].

- When $q < 1$: The error in the approximation is higher than the tolerance, so the solution must be calculated again with a step-size of qh.
- When $q \geq 1$: The computed value is valid and for the next iteration the step-size qh, which is greater than the current h, can be taken.

This error control technique is used for the **Runge-Kutta-Fehlberg Method**. The method uses a calculation of Runge-Kutta of order five to estimate the truncation error of Runge-Kutta of order four.

Similarly, the construction of **variable-step multistep methods** comes from error control procedures. Instead of comparing the error between two different order methods, a **predictor-corrector** strategy is taken. **Adams Variable Step-Size Predictor-Corrector** is a common choice to explain these techniques. It consists in a calculation with the four-step explicit Adams-Bashforth method as an estimation, or *predicted*, solution, y^p_{i+1}, and a second calculation (the corrected solution) with a three-step implicit Adams-Moulton method. This technique transform last implicit method in an explicit method, reducing significantly the complexity and computational cost involved in it. An equation similar to Eq.(3.15) is derived mathematically for this method, therefore, how and when to change the step-size it has been already explained. An important consideration must be mentioned, multistep methods require equal step sizes in the current solution, i.e., the values involved in Eq.(3.14) should be equally-spaced, thus any change in step-size necessitates recalculating the values that take part in the current iteration.

3.2 Stiff ordinary differential equations

The previous section has covered a wide range of methods and techniques involved in the calculation of initial-value problems for ODEs. In the current section it is discussed what stiff ODEs are and why they need a special treatment.

Stiffness is a characteristic that is present in many differential equations that describe real-world phenomena. Stiffness is about efficiency, it depends on the initial value and on the error tolerance imposed. It is common to say that a problem is marked as stiff when explicit methods do not work well. The reason they do not work well is that the integration time step needs to be small enough to be able to capture all the time scales involved in the chemical reactions, which is not very efficiency, otherwise if the time step is not small enough the problem becomes unstable (low convergence rates).

Stiff problems usually appear in chemical reaction systems, where slow and fast reactions coexist (different timescale in each reaction mechanism), thus optimizing the integration of these systems is not an easy task. To illustrate the reader with an example of a chemical reaction system where stiffness is present, let's consider the following reactions

$$\begin{aligned} A &\xrightarrow{0.04} B && \text{(slow)} \\ B + B &\xrightarrow{3\cdot 10^7} C + B && \text{(very fast)} \\ B + C &\xrightarrow{10^4} C + B && \text{(fast)} \end{aligned} \tag{3.16}$$

that leads to the next initial-value problem

$$\begin{aligned} A &: y_1' = -0.04y_1 + 10^4 y_2 y_3 & y_1(0) &= 1 \\ B &: y_2' = 0.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2 & y_2(0) &= 0 \\ C &: y_3' = 3 \cdot 10^7 y_2^2 & y_3(0) &= 0 \end{aligned} \tag{3.17}$$

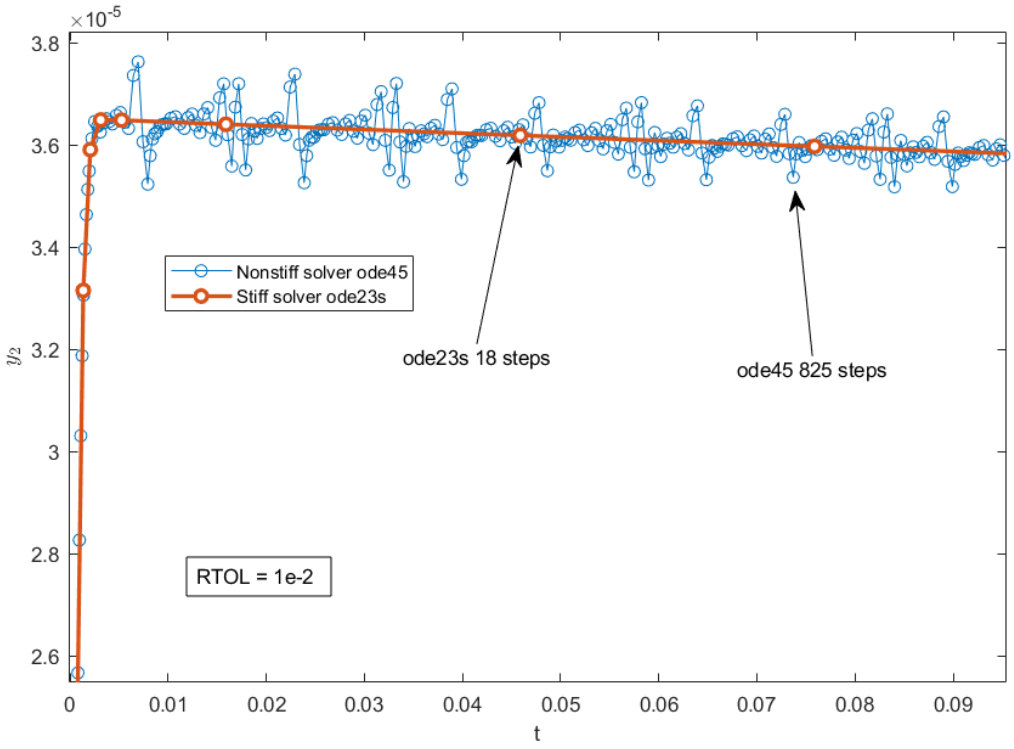


Figure 3.2: y_2 solution of the Eq.(3.17) using both, *ode45* and *ode23s* solvers, for a time range of $0 \leq t \leq 0.1$ seconds with a relative error tolerance of $1 \cdot 10^{-2}$.

The system can be solved (or at least try to) using any of the methods mentioned in the previous section. In Figure 3.2 it is shown the solution of y_2 by using two different solvers that come with the **MATLAB** software package [22]. This example has been reproduced following the explanation of the Chapter 4 of [23].

- **ode45** is a nonstiff explicit Runge-Kutta solver with an error control technique based on the pair RK order four and RK order five, similar to Runge-Kutta-Fehlberg method. In [24] the reader will find a detailed explanation of the method.
- **ode23s** is a stiff implicit solver based on Backward Differentiation Formula (BDF) or similar [25] (BDF will be explained shortly).

As it can be appreciated in Figure 3.2, the difference in the number of steps taken by each method is huge. It is true that the computational cost of a single step of the implicit method is greater than the step computed with the explicit one, but even then, overall,

the time to solution of the explicit method is greater. The oscillations that the *ode45* solver produce in the marching-steps is caused by an instability situation, so to control the error it is obligated to use very small steps. There are techniques to detect when a nonstiff integrator may have encountered stiffness and intelligently change to a stiff solver. In summary, this experiment helps to illustrate that given an initial-value problem with a stiff condition, an explicit method is completely inefficient, therefore, only a certain type of implicit methods can tackle the problem.

Backward Differentiation Formula drive to what is known as **BDF-methods**, which have the form of Eq.(3.18). They are implicit methods that are proven to perform extremely well in stiff problem conditions [26, 27]. BDF are linear multistep methods that approximate the derivative of the function, $f(t_{n+1}, y_{n+1})$, using information from already computed values (in line with what has been explained in multistep methods), see Figure 3.3 for a graphic illustration of the BDF and its derivative.

$$\sum_{k=0}^{s} a_k y_{n-k+1} = h\beta f(t_{n+1}, y_{n+1}) \tag{3.18}$$

In Eq.(3.18), y_{n+1} is the target calculation, a_k and β are coefficients chosen to achieve accuracy of order s.

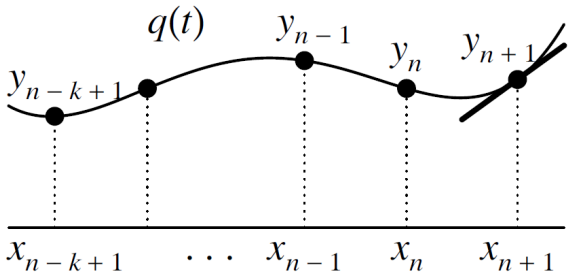


Figure 3.3: Graphic definition of BDF [28].

For an exhaustive explanation of numerical methods for ODEs, stability, stiffness and more, consulting Hairer et al.'s books is highly recommended [23, 28].

BDF methods are the chosen ones for the development of this thesis due to the stiffness present in chemical systems. Therefore, the rest of the chapter focuses on a brief explanation of how to solve an iteration of a BDF method, i.e., what it takes to solve a nonlinear implicit system of equations.

3.3 Numerical solutions of non-linear systems of equations

In the last section, it has been mentioned that implicit BDF methods yield non-linear system of equations. In order to solve it, **root-finding** numerical methods should be applied. **Newton's Method** and its variants are the most common approach for this

situation. Newton's Method is derived from the optimization of a **fixed-point method** with the form of Eq.(3.19).

$$g(x) = x - \phi(x)f(x) \tag{3.19}$$

where the optimal $\phi(x) = 1/f'(x)$ [20]. The nomenclature for systems of non-linear equation is as follows, given

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ \vdots \qquad & \qquad \vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned} \tag{3.20}$$

from the differentiation formula of the BDF method, the n non-linear equations in n unknowns can be represented by

$$\mathbf{F}(x_1, x_2, \ldots, x_n) = (f_1(x_1, x_2, \ldots, x_n), f_2(x_1, x_2, \ldots, x_n), \ldots, f_n(x_1, x_2, \ldots, x_n))^t$$

and using a vector notation, then the system acquires the following form

$$\mathbf{F}(\mathbf{x}) = 0 \tag{3.21}$$

Moreover, Eq.(3.19) in n-dimensional case evolves to the expression:

$$\mathbf{G}(\mathbf{x}) = \mathbf{x} - \mathbf{J}(\mathbf{x})^{-1}\mathbf{F}(\mathbf{x}) \tag{3.22}$$

where $\mathbf{J}(\mathbf{x})$ is the **Jacobian** matrix, defined by

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}) & \ldots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}) & \ldots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(\mathbf{x}) & \frac{\partial f_n}{\partial x_2}(\mathbf{x}) & \ldots & \frac{\partial f_n}{\partial x_n}(\mathbf{x}) \end{bmatrix} \tag{3.23}$$

Finally, **Newton's method for non-linear system** is described by Eq.(3.24), where superscript k represents the number of the iteration (remember that this is an iterative numerical method).

$$\boxed{\mathbf{x}^{(k)} = \mathbf{G}\left(\mathbf{x}^{(k-1)}\right) = \mathbf{x}^{(k-1)} - \mathbf{J}\left(\mathbf{x}^{(k-1)}\right)^{-1}\mathbf{F}\left(\mathbf{x}^{(k-1)}\right)} \tag{3.24}$$

Two common approaches exists to construct the Jacobian matrix,

- **Analytical evaluation**. It consists in manually programming the derivatives of each of the functions of the system for all the variables of the $\mathbf{x}$ vector. Then, the Jacobian construction in the iteration k, $\mathbf{J}(\mathbf{x})$, requires n^2 function evaluations.
- **Finite Differences**. This approach consists in approximating the Jacobian matrix by approximating the partial derivatives as shown in Eq.(3.25), where h is the step (a small value) and e_i is a vector of all zeros apart from a 1 in the i position.

$$\frac{\partial f_j}{\partial \mathbf{x}_i^{(k)}} \approx \frac{f_j(\mathbf{x}^{(k)} + e_i h) - f_j(\mathbf{x}^{(k)})}{h} \tag{3.25}$$

This method is expected to have quadratic convergence when a sufficiently accurate initial vector value is given. Hence, the major weakness of the method is that it needs a good first approximation and this is not always possible, so it is often common to start the root-finding problem with another method, such as **Steepest Descent Method**, that have less rate of converge, but helps to reach a sufficiently good start point before moving to Newton's Method [20]. Furthermore, the computational cost of evaluating the Jacobian matrix at each iteration is really high, a problem that has motivated the development of similar methods, also known as **Modified Newton's methods**, that seek to reduce the computation needed for each iteration while trying to keep the rate of convergence the higher possible[2]. Some examples of these methods are:

- **Chord method**. Here the Jacobian matrix is calculated only one time, at the beginning of the iteration process. The concept behind this method is to take the direction of where the root is supposed to be and keep that direction for the rest of the algorithm.
- **Shamanskii's method**. This method is less radical than Chord method, it construct the Jacobian matrix each m iterations or when the convergence ratio slow down a certain threshold.
- **Broyden's method**. It lies within the **Quasi-Newton methods**, that are the generalization of the **Secant method** to non-linear systems of equations. Instead of calculating the Jacobian matrix at each iteration, this method has been developed to construct an initial matrix (usually using finite difference) and, since then, only update the matrix by adding a relative cheap calculation.

In the current section it has been presented Newton's method and it has been briefly described some of its derivations. However, in Eq.(3.24), the calculation of the inverse of the Jacobian matrix is an expensive operation, and actually, it is not really need it. In the next section it is explained how this inverse matrix is avoided and, therefore, how to proceed in the Newton's iteration.

3.4 Direct methods for solving dense linear systems

Newton's method's second term, $-\mathbf{J}(\mathbf{x})^{-1} \cdot \mathbf{F}(\mathbf{x})$, is a product between a matrix (inverse the of Jacobian) and a vector (the evaluation of the system equations in $\mathbf{x}$), whose result is a vector of the same length as $\mathbf{F}(\mathbf{x})$. So, to avoid the inverse calculation, the matrix-vector product can be addressed as a linear system described as follows

$$\mathbf{y} = -\mathbf{J}(\mathbf{x})^{-1} \cdot \mathbf{F}(\mathbf{x}) \longrightarrow \mathbf{J}(\mathbf{x}) \cdot \mathbf{y} = -\mathbf{F}(\mathbf{x})$$

and

$$\mathbf{J}(\mathbf{x}) \cdot \mathbf{y} = -\mathbf{F}(\mathbf{x}) \longrightarrow \boxed{\mathbf{A} \cdot \mathbf{x} = \mathbf{b}} \tag{3.26}$$

In Eq.(3.26), $\mathbf{A}$ is a $n \times n$ matrix (Jacobian is a square matrix) and the length of $\mathbf{x}$ and $\mathbf{b}$ is n too. The size of the system corresponds to the number of species in the combustion

[2] A quadratic convergence is only achieved with Newton's Method, but some variations of it achieve superlinear ratios while reducing notably the computational cost.

mechanism at a given coordinate of the computational domain (refer again to the Section 2.3 of the Chapter 2 if needed).

The matrix **A**, that is the Jacobian construction at some point of the algorithm (Newton's method or its modifications), is usually a dense matrix, which means that almost every coefficient, if not all, has a nonzero value. In order to solve this square, dense and linear system, the best suitable approach is the **LU factorization of A with Backward and Forward Substitutions**. Hence, the rest of the section is focused on explaining that procedure.

3.4.1 LU matrix factorization

The LU matrix factorization is a decomposition of the linear system's matrix, such as $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$, where **L** is a unit lower triangular matrix and **U** is an upper triangular matrix. This factorization is a derivation of the traditional **Gaussian elimination**. The advantage of using the LU factorization is that the system of Eq.(3.26) can be easily solved in two-steps[3]. A **Backward Substitution** for the lower triangular system of the Eq.(3.28) and a **Forward Substitution** for the upper triangular system of the Eq.(3.29).

$$\mathbf{L} \cdot \mathbf{U} \cdot \mathbf{x} = \mathbf{b} \tag{3.27}$$

$$\mathbf{L} \cdot \mathbf{z} = \mathbf{b} \tag{3.28}$$

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{z} \tag{3.29}$$

The advantage of using modified Newton's methods which do not evaluate the Jacobian matrix at each iteration, such as the Chord or Shamanskii's method mentioned before, is that the LU factorization remains valid until the Jacobian is changed. This strategy reduce considerably the computational cost of the non-linear method. In Chapter 6 of [20] it is demonstrated that solving Eqs.(3.28) and (3.29) requires $O(n^2)$ arithmetic operations whereas solving with Gaussian elimination takes $O(n^3)$. Thus, as long as the LU factorization, who requires $O(n^3)$ operations to be constructed, exploits or maximizes the number of iterations for the non-linear method, a speed-up will be appreciated. Just to mention that LU factorization will fail if a zero value is present in the diagonal of the matrix, or severe round-off error propagation may be encountered when the matrix is not **strictly diagonally dominant**. To avoid these scenarios, **pivoting** techniques are used by robust linear-solver software packages. For instance, MATLAB linear solver use **partial pivoting**, a technique that uses a permutation matrix for keeping track of the changes of the original matrix, as shown in Eq.(3.30).

$$\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{x} = \boxed{\mathbf{L} \cdot \mathbf{U} \cdot \mathbf{x} = \mathbf{P} \cdot \mathbf{b}} \tag{3.30}$$

[3]Maybe, the more intuitive form is to solve **x** by calculating the inverse of **A**, but this approach, apart from turning back to the start point of the current discussion, is completely inefficient.

Chapter 4

High Performance Computing in engineering simulations

4.1 Supercomputers and its hierarchical parallelism

In Chapter 2 it has been described the fundamental bases of numerical combustion simulations and how splitting schemes facilitate the calculation of reacting flows. The evaluation of the species' source term derive to an ODE system that involves the selected chemical mechanism of the combustion process. The ODE system has been numerically analyzed in Chapter 3, where it has been explained numerous methods for its integration, paying special attention to stiffness. Moreover, the methods that work well in stiff initial-value problems involve the calculation of non-linear and linear systems of equations. The main numerical methods that give solution to latter systems have been briefly explained too. Given a computational domain discretized in hundreds of thousands or millions of nodes, and having to solve ODE systems at each of those nodes, the amount of computation required by all these processes is infeasible without parallel supercomputers.

A parallel supercomputer is a set of inter-connected computing nodes[1] that communicate with each other through a high-speed network. Therefore, in combustion simulations, the computational domain is distributed between multiples nodes (as many as needed) that must communicate via the network when accessing data that resides in other node's memory. Each node has its own private memory that is accessed by the multiple processing units that are found inside. This hierarchy drives to distinguish the concepts of **distributed and shared-memory parallelism**. Figure 4.1 illustrates the classical hierarchy of modern supercomputers. It is made of multiple racks that are composed by nodes, and each node can contain several Central Processor Units (CPUs) and accelerators (usually Graphic Processor Units (GPUs)). Finally, each processor unit is composed by multiple cores, that are the minimum processor units that perform the computational operations.

To take the most of the hierarchical parallelism of supercomputers, the software that runs on it has to be able to exploit all the computational capability, a responsibility that

[1]Notice that here, *nodes*, refers to a unit of the supercomputer not a discretized point of the mesh.

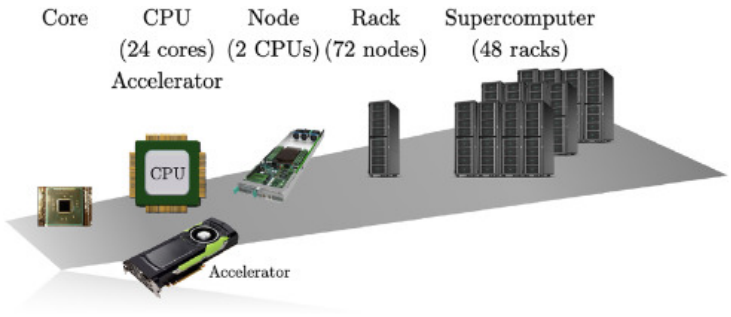


Figure 4.1: Hierarchy of a parallel supercomputer. The numbers in parentheses refer to the configuration of Marenostrum 4 supercomputer, hosted at Barcelona Supercomputing Center [15].

lies on the programmer's side. This is the reason why the programmer should know the underlying heterogeneous hardware architectures, together with the set of HPC tools and techniques, to write efficient parallel programs.

This chapter describes the concepts of distributed-memory, shared-memory and other types of parallelisms that live in a supercomputer, with a special focus on the programming languages or Application Programming Interfaces (APIs) that are used at intra-node levels, due to the fact that chemical evaluation, that has been explained in the previous chapters, is currently calculated at a node level. This is because the computational domain has already been distributed, consequently, the ODE systems at each mesh point does not need external communication to proceed.

4.2 Distributed-memory parallelism

Distributed-memory parallelism, also known in the literature as **inter-node parallelism** [29], is based on the Direct Connection Machine (DCM) model [30]. This model consists in multiple processes, with their own private memory, running all together the same program. They do not share memory, thus the information or data exchange is performed through message passing (see Figure 4.2). The basics primitives in these systems are

- **Send**(v, i), which is writing the information v in the processor's buffer P_i.
- **Receive**(v), which is reading the information v and cleaning the buffer, saving v's information in a local variable.

Although this model can be applied at node-level (spawning multiples processes in a single CPU), it is designed for multiple nodes who are connected by a network. In engineering simulations, the computational domain is partitioned into blocks that are distributed between nodes. Then the computation is carried out independently by each node. Eventually, the nodes should synchronize to exchange the data that is on the edge of the partitioned subdomains. However, message passing usually stops the computation, which is a penalty in the performance. Thus, in order to develop efficient programs in distributed-memory systems, the following principle guides should be considered [29]:

- **Balanced computational load**. If the computational load is not well distributed

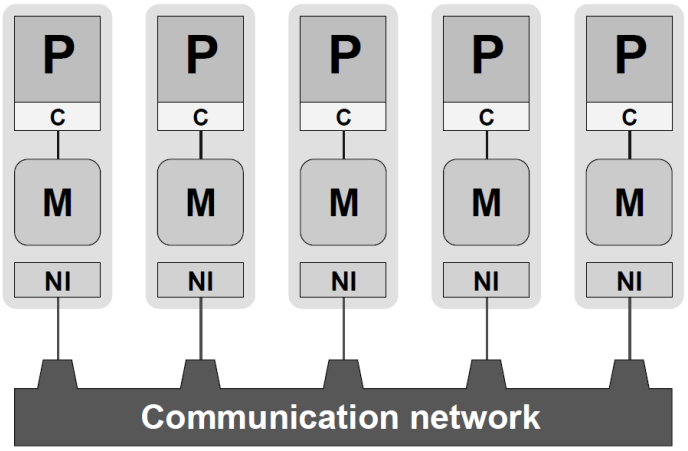


Figure 4.2: Simplified scheme of the DCM model. Separate processes (P), communicate via network interfaces (NI). No process can access another process' memory (M), but each processor have its own private memory [31].

among the nodes, the progress of the overall problem would be delayed by the node who has the larger load.

- **Maximize computation-to-communication ratio**. The problem distribution should prioritize computational load, instead of delivering subdomains that have great amounts of communication, which would delay the computation of every other node.
- **Minimize global communication**. Global communication delays the computation of all nodes simultaneously, parallel algorithms should minimize or eliminate, when possible, this type of communication.

The programming model that follows the guidelines of the DCM theoretical model is Message Passing Interface (MPI). It is a standard API used worldwide for scientists, research institutes and private companies. In concrete implementations, such as **Open MPI** [32], the main actions are found as library functions or subrutines in the C and Fortran programming languages. The current version of the MPI specification is 4.0, a document with the details can be found in [33].

4.3 Intra-node parallelism

In contrast with distributed-memory parallelism, at a node-level, parallelism is constantly changing and evolving very quickly over the years. Hence, legacy software is rapidly outdated in terms of optimizing the available parallelism of the machine. It is rather simplistic to categorize all the optimizations and parallelism that occurs in a node as shared-memory parallelism. In a typical supercomputer's node a complex memory hierarchy is present, it is usually composed by the main memory or Random Access Memory (RAM), cache memories and processor registers. Furthermore, nowadays, it would be rare for heterogeneous devices, such as GPUs or Field-Programmable Gate Arrays (FPGAs), not to be present on a supercomputer's node. Usually, in shared-memory environments, applicable

both to CPUs or GPUs memory hierarchies, the performance of a compute-bound algorithm highly depends on the optimal utilization of the memory hierarchy, thus a couple of notions will be given before addressing multiprocessing techniques.

Scientific and engineering code applications work with large n-dimensional matrices. However, from the point of view of the code, the matrices are stored in contiguous memory, treating them as one-dimensional vectors. Depending on how the matrices are stretched out, they can be classified as **column-major order** or **row-major order**. This can be a restriction of the programming language, for instance, Fortran's matrix storage strategy is always colum-major order, while in C/C++ is row major. Figure 4.3 illustrates a three-dimensional matrix in row-major order.

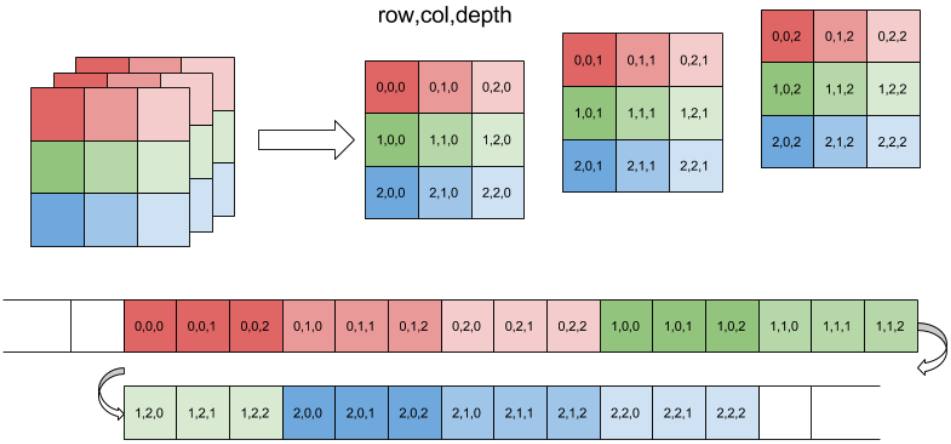


Figure 4.3: A scheme of a three-dimensional matrix stretched out in contiguous memory in row-major order.

These matrices are stored in the main memory where the access to its elements is very slow, compared to the time of accessing caches or registers data (the processor has to wait for the element to continue its computation, an undesirable situation). But, when a matrix element is requested from the RAM, it is retrieved and stored in cache memories along with nearby elements. So, designing a program that can exploit that scenario, specially in memory bound applications, is extremely beneficial in terms of performance.

4.3.1 Cache utilization

Cache utilization is not only a concern in parallel computing but also in serial code. As described before, the optimal utilization of cache memories can play a key role in the performance of a program. Particularly, CFD's codes suffer from low computation intensity, i.e., low ratio of arithmetic operations per memory elements accessed per operation, making cache utilization programming techniques an essential design step for optimizing programs. To show in some detail what these techniques consist of, let's consider an example of an outer product of vectors a and b, of lengths n and m, resulting in an $n \times m$ matrix C, stored in column-major order (see Listing 4.1). At some point of the execution the elements of innermost-`i`-loop, i.e., all the elements of the a vector, will be fetched from the RAM to cache. However, if the length n is too large, the cache will not have

enough space to hold all the elements of a at once, resulting in multiple *cache misses*. It should be considered that apart from the elements of a, cache memory will be holding some elements from b and C. Therefore, the optimum scenario would be that the cache and registers could hold all the elements of a, the entire `j`-column of C and the current value of $b(j)$ in order to maximize the computation intensity of the innermost loop.

```
for (int j = 0; j < m; j++)
{
  for (int i = 0; i < n; i++)
  {
    C[i + j * n] = a[i] * b[j];
  }
}
```

Listing 4.1: Outer product without any optimization.

An optimization is shown in Listing 4.2. Here, the innermost-`i`-loop has been changed for a third `jj`-loop that iterates blocks of vector b of size B, a concept known as **cache blocking**. This leads to better cache reuse, where the block B is chosen to fit entirely in cache. But a better optimization is presented in Listing 4.3, where the innermost loop is changed to minimize *cache write misses* of C `j`-column matrix. Notice that Listing 4.2 is just as optimal as Listing 4.3 if C matrix is stored in row-major order.

```
for (int j = 0; j < m; j = j + B)
{
  for (int i = 0; i < n; i++)
  {
    for (int jj = j; jj < (j + B - 1); jj++)
    {
      C[i + jj * n] = a[i] * b[jj];
    }
  }
}
```

Listing 4.2: Outer product with cache blocking optimization for row-major ordering.

```
for (int i = 0; i < n; i = i + b)
{
  for (int j = 0; j < m; j++)
  {
    for (int ii = i; ii < (i + b - 1); ii++)
    {
      C[ii + j * n] = a[ii] * b[j];
    }
  }
}
```

Listing 4.3: Outer product with cache blocking optimization for colum-major ordering.

4.3.2 Vectorization

Another concept that concerns to both, serial and parallel code, is **vectorization**. Vectorization is the ability of modern processors to execute arithmetical operations to multiple data simultaneously. It is also known as Single Instruction Multiple Data (SIMD), that comes from Flynn's taxonomy of computer architectures [34]. It consists in data parallelization at processor unit/core level. Consider the vector addition code of Listing 4.4, the code is just a single loop that iterates through all the elements of a and b, adding its values and writing the result in c.

```
for (int i = 0; i < n; i++)
{
  c[i] = a[i] + b[i];
}
```

Listing 4.4: Vector addition operation.

Vector addition can be sped up using the intrinsic parallelization in modern processor architectures. In Listing 4.5 it is shown the explicit use of vectorization using the Advanced Vector Extensions (AVX) set of instructions [35]. Figure 4.4 may help the reader to understand graphically what is Listing 4.5's code doing.

```
__m256 *ptr1, *ptr2, *ptr3;
for (int i = 0; i < n; i+=4)
{
  ptr1 = (__m256*)&a[i];
  ptr2 = (__m256*)&b[i];
  ptr3 = (__m256*)&c[i];
  *ptr3 = __mm256_add_ps(*ptr1, *ptr2);
}
```

Listing 4.5: Vector addition operation with explicit AVX instructions.

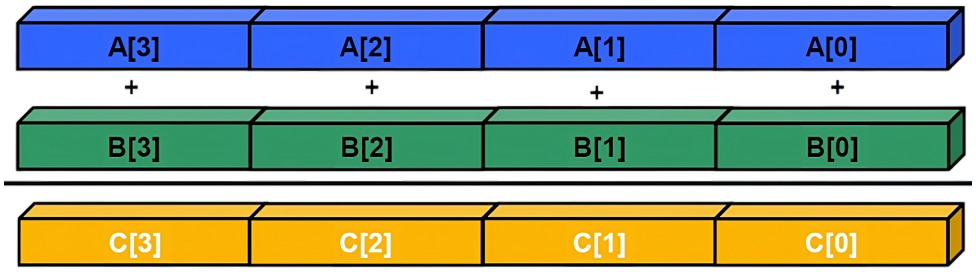


Figure 4.4: Graphic illustration of vectorization applied to a vector addition operation.

Vectorization is usually handled by the compiler at compile time, releasing the programmer of such a tough task. However, sometimes the compiler is not intelligent enough to identify blocks of code that can be vectorized. This could happen with the presence of control flow instructions, such as `while` and `if`, or with the presence of function calls inside `for` loops. So, the programmer should indentify vectorization opportunities, and try to write clean `for` loops, easy to vectorize, to the compiler, when possible.

4.3.3 Shared-memory parallel programming model with OpenMP

The most popular shared-memory parallel theoretical model is presented in [36], known as Parallel Random Access Machine (PRAM). It explains a scenario where multiple processors run the same code (where the code flow can take different paths depending on the processor's ID), having simultaneous access to a shared memory (there are not cache memories in this model) and where they communicate through variables, i.e., reading and writing variables from/to the shared memory. From this theoretical model, multiple shared-memory parallel programming models have been developed. In the 1990's the **POSIX Threads API** [37], for the C language, to run multiple threads in parallel or concurrently, was extremely popular. Nowadays, the reader will find multiple solutions, such as the **Intel** Threading Building Blocks (TBB) C++ library [38]. But there is no doubt that the dominant API among the scientific community is **OpenMP**, moreover, it has become a shared-memory parallel programming standard [39]. OpenMP API is mainly supported in Fortran, C and C++ languages. This section covers the basics of OpenMP programming model with some examples.

OpenMP is based on a set of compiler directives that follows a *fork-join* scheme, as illustrated in Figure 4.5b. Two actions are identified in the scheme:

- **Fork**. It spawns a new task, can be a process or a thread, who starts running from the same point where the *master*, is at the moment of invoking it.

- **Join**. An instruction that waits until one or more concurrent tasks finish the work.

OpenMP implements the *fork-join* model using threads instead of processes because the former fits better the shared-memory model, some of their properties are:

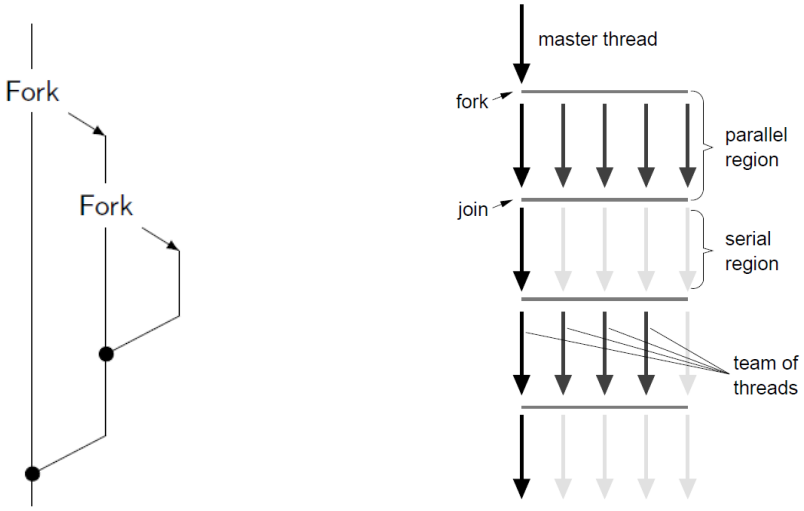
- Threads are spawned inside the same process, so they share resources and memory. However, each thread has its own execution context and a reduced private memory space.

- The overhead of spawning new threads differs by orders of magnitude with respect processes, with the former being faster.

- Synchronization is faster.

To show some OpenMP compiler directives, take a look at the Eq.(4.1). The analytical integration of the function results in the π number. A numerical approximation of the integral using a **mid-point numerical quadrature** method, or **middle Riemann sum** is also given [20].

$$\pi = \int_{a=0}^{b=1} \frac{4}{1+x^2}\, dx \;\approx\; h\sum_{i=0}^{n-1} f(x_i + \frac{1}{2}h), \qquad x_i = a + ih \tag{4.1}$$



(a) *fork-join* scheme in a general UNIX-like system.

(b) OpenMP *fork-join* scheme [31].

Figure 4.5: Illustrations of *fork-join* schemes. On left-hand side a general approach and on the right-hand side the approach followed by OpenMP.

```
1  double f(double x)
2  {
3    return 4 / (1 + x * x);
4  }
5
6  double integrate(int n, double a, double h)
7  {
8    double s = 0.0f;
9    for (int i = 0; i < n; i++)
10   {
11     s += f(a + (i + 0.5) * h);
12   }
13   return s * h;
14 }
```

Listing 4.6: Serial integration of Eq.(4.1).

Listing 4.6 illustrates the numerical integration written in the C programming language. The code inside the `for` loop is parallelizable because there are no dependencies between iterations. Thus, an OpenMP parallel version of the code is shown in Listing 4.7. There, it can be identified some directives:

- `#pragma omp parallel`. This directive initiates a parallel region, spawning threads who concurrently run the code that is delimited by the curly braces. This is the point where the *fork* of the Figure 4.5b happens. At the end of the parallel region, i.e., at the closing curly brace, an implicit synchronization barrier hides the *join* call, where the execution stops until all the threads of the team reach that point.

- `#pragma omp for`. The `for` directive distributes the iterations of the loop among the threads, so every thread receives a chunk of iterations to work on independently. Due to the variable, s, is declared inside the parallel region, each thread has its own copy of it, so there is no *race condition*. However, with this approach, each thread calculates a partial sum of the total integration. The way to prevent a race condition when adding the partial sums together drives to the next directive.

- `#pragma omp critical`. Concurrent write access to shared variables is what is known as a race condition, a situation that should be avoided in parallel computation because it leads to undeterminate behaviors or results. The `critical` directive solves this situation by serializing the code, which means that only one thread can run the code that is inside the critical section at a time. Obviously, serializing a code block in a parallel region is a penalty in performance, so it should be avoided when possible.

```
1  double integrate(int n, double a, double h)
2  {
3    double result = 0.0f;
4    #pragma omp parallel
5    {
6      double s = 0.0f;
7      #pragma omp for
8      for (int i = 0; i < n; i++)
9      {
10         s += f(a + (i + 0.5) * h);
11     }
12     #pragma omp critical
13     {
14       result += s;
15     }
16   }
17   return result * h;
18 }
```

Listing 4.7: Parallel integration of Eq.(4.1) with **critical** directive.

```
double integrate(int n, double a, double h)
{
  double s = 0.0f;
  #pragma omp parallel for reduction(+:s)
  for (int i = 0; i < n; i++)
  {
    s += f(a + (i + 0.5) * h);
  }
  return s * h;
}
```

Listing 4.8: Parallel integration of Eq.(4.1) with **reduction** clause.

Before finishing this introduction to OpenMP, in Listing 4.8 an enhanced version of the parallelization of Listing 4.7 is presented. Here, a convenient directive with the `reduction` clause is used. `#pragma omp parallel for` is an elegant combination of the directives `parallel` and `for` when the parallel region is only needed in `for` loops. The `reduction` clause plays the same role that the critical region of the previous example, but using OpenMP built-in functionality. Furthermore, notice that this latter implementation is the same as the serial one except for the directive. Due to non-OpenMP compilers will just ignore OpenMP directives, Listing 4.8 is a valid solution for both serial and parallel executions. In Table 4.1 are displayed the integration times for each implementation, it can be appreciated that the last one has slightly better performance than manually using the critical directive. For a comprehensive overview of OpenMP, the book [40] is highly recommended.

Parallelization	**Time [s]**
Serial	3.029893
Critical directive	0.534867
Reduction clause	0.490955

Table 4.1: Integration time of Eq.(4.1) using $n = 1,000,000,000$ intervals in an eight-core *AMD Ryzen 5000* processor.

4.3.4 Heterogeneous parallelism with CUDA

Nowadays, the presence of CPU-GPU heterogeneous architectures in supercomputer's nodes is becoming very common. This is because GPUs deliver extremely high throughput, with high power efficiency, in data-parallel problems. However, GPU does not perform independently but is used as a co-processor to a CPU. They are connected through a PCI-Express bus, where the CPU is called the *host* and the GPU is called the *device*. Figure 4.6 illustrates the heterogeneous architecture, where it can be appreciated that the GPU presents (ignoring the difference in computing cores) a similar architecture and memory hierarchy than CPU's. Thus, a GPU, when isolated, fits perfectly the shared-memory parallel programming model.

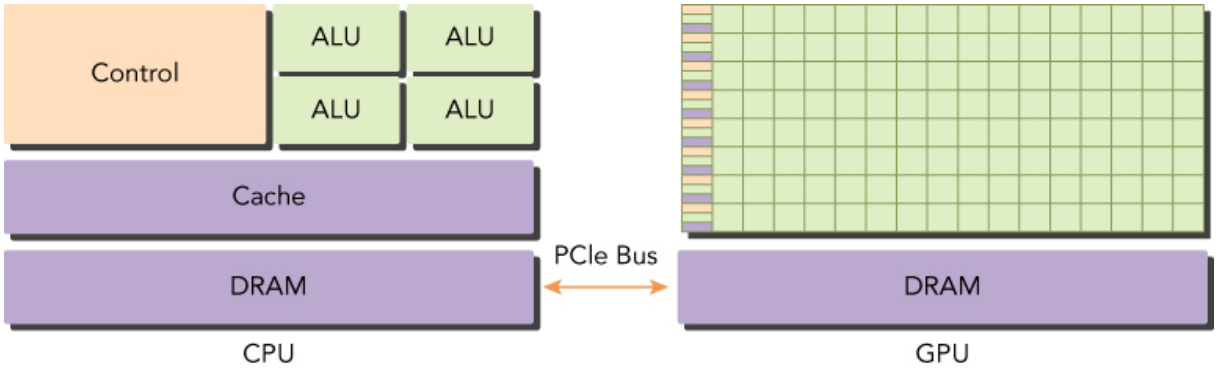


Figure 4.6: CPU + GPU heterogeneous architecture logic scheme [41].

The reason for the rise in popularity of this new heterogeneous architecture, that adds complexity to the programmer, is due to CPUs are well-suited for control-intensive tasks but are orders of magnitude slower than GPUs when it comes to compute-intensive tasks, and the other way around. Therefore, CPU + GPU architectures have complementary attributes that can be exploited in parallel computing applications to increase performance, i.e., reduce the time-to-results in engineering simulations.

A CPU-GPU application is made up of *host code* and *device code*. Host code runs on CPU and is responsible for the core logic of the application, i.e., Input/Output, algorithms, flow control, and managing the device (GPU). In Figure 4.7 is shown a scheme of the typical CPU-GPU application, where the CPU execute host code in serial, or with multiple CPU threads (for instance, using OpenMP), until it finds a compute-intensive portion of the algorithm. Then, CPU instructs the GPU to run that compute-intensive part. In the meantime, CPU can continue executing host code because device code run asynchronously, only when CPU needs the results, a synchronization is performed.

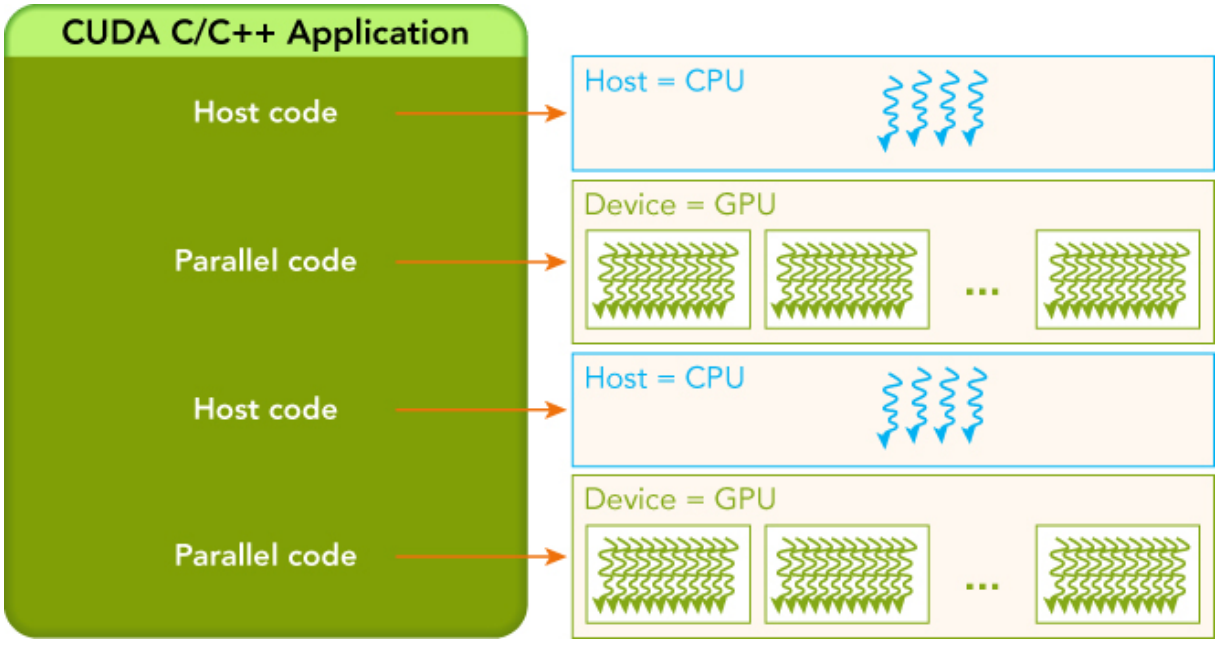


Figure 4.7: Execution scheme of a CPU-GPU application [41].

To handle CPU-GPU applications, **NVIDIA** has developed **CUDA**, a programming model to control their GPUs [42]. CUDA Toolkit is a developer platform that comes with CUDA-accelerated libraries, compilers, runtime APIs to manage the device and more. CUDA is presented to the developer as a language extension of the industry-standard C/C++. CUDA programming model propose two hierarchy structures to manage parallelism. Memory hierarchy enables the programmer to optimize the code making an efficient use of the global memory, shared memory, caches and registers, similar to the typical CPU approach. What's new in CUDA model is the thread hierarchy structure that enables organizing GPU's threads in multiple groups. A proper knowledge of how to organize threads is a critical part of CUDA programming to write efficient code. There are two levels for grouping threads, blocks of threads and grid of blocks, each group with an organization up to three dimensions (x, y, z). In Figure 4.8 is shown a graphical example of the thread hierarchy with a 2D grid containing 2D blocks of threads.

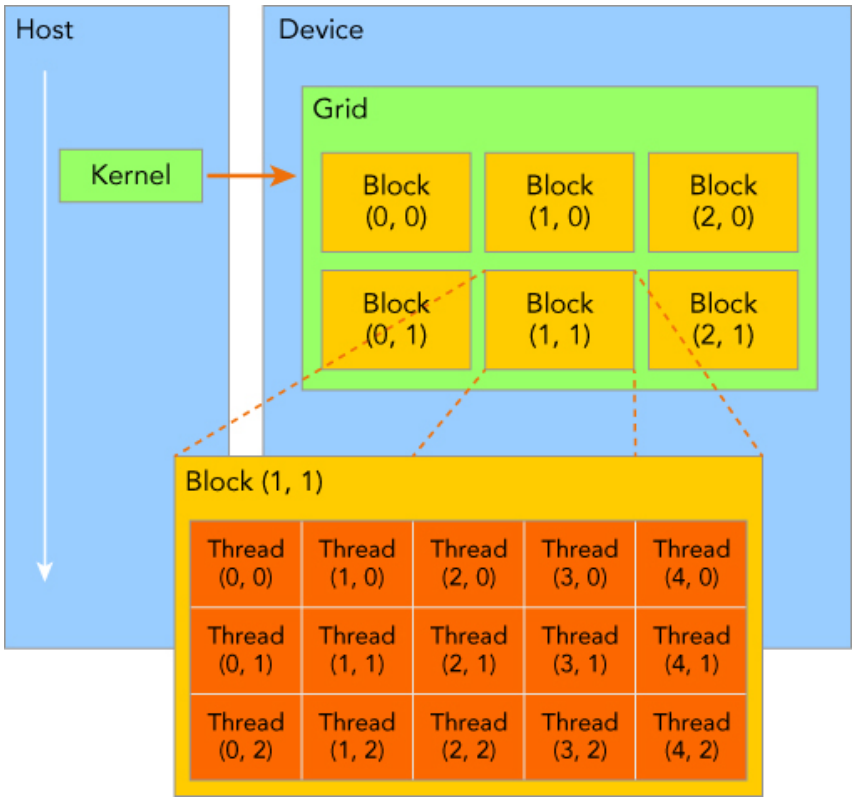


Figure 4.8: CUDA thread hierarchy structure of a 2D grid of 2D blocks of threads [41].

```
1 cudaMalloc((double **)&d_a, bytes);
2 cudaMalloc((double **)&d_b, bytes);
3 cudaMalloc((double **)&d_c, bytes);
4
5 cudaMemcpy(d_a, a, bytes, cudaMemcpyHostToDevice);
6 cudaMemcpy(d_b, b, bytes, cudaMemcpyHostToDevice);
7
8 dim3 block(blockSize);
9 dim3 grid((n + block.x - 1) / block.x);
10
11 gpu_vector_addition<<<grid, block>>>(d_a, d_b, d_c, n);
12
13 cudaMemcpy(c_gpu, d_c, bytes, cudaMemcpyDeviceToHost);
14
15 cudaFree(d_a);
16 cudaFree(d_b);
17 cudaFree(d_c);
```

Listing 4.9: CUDA runtime API calls to launch a kernel.

To illustrate the reader with a brief example of CUDA programming, let's show how to code a simple vector addition operation, as it was explained in Listing 4.4 for CPU in standard C language. Listing 4.9 shows how to send to the GPU the vector addition parallel execution, the program consists of the following steps:

1. Allocate GPU memory, lines 1-3. Notice that is almost identical to the standard "`malloc`" C function to allocate CPU memory.

2. Transfer data from CPU to GPU memory, lines 5-6. Remember that the GPU has its own global memory and cannot directly access to CPU's RAM.

3. Invoke the CUDA **kernel**, line 11. The kernel is the function that is executed by the GPU. Here, it is defined the thread execution configuration (the mentioned two-level thread hierarchy) inside triple-angle-brackets.

4. Transfer result data back from GPU to CPU memory, line 13.

5. Destroy GPU memories allocations, lines 15-17. Again, very similar to the standard "`free`" C function.

Finally, the basic idea that you have to keep in mind when writing a CUDA kernel is that it is just a serial piece of code that will be run by a single thread. Listing 4.10 shows the vector addition kernel function call (line 11 of Listing 4.9). Notice that, in contrast with Listing 4.4, the kernel has no `for` loop, instead is used the built-in variables (`blockIdx`, `blockDim` and `threadIdx`) to identify the thread and access the global index of the vectors. In line 5, the `if` condition is used to prevent threads from invalid memory access (exceeded array bounds), see Figure 4.9.

```
__global__ void gpu_vector_addition(double *a, double *b,
                                    double *c, const int n)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < n)
  {
    c[idx] = a[idx] + b[idx];
  }
}
```

Listing 4.10: Vector addition CUDA kernel.

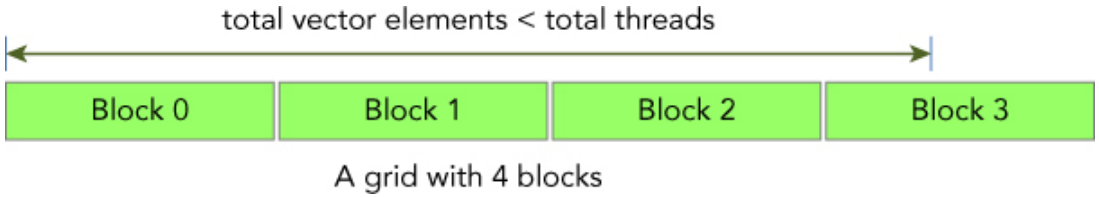


Figure 4.9: When invoking CUDA kernels, it is common that the number of threads is greater than the number of elements that will be accessed, thus illegal global memory accesses must be prevented [41].

Chapter 5

A Parallel ODE Integrator Library

The main objective of the current Master's Thesis is the development of a library that accelerates the integration of the chemical source term shown in the Eq.(2.5), in Chapter 2, whose derivation comes from the mathematical modeling of reactive flows in turbulent combustion simulations. Chapter 3 shows the mathematical theory of how to solve Eq.(2.5), a stiff ODE system, but instead of developing a new ODE solver to address the problem, widely known open-source solutions have been chosen as core dependencies. With all this context, the current chapter describes the software architecture of a parallel ODE library that makes use of the HPC tools and techniques described in Chapter 4.

The outline of the chapter is as follows. Section 5.1 describes the library dependencies. Then, from Section 5.2 to Section 5.5, the architecture of the library is explained in detail, using Unified Modeling Language (UML)[1] and some pieces of code when needed. Section 5.6 exposes how the validation of the solution has been addressed, why unit testing is important in software development and the approach taken to include it in the library. Section 5.7 shows an application example of how to use the library and how to take the advantage of its runtime configuration. Finally, in Section 5.8, a high-level application to collect data and results to a local database is described.

5.1 Library's dependencies

As mentioned at the introduction of the chapter, in this thesis no ODE solver has been developed, instead two open-source codes, which are specialized to perform the solution of stiff ODE problems, has been taken as core dependencies. Those libraries are **Cantera** and **Sundials-CVODE**. Moreover, the **pyJac** library has been included as a tool to evaluate analytically the Jacobian matrix and **Magma** to solve dense linear systems on the GPU. Lastly, **GoogleTest** is a world-wide known library that is used for testing purposes in C++ programming language.

[1] If the reader is not familiar with UML, we recommend the book [43], it is a great and accessible introduction.

5.1.1 Cantera

Taken from its official website, *Cantera is an open-source suite of tools for problems involving chemical kinetics, thermodynamics, and transport processes* [44]. It supports multiple language interfaces, such as Python, Fortran, MATLAB and C/C++. Cantera is designed using the Object-Oriented Programming (OOP) paradigm. From the thesis perspective, this library exposes an API that helps to work with chemical kinetics problems in a comfortable way. Listing 5.1 shows the internal integration method of the *CanteraIntegrator* and *CanteraIntegratorOMP* classes (from the library developed in this thesis), whose architectures will be explained in Section 5.4. What we want to show here is the part of Cantera's API needed for solving the combustion chemical integration problem.

```
1  void
2  CanteraIntegrator::integrateSystem (
3      double &temperature,
4      double &enthalpy,
5      double *species,
6      double dt)
7  {
8      auto solution = Cantera::newSolution(this->mechanism);
9      auto thermo = solution->thermo();
10     Cantera::IdealGasConstPressureReactor reactor;
11     Cantera::ReactorNet canteraIntegrator;
12
13     thermo->setState_TPY(temperature, this->pressure, species);
14     reactor.insert(solution);
15
16     canteraIntegrator.setInitialTime(0.0);
17     canteraIntegrator.setTolerances(this->reltol, this->abstol);
18     canteraIntegrator.addReactor(reactor);
19
20     canteraIntegrator.advance(dt);
21
22     // Take solution
23     temperature = reactor.temperature();
24     enthalpy = thermo->enthalpy_mass();
25     thermo->getMassFractions(species);
26 }
```

Listing 5.1: Chemical integration method of *CanteraIntegrator* class that make use of Cantera's API.

Cantera's inputs in Listing 5.1 are:

- **Temperature** as a function input, line 3.
- **Enthalpy** as a function input, line 4.
- **Species** mass fraction vector as a function input, line 5.

- **Time step** as a function input, line 6.
- **Chemical mechanism** already a property of the class, line 8.
- **Pressure** already a property of the class, line 13. Notice that a constant pressure condition is imposed.
- **Solver error tolerances** already properties of the class, line 17. These options are needed by the ODE solver.

The outputs are:

- **Temperature**, line 23.
- **Enthalpy**, line 24
- **Species** vector, line 25.

As the reader may appreciate, Cantera's API provides an abstract layer of the thermo-chemical and mathematical processes that are involved in the chemical integration that, from the programmer's perspective, is very easy to use at the cost of having no control over the operations. The underlying ODE solver that Cantera uses for the chemical integration is **CVODE**, that is explained in the next section.

5.1.2 Sundials - CVODE

Sundials is a software package that groups various mathematical libraries for solving Non-linear equations, Non-linear Ordinary Differential equations and Differential Algebraic equations [45, 46]. Particularly, CVODE is a solver for ODE IVPs in n-space. It uses variable-order and variable-step multistep methods. For nonstiff problems, CVODE uses the implicit Adams Moulton methods, but more important for our case is that for stiff problems uses BDF formulas of the form of Eq.(3.18) (visit Chapter 3 if needed). CVODE is the library that Cantera depends on for the chemical integration, thus the direct use of CVODE is more complex than Cantera, but gives more control over the data and operations and the possibility of further optimization. As well as it has been done for Cantera, let's show CVODE's API by analyzing the serial integration method of the *CVodeIntegrator* class. Listing 5.2 shows the integration method inputs and Sundials-CVODE's variables declarations.

```
1 void CVodeIntegrator::integrateSystem (
2     double *system,
3     double dt)
4 {
5     N_Vector y;
6     SUNMatrix J;
7     SUNLinearSolver LS;
8 ...
```

Listing 5.2: Chemical integration method of *CVodeIntegrator* class that make use of Sundials-CVODE's API. Part I.

- `system`, from line 2, is an array with the length of the species mechanism. The first entry is the temperature, and the rest are the species except last one. Remember Eq.(3.2), where it was explained that, due to mass conservation of the species, only $n-1$ species are required to be solved.

- `dt`, line 6, is the time step, same as in Cantera's integrator.

- `sundials::Context`, `N_Vector`, `SUNMatrix` and `SUNLinearSolver` are the Sundials objects declarations to allocate memory space for the computational operations; `y` for the species, `J` for the Jacobian matrix and `LS` for the linear system.

Listing 5.3 exposes CVODE configuration and the integration call, it consists of the following steps:

1. Allocate species memory, using the `systemSize` property, line 2.

2. Copy initial values of the problem (temperature and species), line 5.

3. Set BDF ODE solver, line 7

4. Set CVODE initial values and specify $dy/dt = f(y,t)$ function, line 9.

5. Set solver error tolerances, line 11.

6. Allocate memory for the Jacobian matrix, line 13.

7. Set linear dense solver, lines 15-17.

8. Specify analytical function or finite differences Jacobian matrix evaluation, line 19.

9. Set specific user data to be accessed in the Jacobian and $f(y,t)$ function evaluations, line 23.

10. Solver integration, line 25.

11. Copy back temperature and species solution.

```
...
y = N_VNew_Serial(this->systemSize, sunctx);

double *yptr = N_VGetArrayPointer(y);
copy(system, system + this->systemSize, yptr);

cvode_mem = CVodeCreate(CV_BDF, sunctx);

CVodeInit(cvode_mem, this->dydtFunc(), t0, y);

CVodeSStolerances(cvode_mem, reltol, abstol);

J = SUNDenseMatrix(this->systemSize, this->systemSize, sunctx);

LS = SUNLinSol_Dense(y, J, sunctx);

CVodeSetLinearSolver(cvode_mem, LS, J);

CVodeSetJacFn(cvode_mem, this->jacobianFunc());

CVodeSetMaxNumSteps(cvode_mem, maxSteps);

CVodeSetUserData(cvode_mem, uData.get());

CVode(cvode_mem, dt, y, &t0, CV_NORMAL);

copy(yptr, yptr + this->systemSize, system);
...
```

Listing 5.3: Chemical integration method of *CVodeIntegrator* class that make use of Sundials-CVODE's API. Part II.

Finally, in Listing 5.4 the memory space used by the system integration is freed.

```
...
CVodeFree(&cvode_mem);
SUNLinSolFree(LS);
N_VDestroy(y);
SUNMatDestroy(J);
}
```

Listing 5.4: Chemical integration method of *CVodeIntegrator* class that make use of Sundials-CVODE's API. Part III.

It is easy to appreciate that CVODE requires more configuration details and data management that Cantera's interface, but its API allows further customization and optimization. Furthermore, the reader may have noticed that two key functions are specified in the

CVODE configuration, i.e., $f(y,t)$ and the Jacobian evaluation function. In Cantera, those functions are handled internally by specifying the chemical mechanism, but with CVODE is the programmer's concern. So this topic drives to the next dependency which is the **pyJac** library.

5.1.3 pyJac

As it has been described in Chapter 3, the two main ways of Jacobian matrix evaluations are; analytically, by deriving formulas, or numerically, by using finite differences methods. **pyJac** is a Python package that generates analytical Jacobian matrices for chemical kinetics applications [47]. It is an open-source tool that produces the source code of the functions, or subroutines, for the evaluation of the chemical reaction mechanism and its associated analytical Jacobian matrix (a mechanism chosen by the user). For CPU operations, the source code generated is via the C/C++ language, while for GPU operations is via CUDA language. The package assumes a constant-pressure hypothesis, so it matches the requirements of our physical problem.

Regarding parallelization, for the CPU solutions, pyJac is thread-safe, so it can be used in OpenMP parallel regions. And for GPU solutions, pyJac adopted a "per-thread" parallelization (instead of "per-block"), it means that every GPU thread evaluates a single mechanism with its associated Jacobian matrix. With this approach, each GPU thread is involved in a different ODE system problem. Therefore, a GPU "per-thread" parallelization means that CVODE solver is handling multiple ODE systems simultaneously, using the GPU for Jacobian matrix factorization and for solving the linear systems. Currently, Sundials-CVODE is capable of GPU parallelization by adopting a library dependency, Magma.

5.1.4 Magma

Magma stands for *Matrix Algebra on GPU and Multi-core Architectures*. It is a linear algebra library for heterogeneous computing [48]. Magma presents similar interfaces for supporting widely known packages, such as LAPACK and BLAS [49, 50], to easily porting algebra software components to GPU.

Magma supports *dense batched LU methods* [51]. These routines consist in solving LU factorization and linear systems for multiple problems at once (see Figure 5.1). This functionality is useful when each of the systems is not large enough to be worth solving with the GPU (it does not offset the penalty of data movements) but many of them can be solved in parallel. This is the case of the chemical reaction scenario, where common mechanisms involve from 10 to 2000 species, so GPU has enough memory space to hold many of those systems. Magma is supported by Sundials-CVODE's solver to parallelize the integration of multiple ODE systems using GPU devices.

5.1.5 GoogleTest

The last dependency of the library is **GoogleTest**. It is a testing framework to write C++ tests [52]. The library includes this framework to manage the validation of the

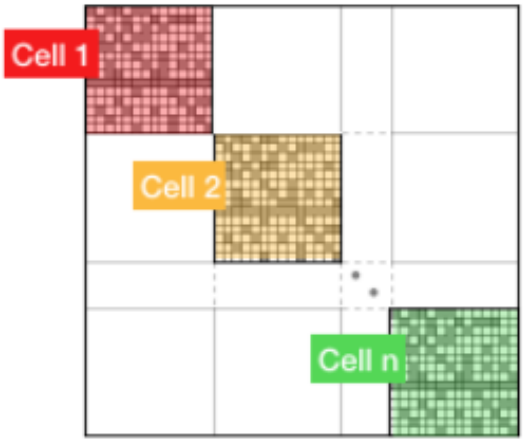


Figure 5.1: Matrix representation for dense batched LU methods, handled by Magma package.

solutions. Unit tests help prevent regressions in a continuous development code base. Therefore, having a test battery opens the possibility to set up a continuous integration pipeline.

5.2 The Context object and its components

The remaining sections will describe the implementation details of the library. The current section explains the classes that compound the Context object, which are:

- OutFileService
- BaseLogger
- Mesh

5.2.1 OutFileService

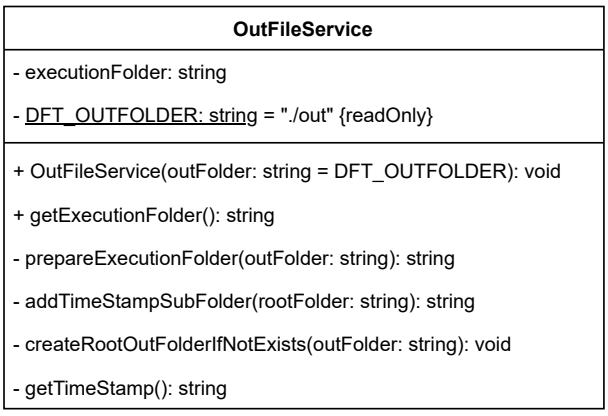


Figure 5.2: UML diagram OutFileService class.

The OutFileService class, as the name suggests, is a component that is used as a service inside the library. It holds the Unix-like path where all the outputs of the execution will be saved. This path can be accessed by calling the method "`getExecutionFolder(): string`". Figure 5.2 shows the methods and properties of the class. By default, the output will be saved in the relative path "`./out`", but the constructor allows you to choose another one. Once the path is defined, the instance will create the output directory if it does not exist and, inside this directory, another with the current timestamp as its name.

5.2.2 Thread-safe logging system

Either in development phases or trying to find a bug in a code that is in production, having a good logging system is almost mandatory. To do so, it has been developed a thread-safe logging system that supports log messages in concurrent and parallel regions, such as OpenMP's.

A thread-safe system means that the system is protected against possible race-conditions. This is the case when two or more threads want to write a log message simultaneously. To avoid that situation, the code must be serialized using *mutual exclusion* (mutex) objects, critical regions or similar techniques. However, Input-Output operations are really slow, compared to reading-writing memory variables. Therefore, to minimize the penalization of serializing an Input-Output instruction, the system follows a *Master-Worker pattern* with only one worker. With a master-worker scheme, the Input-Output operation is decoupled from the thread that has the message to write, to a thread whose sole task is to write log messages. The latter thread can be called the *logging-thread*. Thus, the thread that has the message, instead of directly writing in the Input-Output buffer, it put the message in a First-In, First-Out (FIFO) queue. The logging-thread is continuously taking messages from the queue and writing it into the output buffer.

Figure 5.3 shows the architecture of the logging system in a UML diagram. Four classes can be identified, **ThreadSafeQueue**, **BaseLogger**, **ConsoleLogger** and **FileLogger**. Let's explain them a little further.

ThreadSafeQueue

C++ standard library does not provide a thread-safe queue, so it has been necessary to implement one that uses mutex. Just two actions has been implemented, *push* to add a message to the queue and *pop* both for taking and deleting a message from the queue.

BaseLogger

The BaseLogger is the abstract class of the two forthcoming classes. It is the engine of the whole system. When initializing a class that inherits from the BaseLogger, the constructor method start a new thread (the logging-thread) with the task of "`processQueue()`", that is an infinite loop that takes messages from the ThreadSafeQueue and writes them using the "`log(msg: string): void`" method. The latter function is an abstract method, so concrete classes must implement this method in their own way. Moreover, the *LogLevel* enumeration allows you to choose the level of messages to write, DEBUG will write all types of messages, whereas INFO will ignore debug-type messages.

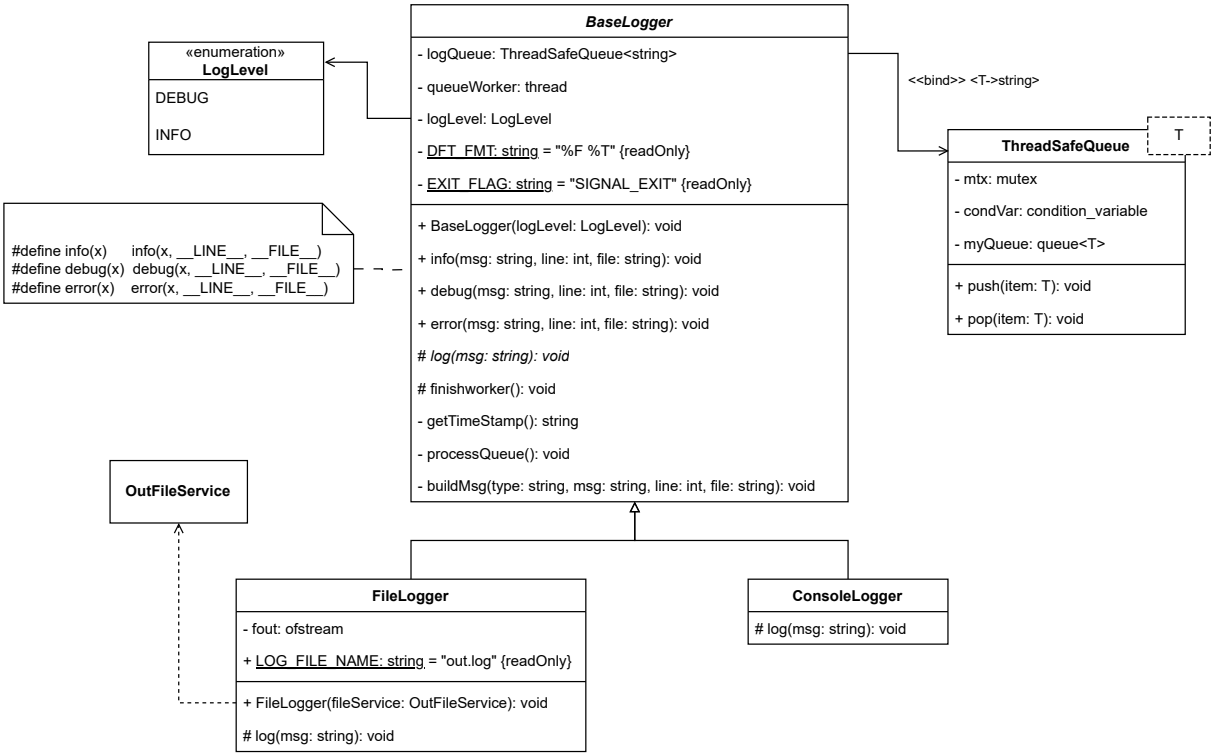


Figure 5.3: UML diagram of the thread-safe logging system.

The public interface that concrete loggers present is the same for all. It consists in the following methods:

- `info(msg: string, line: int, file: string): void`
- `debug(msg: string, line: int, file: string): void`
- `error(msg: string, line: int, file: string): void`

These methods define the type of message being written, i.e., information, debug or error message. Actually, due to the macro defined on the left of the Figure 5.3 there is no need to specify the line and the file when calling those methods. An example of the message is the following:

```
[139748955570176][2023-05-26 17:40:14,569][main.cpp:45] INFO: Integrating systems
```

The message is composed of:

1. Thread ID.
2. Date and Time.
3. File and line from where the message was sent.
4. Type of message: `INFO`, `DEBUG`, `ERROR`.
5. Message content.

Finally, when an instance of the BaseLogger is destroyed, the destructor method puts an

exit signal in the queue. Eventually, the logging-thread reads the exit signal from the queue, exiting the infinite loop and joining the master thread.

ConsoleLogger

ConsoleLogger is a concrete class of the abstract BaseLogger. It overrides the "`log(...)`" method to send the message to the console, also known as the standard output (stdout).

FileLogger

The FileLogger overrides the "`log(...)`" method to write the logs in a permanent file. It needs the OutFileService class in the constructor because it needs to request the path where to create the log file.

5.2.3 Data structure

This section explains how the data are managed and stored inside the library. To begin with, it is important to define what a mesh point is. As described in Chapter 2, to integrate the Eq.(2.5), it is necessary to know the initial thermochemical state of each discrete point of the mesh. Thus, from the point of view of the library, to integrate a single ODE system, a mesh point may have the following properties:

- Temperature, in Kelvin.
- Number of species of the mechanism and their mass fraction values.
- The specific enthalpy, in J/kg.
- The coordinates of the point with respect to a reference system, in meters.

The first two properties are absolutely necessary for its integration because they are the unknowns of the PDE system. The specific enthalpy is a secondary variable that can be calculated from the first two, so it is optional. And the coordinates are also optional.

Point subsystem

In Figure 5.4 is illustrated the UML diagram of the **Point** class and its closest dependencies and relationships. The Point presents a large public interface where all its properties can be accessed. However, a Point instance may not have all the properties, but temperature and species are mandatory. The method "`isReady(): bool`" ensures that the latter mentioned properties are set, by returning a "`true`" value, otherwise it returns "`false`". To be able to create a Point with different properties in runtime, a widely known creational design pattern has been used, the *Builder Pattern*. This pattern needs an auxiliary class, the **PointBuilder** class, that allows you to produce different types of Points depending on the items that had been added (temperature, species, enthalpy, etc). Finally, the **Coords** class provides an abstract layer to handle 3D coordinates in a comfortable way.

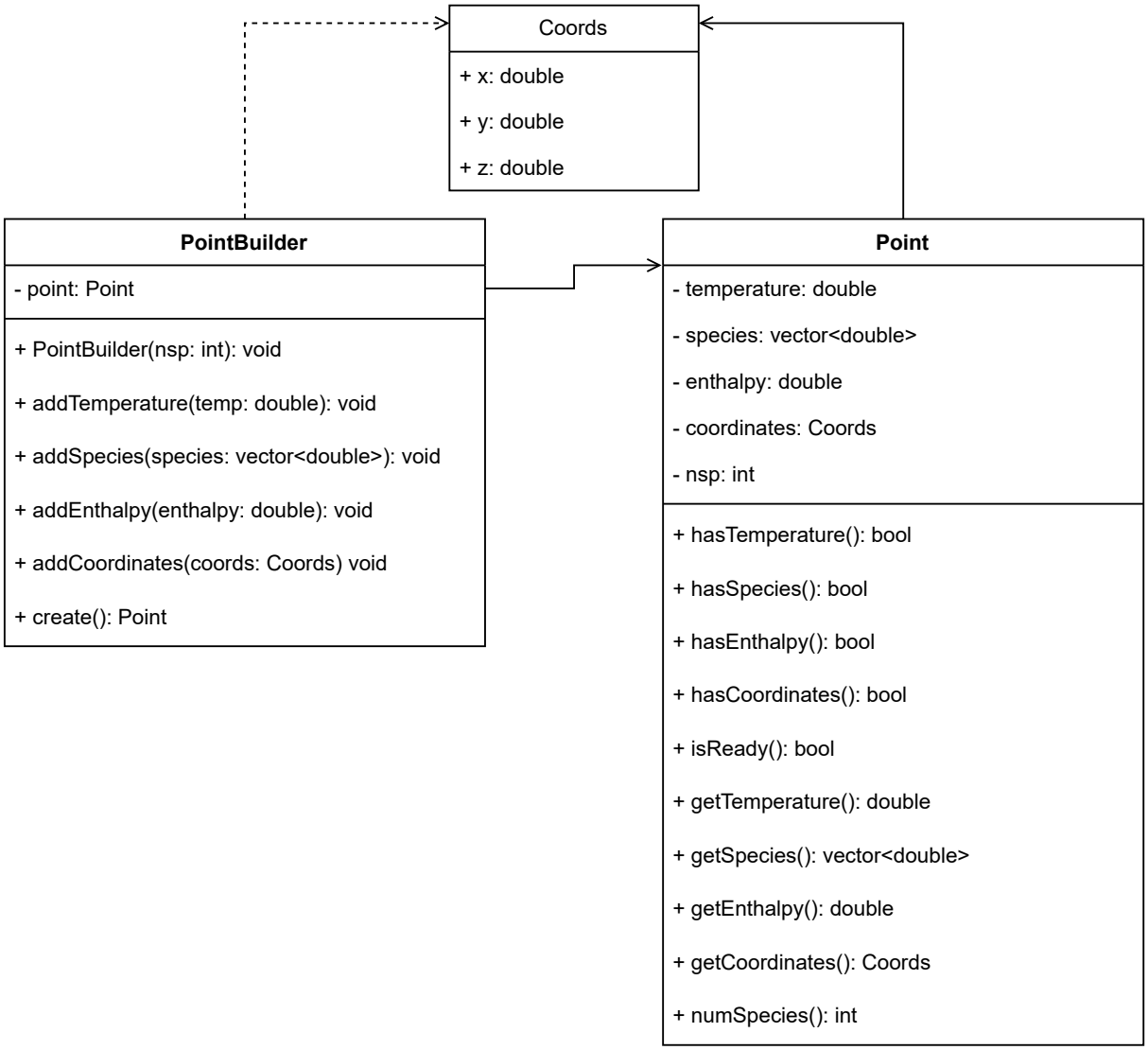


Figure 5.4: UML diagram of the Point subsystem.

Mesh system

The **Mesh** class is the core of the library, it is where all the systems are stored and, once integrated, is the object that has the solutions. Figure 5.5 shows the properties and methods of the class. They are somehow similar to the Point class, but instead of storing a single system, the Mesh class has vectors and matrices to store multiple systems. The class is used as follows:

"`addPoint(newPoint: Point): void`" method is used to add new points/systems to the Mesh. Once the first Point is added, the Mesh class registers its type, i.e., what properties it has. Then subsequent added points must have the same properties, a characteristic that is controlled by the method "`isCompatible(point: Point): bool`". The public methods relevant to integrator classes (see the Section 5.4) are:

- "`numSystems(): int`". To check how many systems the mesh holds.

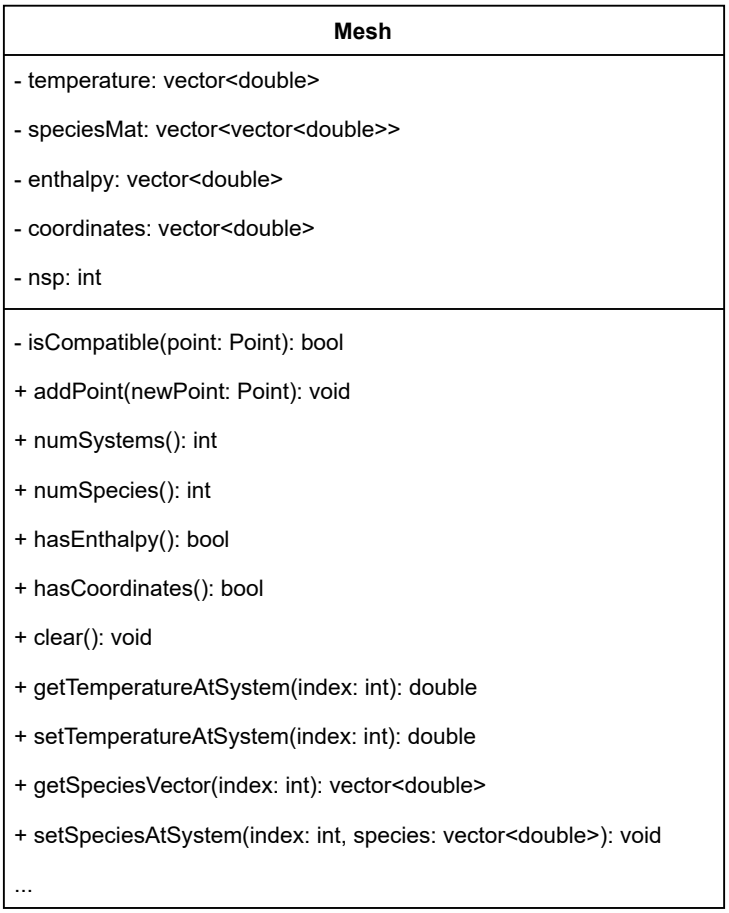


Figure 5.5: Properties and method of the Mesh class.

- "`numSpecies(): int`". It returns the size of the chemical mechanism.
- "`get...AtSystem(index: int): <T>`". Returns the value of the requested property (temperature, species, enthalpy or coordinates) at the given index.
- "`set...AtSystem(index: int, property: <T>): void`". Change the value of the selected property at the given index. These methods are mainly used to save the results.

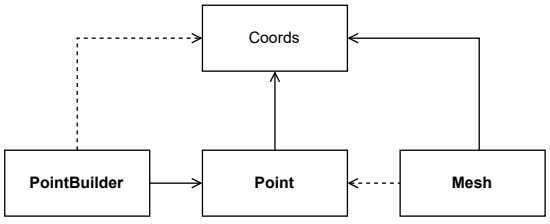


Figure 5.6: UML diagram of the Mesh system.

Figure 5.6 illustrates the relationships between the Mesh system. Notice that the Mesh is not composed by Points, is just a dependency relationship. This is because, to optimize the memory access pattern, the Mesh class stores the properties of each system in contiguous memory by extracting the data of each point.

5.2.4 Context

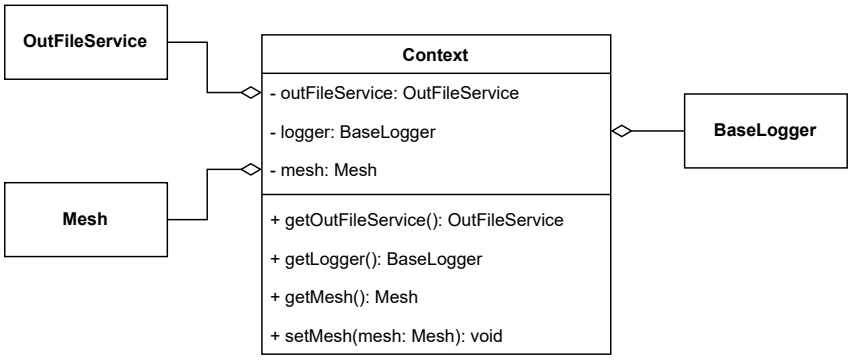


Figure 5.7: UML diagram of the Context class and its components.

As described at the beginning of this section, the Context object is composed by an OutFileService instance to keep track of the output directory path, a Mesh instance that stores the data and a BaseLogger instance (see Figure 5.7). It is important to highlight that the BaseLogger abstract class cannot be instantiated, but a concrete class of it. This is one of the four pillars of the Object-Oriented Programming, **Polymorphism**. Actually the four pillars of OOP paradigm are used extensively in the library, those pillars are:

- **Abstraction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

5.3 Input and Output

Now that it has been explained how the library stores the data, it is time to show how to insert that data into the Mesh, and how to get the results from it. This is called the Input-Output system (or module), and its UML diagram is shown in the Figure 5.8. Let's break it down in parts.

On the left-hand side of the diagram it is the **Reader** branch. The Reader abstract class has an association with the Context class because it needs its Mesh and BaseLogger instances. Furthermore, it has an abstract method "`read(): void`" that must be implemented by concrete classes. This method must be used to read the data from a certain source and insert it into the Mesh instance. At the time of writing this thesis, only a concrete class has been developed, the **CsvReader** class. This is because no more data sources, apart from Comma-Separated Values (CSV) files, has been used. However, if in the future any developer of the library needs to insert data from other sources, he/she should create another class that inherits from the Reader class and implement its abstract method "`read(): void`". Regarding the CsvReader, it needs the absolute path of the input CSV file for the construction of the object, then, when invoking the "`read(): void`" method, the instance will read the rows, line by line, and it will insert the systems into the Mesh instance.

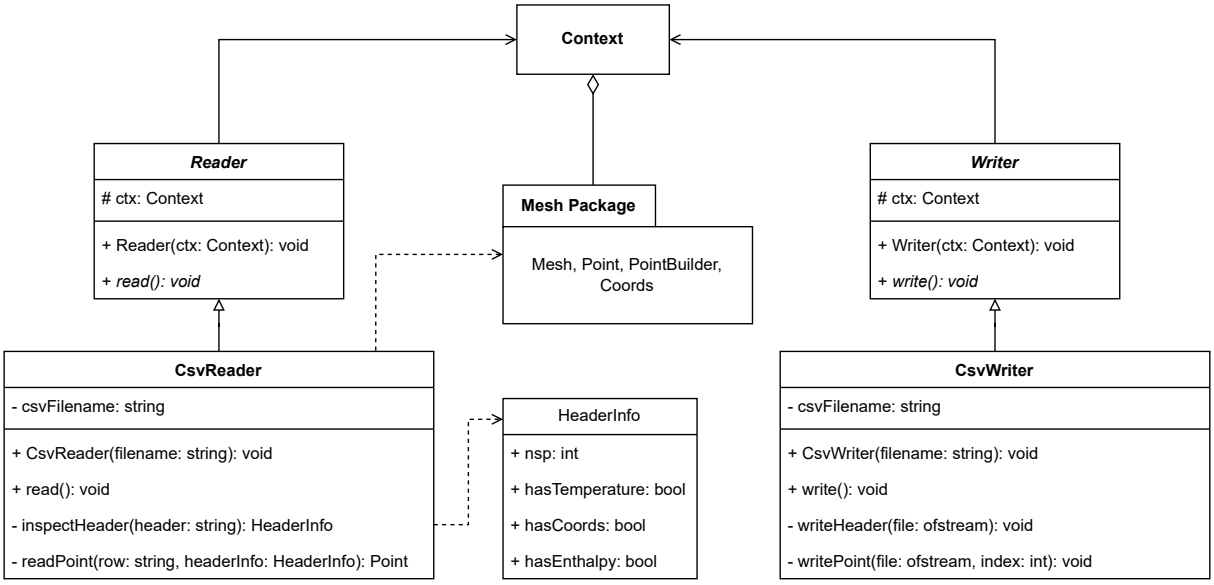


Figure 5.8: UML diagram of the Input-Output system.

On the right-hand side of the diagram it is the **Writer** branch. Similarly to the Reader branch, the Context class is associated with the **Writer** abstract class. Again, only a concrete class has been developed, the **CsvWriter**. This class makes use of the OutFileService instance of the Context object to get the output path where to write the output CSV file. Finally, when calling the method "`write(): void`", the CsvWriter writes sequentially line by line the systems that reads from the Mesh instance.

5.4 Integrators

There are five different integrators at the moment of writing this thesis:

- **CanteraIntegrator**
- **CanteraIntegratorOMP**
- **CVodeIntegrator**
- **CVodeIntegratorOMP**
- **CVodeIntegratorGPU**

They all present the same public interface, which means that they all share the highest superclass, the **Integrator** abstract class. Figure 5.9 shows the properties and methods of the Integrator abstract class. Those methods are:

- `init(ctx: Context, config: IntegratorConfig): void`
- `integrate(dt: double): void`
- `clean(): void`

Once more, the Context class is an association of the Integrator class, it is obvious that

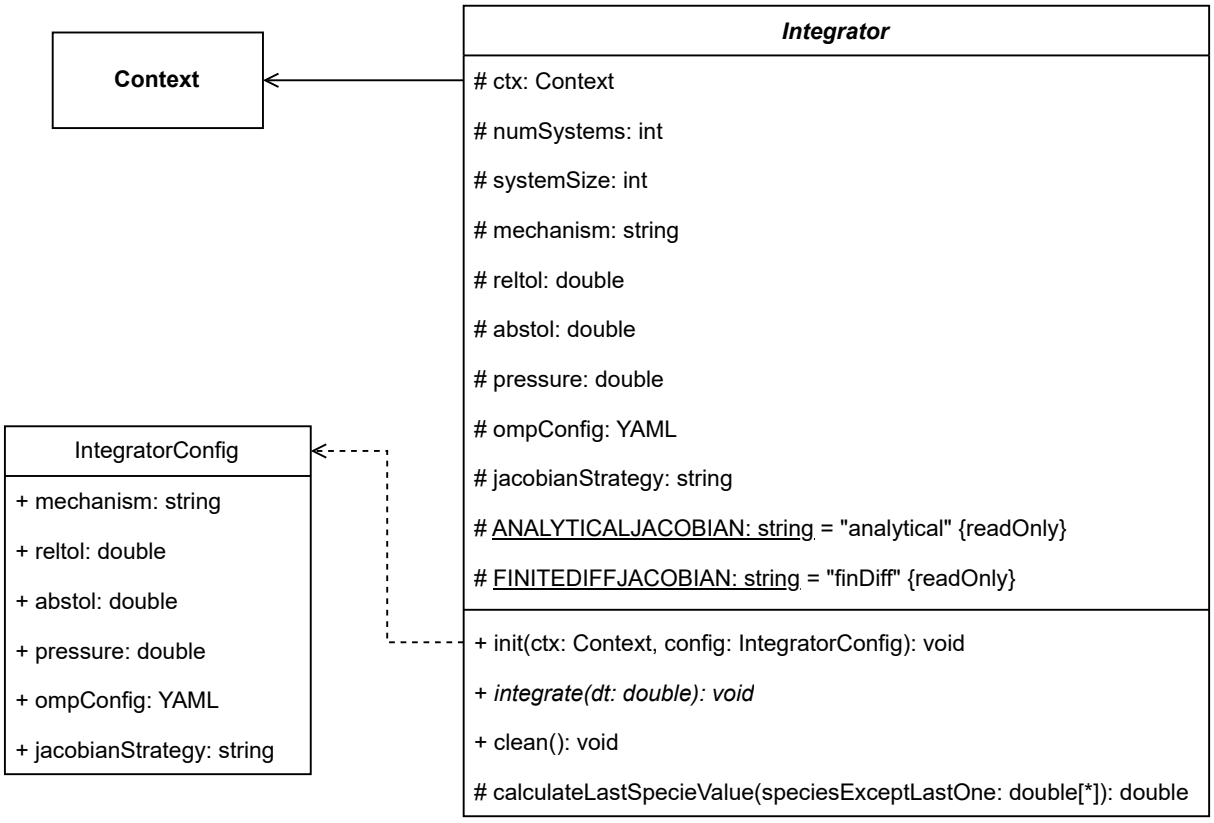


Figure 5.9: Properties and methods of the Integrator abstract class.

the latter class needs the Mesh instance for the integration and the BaseLogger for traceability and debug purposes. Furthermore, the "`init(...)`" call also request an instance of the **IntegratorConfig**, a class that holds all the parameters that every integrator needs. The "`integrate(dt: double): void`" method is the abstract method that every integrator that inherits from this abstract class must implement in its own way. Lastly, the "`clean(): void`" can be overridden to free memory allocations (or any other similar operation), this method should be called once the integration has finished.

Once described the basic operations that integrators have, let's explain a little further the concrete implementation of those integrators.

5.4.1 Cantera integrators

CanteraIntegrator class is the basic Cantera serial integrator. Listing 5.5 shows the source code that overrides the abstract method "`integrate(dt: double): void`" of the Integrator superclass. First, the code gets all the data from the Mesh instance, as references (using C/C++ raw pointers). Then, a `for` loop that iterates through all systems calls sequentially the integration method presented in Listing 5.1. The conditional sentences are necessary to distinguish whether the Mesh points have been inserted with the enthalpy property, otherwise that property is ignored when collecting the solution.

```
void CanteraIntegrator::integrate(double dt)
{
    Integrator::integrate(dt);
    double *temperatures = nullptr;
    double *enthalpies = nullptr;
    double *species = nullptr;

    if (this->mesh->hasEnthalpy())
        enthalpies = this->mesh->getEnthalpyPointer();

    temperatures = this->mesh->getTemperaturePointer();

    for (int i = 0; i < this->numSystems; i++)
    {
        species = this->mesh->getSpeciesPointer(i);
        if (this->mesh->hasEnthalpy())
        {
            this->integrateSystem(temperatures[i],
                                  enthalpies[i],
                                  species, t);
        }
        else
        {
            double dummyEnthalpy;
            this->integrateSystem(temperatures[i],
                                  dummyEnthalpy,
                                  species, t);
        }
    }
}
```

Listing 5.5: Integrate method overriden by the serial CanteraIntegrator.

As illustrated in the top-left corner of Figure 5.10, **CanteraIntegratorOMP** inherits from the serial CanteraIntegrator. This integrator is an OpenMP parallel version of the latter. It does not override the core integration method, "`integrateSystem(...)`" from Listing 5.1, but overrides the "`integrate(dt: double): void`" as shown in Listing 5.6. Almost no differences can be found among Listings 5.5 and 5.6, just the "`#pragma omp parallel for schedule(runtime)`" compiler directive. The parallelization of the loop is straightforward because iterations are independent of each other. The "`schedule(runtime)`" clause permits a scheduling configuration in runtime, as it will be shown in Section 5.5.

```
void CanteraIntegrator::integrate(double dt)
{
    Integrator::integrate(dt);
    double *temperatures = nullptr;
    double *enthalpies = nullptr;
    double *species = nullptr;

    if (this->mesh->hasEnthalpy())
        enthalpies = this->mesh->getEnthalpyPointer();

    temperatures = this->mesh->getTemperaturePointer();

    #pragma omp parallel for schedule(runtime)
    for (int i = 0; i < this->numSystems; i++)
    {
        species = this->mesh->getSpeciesPointer(i);
        if (this->mesh->hasEnthalpy())
        {
            this->integrateSystem(temperatures[i],
                                  enthalpies[i],
                                  species, t);
        }
        else
        {
            double dummyEnthalpy;
            this->integrateSystem(temperatures[i],
                                  dummyEnthalpy,
                                  species, t);
        }
    }
}
```

Listing 5.6: Integrate method overridden by the parallel CanteraIntegratorOMP.

5.4.2 CPU CVODE integrators

The serial **CVodeIntegrator** is a bit more complex than Cantera's because it comprises more pieces. As shown on the right-hand side of the Figure 5.10, CVodeIntegrator implements the **MechanismDriver** interface to provide the CVODE solver with the chemical reaction and the analytical Jacobian matrix functions. These functions link to the second dependency (Sundials-CVODE is the first one), pyJac. Moreover, CVODE's interface allows you to have arbitrary data to access them when the latter functions are called. These data are reflected in the **UserData** class association. Here, the "`integrate(...)`" method is overridden as shown in Listing 5.7. Three steps can be identified:

1. Get data from the Mesh instance and reorder it to use it in the CVODE solver.

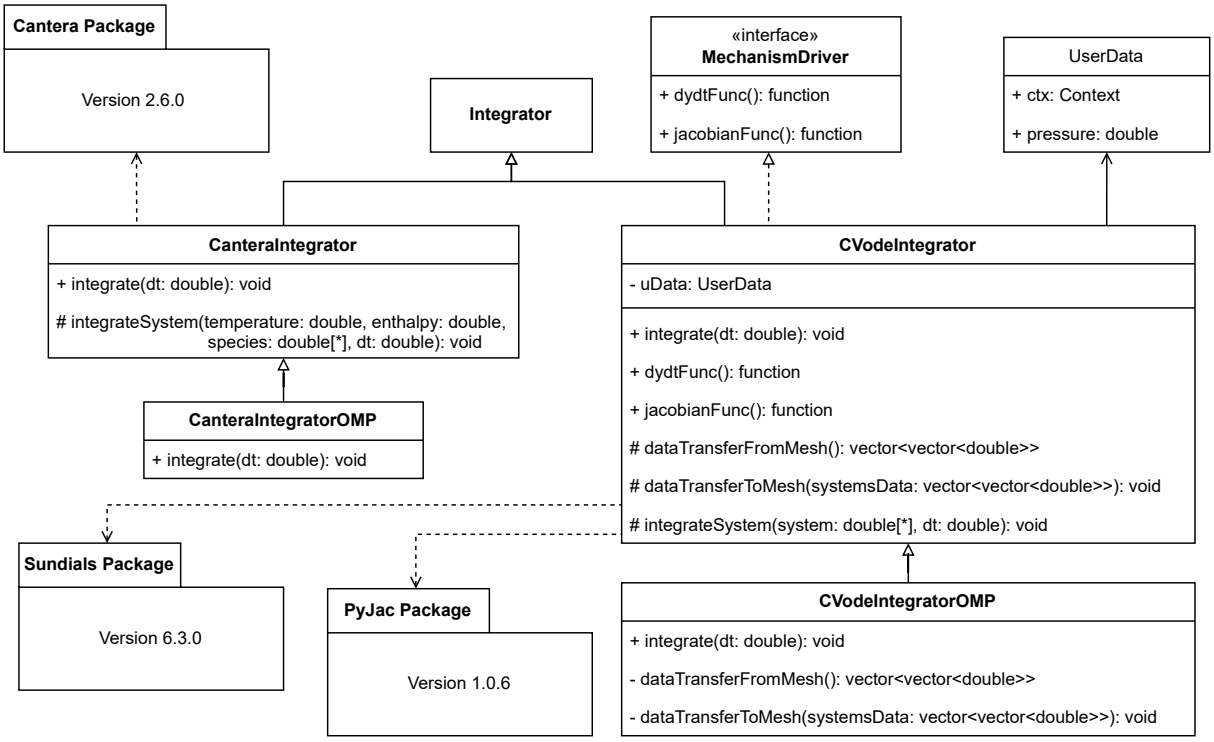


Figure 5.10: UML diagram of the CPU integrators.

2. Integration `for` loop.

3. Reorder data and insert it back into the Mesh instance.

```
1  void CVodeIntegrator::integrate(double dt)
2  {
3      Integrator::integrate(dt);
4
5      vector<vector<double>> systemsData =
6                                      this->dataTransferFromMesh();
7
8      for (int i = 0; i < this->numSystems; i++)
9      {
10         this->integrateSystem(systemsData[i].data(), t);
11     }
12
13     this->dataTransferToMesh(systemsData);
14 }
```

Listing 5.7: Integrate method overridden by the serial CVodeIntegrator.

Listing 5.8 shows in detail the source code of the first step. It consists in allocating memory space for all ODE systems. Then, in a `for` loop, each system is collected from the Mesh instance and stored in contiguous memory, where the temperature is the first item, followed by the corresponding species except last one. The second step, the integration,

has been explained in Section 5.1 when analyzing the Listings 5.2, 5.3 and 5.4. The third step is analogous to the first one, but the other way around. Inside the `for` loop of Listing 5.9 the temperature is extracted from the first item of the corresponding row of the "`systemsData`" matrix. Notice that the last species is not collected directly from the results, instead it is calculated using the "`calculateLastSpecieValue(...)`" method that applies Eq.(3.2).

```
vector<vector<double>>
CVodeIntegrator::dataTransferFromMesh()
{

    vector<vector<double>> systemsData(
        this->numSystems, vector<double>(this->systemSize, 0.0f)
    );

    vector<double> temperatures =
                        this->mesh->getTemperatureVector();

    vector<double> species;
    for (int i = 0; i < this->numSystems; i++)
    {
        species = this->mesh->getSpeciesVector(i);
        systemsData[i][0] = temperatures[i];
        copy(
            begin(species), end(species) - 1,
            begin(systemsData[i]) + 1
        );
    }

    return systemsData;
}
```

Listing 5.8: "dataTransferFromMesh()" method of serial CVodeIntegrator.

```
void
CVodeIntegrator::dataTransferToMesh(
    vector<vector<double>> systemsData
)
{
    double *temperatures =
                this->mesh->getTemperaturePointer();

    double *species;
    for (int i = 0; i < this->numSystems; i++)
    {
        temperatures[i] = systemsData[i][0];
        species = this->mesh->getSpeciesPointer(i);
        copy(
            begin(systemsData[i]) + 1,
            end(systemsData[i]),
            species
        );

        species[this->systemSize - 1] =
                    this->calculateLastSpecieValue(species);
    }
}
```

Listing 5.9: "dataTransferToMesh()" method of serial CVodeIntegrator.

Now that the serial CVodeIntegrator has been explained in detail, the OpenMP parallel **CVodeIntegratorOMP**, that inherits from the latter, is exactly the same except for the fact that every `for` loop that has been mentioned is parallelized using the OpenMP compiler directive that has been used in CanteraIntegratorOMP.

5.4.3 CVodeIntegratorGPU

Version 2.6.0 of Cantera's package does not support GPU parallelization, consequently, only one parallel GPU approach has been developed. The **CVodeIntegratorGPU** class does not inherit from CVodeIntegrator (as CVodeIntegratorOMP does) because its implementation is completely different. Instead, **CVodeIntegratorGPU** inherits directly from the Integrator abstract class and implements the MechanismDriver interface (see Figure 5.11). The arbitrary data for the CVODE also varies a little with respect to the serial and OpenMP versions, **GPUUserData** contains many more properties that are necessary to handle GPU memory during the CVODE integration. As explained in Section 5.1, the CVODE GPU implementation has one extra dependency, apart from Sundials-CVODE and pyJac, which is Magma.

The source code of the CVodeIntegratorGPU is a lot more extensive than the previous integrators (more than 1000 lines), so it is not convenient to put it all here for its explanation. However, a high-level explanation, using the properties and methods, of the UML

diagram of Figure 5.11, is sufficient to understand how it works.

```
while (this->processedSystems < this->numSystems)
{
    SystemsToProcess systemsToProcess =
                        this->calculateSystemsToProcess();

    this->integrateBatch(dt, systemsToProcess);

    this->processedSystems +=
                        systemsToProcess.realSystemsToProcess;
}
```

Listing 5.10: The integration "while" loop of the CVodeIntegratorGPU.

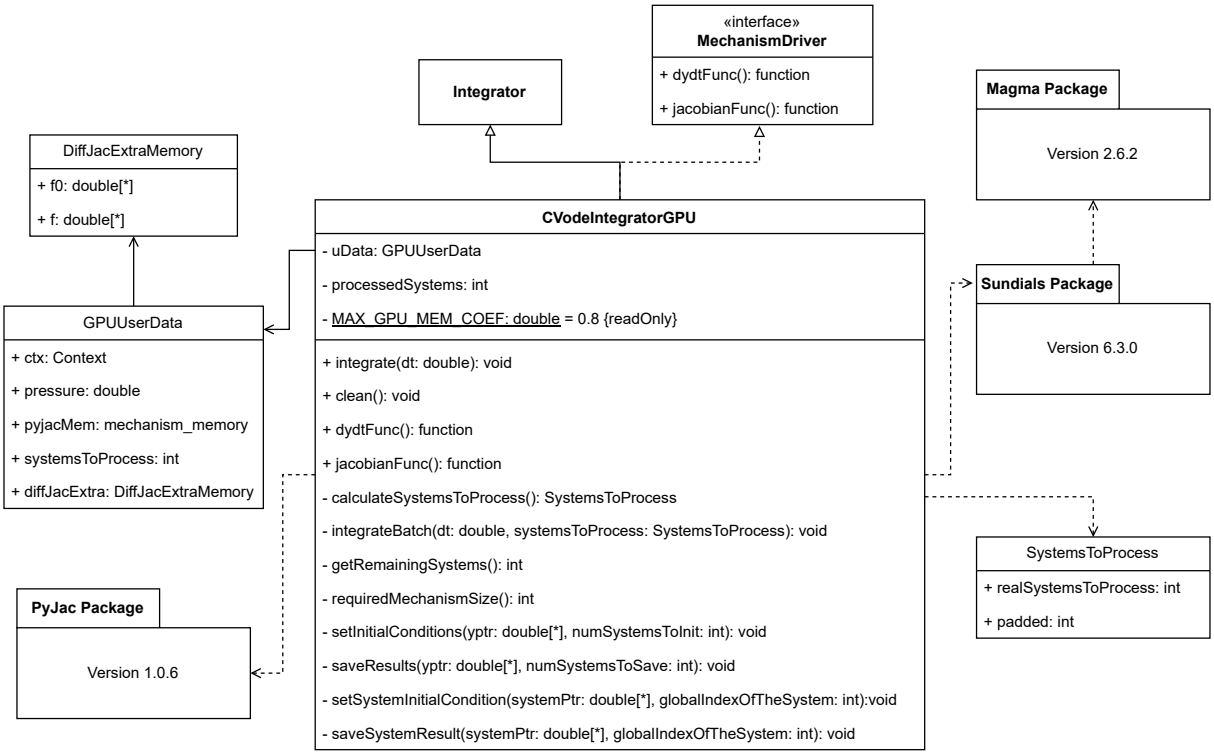


Figure 5.11: UML diagram of the CVodeIntegratorGPU class.

As well as the previous integrators, everything starts from the call "`integrate(...)`". This function drives to the `while` loop of the Listing 5.10. There are three instructions inside the loop:

1. "`calculateSystemsToProcess()`". First, this method calculates the number of systems that can be allocated in the GPU memory. Second, checking the value of the variables "`numSystems`" and "`processedSystems`", it calculates how many systems are left to integrate. Finally, the method returns the minimum value from the two previous calculations, resulting in the number of systems to integrate in the current iteration.

2. "`integrateBatch(dt, systemsToProcess)`" method takes the integration time range and the number of ODE systems to integrate. Internally, this method handles the GPU memory space allocation and the Sundials-CVODE objects and functions to integrate all the system in parallel using the GPU with the Magma library.

3. "`this->processedSystems += systemsToProcess.realSystemsToProcess`". This last step refers to an accumulation variable to keep track of the number of systems that have already been integrated. This is necessary for the exit loop condition.

5.5 Runtime library configuration

At this point of the chapter, all the components that compound the library has been explained with a certain level of detail. We have covered the logging system, the data structure, the Context class, how the input and output of the data are managed and the different types of integrators available. All these subsystems have been designed with an Object-Oriented Programming approach and using multiple design patterns. This gives the library great versatility, which means that with just one-time compilation, the library is highly configurable at runtime.

```
outFileService:
    outFolder: null # if null, default is ./out
reader:
    type: CsvReader
    filename: {absolute_path}/input.csv # Only for file readers
writer:
    type: CsvWriter
    filename: result.csv
integrator:
    type: CVodeIntegratorOMP
    omp:
        threads: 14
        schedule:
            type: static
            chunk: 1
    mechanism: gri30.yaml
    jacobianStrategy: analytical
    reltol: 1.0e-6
    abstol: 1.0e-10
    pressure: 101325.15
    dt: 1.0e-3
logger:
    type: FileLogger
    logLevel: DEBUG
```

Listing 5.11: A YAML file example to configure the library in runtime.

Listing 5.11 shows an example of how the library can be configured during runtime. It

consists in a YAML file with all the specifications needed by the library. Let's cover every field one by one:

- `outFileService`. This field is used to configure the OutFileService class. It has a unique subfield:
  - `outFolder`. This subfield specifies the directory where all the output files, that the library generates during the execution, will be written. The path can be either absolute or relative. If the directory does not exist, it will be created by the library. If it exists, but the path corresponds to a file instead of a directory, the file will be removed and replaced by the directory. Lastly, if the field is filled with the string "null", the library will choose the default directory, "./out".
- `reader`. This field refers to the data input subsystem.
  - `type`. Here it is specified which one of the concrete classes of the abstract Reader class is going to be used. At this moment, only "CsvReader" is available.
  - `filename`. This subfield is only required when using the CsvReader type. It has to be filled with the absolute path of the CSV input file.
- `writer`. Refers to the data output subsystem.
  - `type`. Similar to the reader fields, a concrete class of the Writer abstract class must be chosen. In this first version of the library only "CsvWriter" is available.
  - `filename`. Again, This subfield is only required when using the CsvWriter type. The field must be filled with the filename of the CSV output file without its path (the path is determined by the `outFileService`→`outFolder` field).
- `integrator`. This field gathers all the configuration related to the integrator system.
  - `type`. As it has been seen in Section 5.4, there are five types of integrators: "CanteraIntegrator", "CanteraIntegratorOMP", "CVodeIntegrator", "CVodeIntegratorOMP", "CVodeIntegratorGPU".
  - `mechanism`. Refers to the combustion chemical mechanism. Some sample YAML files can be found in the "resources/mechanisms" directory of the current library repository.
  - `jacobianStrategy`. The available options are "analytical" or "finiteDifferences". It refers to the way the Jacobian matrix will be evaluated. With the former option, the Jacobian matrix is evaluated using the analytical function provided by pyJac, otherwise first order finite difference formulas will be used. This option is just available for CVODE integrators (including the GPU version)
  - `reltol` and `abstol`. These options specify the maximum relative and absolute error tolerances of the ODE solver.
  - `pressure`. Remember that, in the combustion problem that we are dealing with, it has been assumed a constant pressure hypothesis. Here the pressure

is set in Pascal units.

- `dt`. Integration time range.
- `omp`. This field only affects OpenMP integrators.
  - `threads`. Number of OpenMP threads that will be used in parallel regions. If the number is greater than the maximum available in the current node, then the maximum of the node will be taken.
  - `schedule`. OpenMP scheduling options.
    - `type`. Set the type of scheduling. The available options are, "static", "dynamic", "guided" and "auto".
    - `chunk`. This field specifies the size of the iteration chunk that each OpenMP thread will process, once the scheduling strategy has been chosen.

- `logger`. Field that gathers the configuration of the logging system.
  - `type`. As it has been discussed in Section 5.2 (see Figure 5.3), two options are available: "ConsoleLogger" and "FileLogger".
  - `logLevel`. This field is used to set the "`LogLevel`" enumeration class. It has two options: "DEBUG" and "INFO".

The **ODEIntegratorFactory** class is responsible to handle the runtime configuration. In Figure 5.12 is illustrated its UML diagram. This class implements a modified version of the widely known *Abstract Factory* creational design pattern[2]. Due to the fact that the library exploits the concepts of interfaces and polymorphism, the runtime configuration capability allows the user of the library to write a single client program and not having to modify the code if any configuration (for instance, the type of Reader, Writer, Integrator, etc) has to be changed.

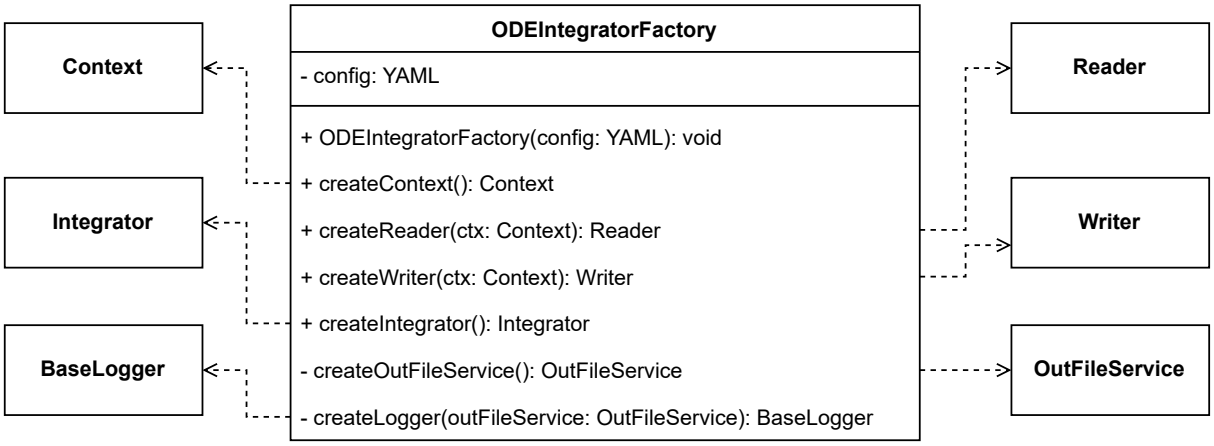


Figure 5.12: UML diagram of the ODEIntegratorFactory class.

[2]The ODEIntegratorFactory class does not follow exactly the Abstract Factory UML diagram but a modified version of it that fits better this use case.

Before moving to the validation section, Figure 5.13 illustrates a high-level view of the library with its relationships and external dependencies. At this point, the reader has a thorough understanding of how the **ODEIntegratorLibrary** works.

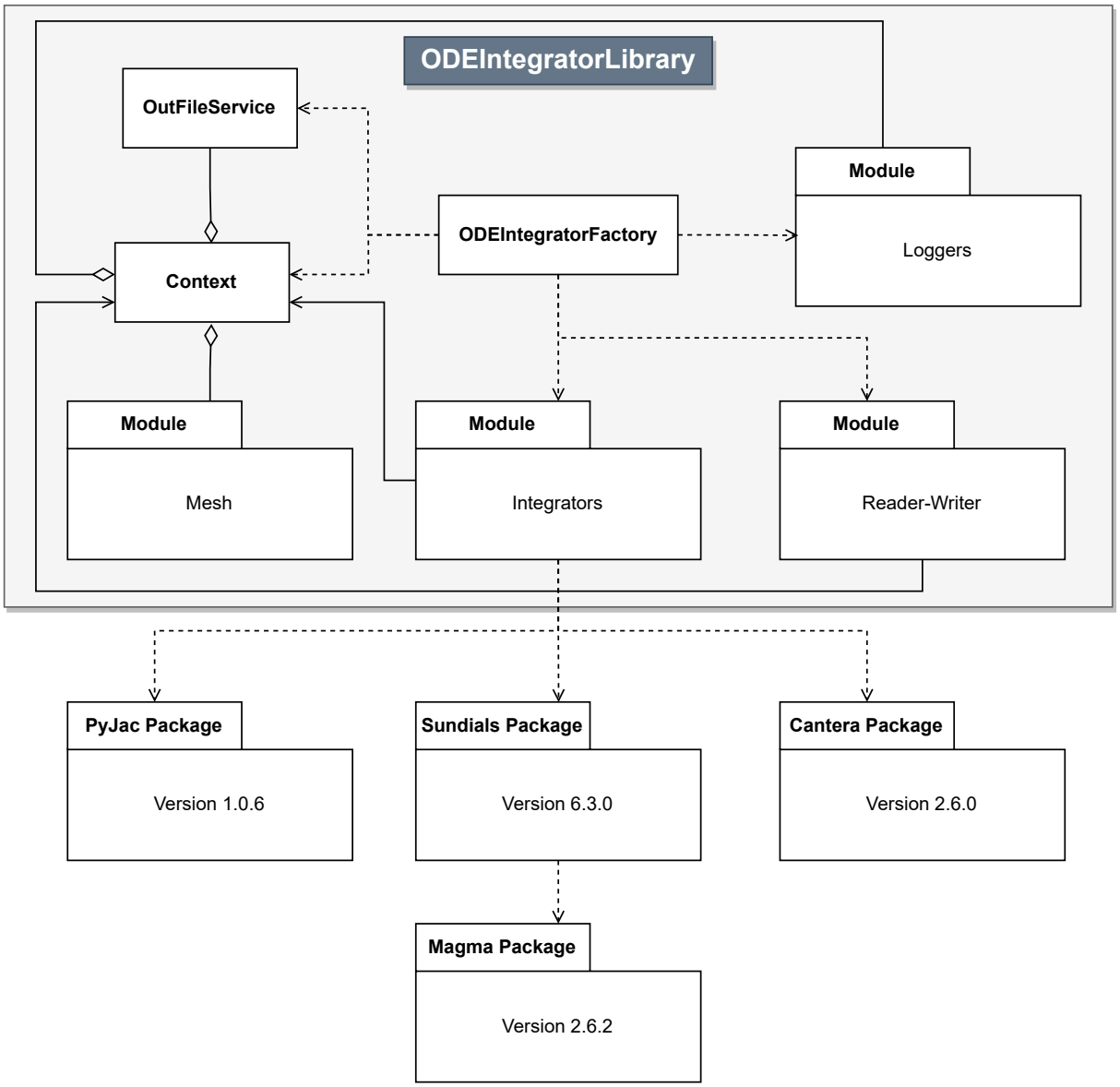


Figure 5.13: High-level UML diagram of the ODEIntegratorLibrary and its dependencies.

5.6 Unit testing

When a software project starts to get bigger, it becomes to be difficult to add or change functionality because there is the fear of breaking something by accident. Unit testing is a very common technique in the software development industry to help developers prevent adding bugs in the program. Moreover, adopting this philosophy, it enforces developers to write architectures that tend to be less coupled where each code piece adopts just one responsibility.

For the validation of every software subsystem of the current library, several test cases have been written. The GoogleTest testing framework has been adopted as a dependency for this task. GoogleTest provides an execution testing environment that can be easily extended to adapt it for many scenarios. Figure 5.14 shows the UML diagram of the testing architecture developed for the library validation. The highest superclass, **Test**, is part of the GoogleTest framework. It provides lots of handy features to create robust test cases. The methods "`SetUp(): void`" and "`TearDown(): void`" can be overridden in child classes. These methods are called internally by the GoogleTest engine when executing the tests.

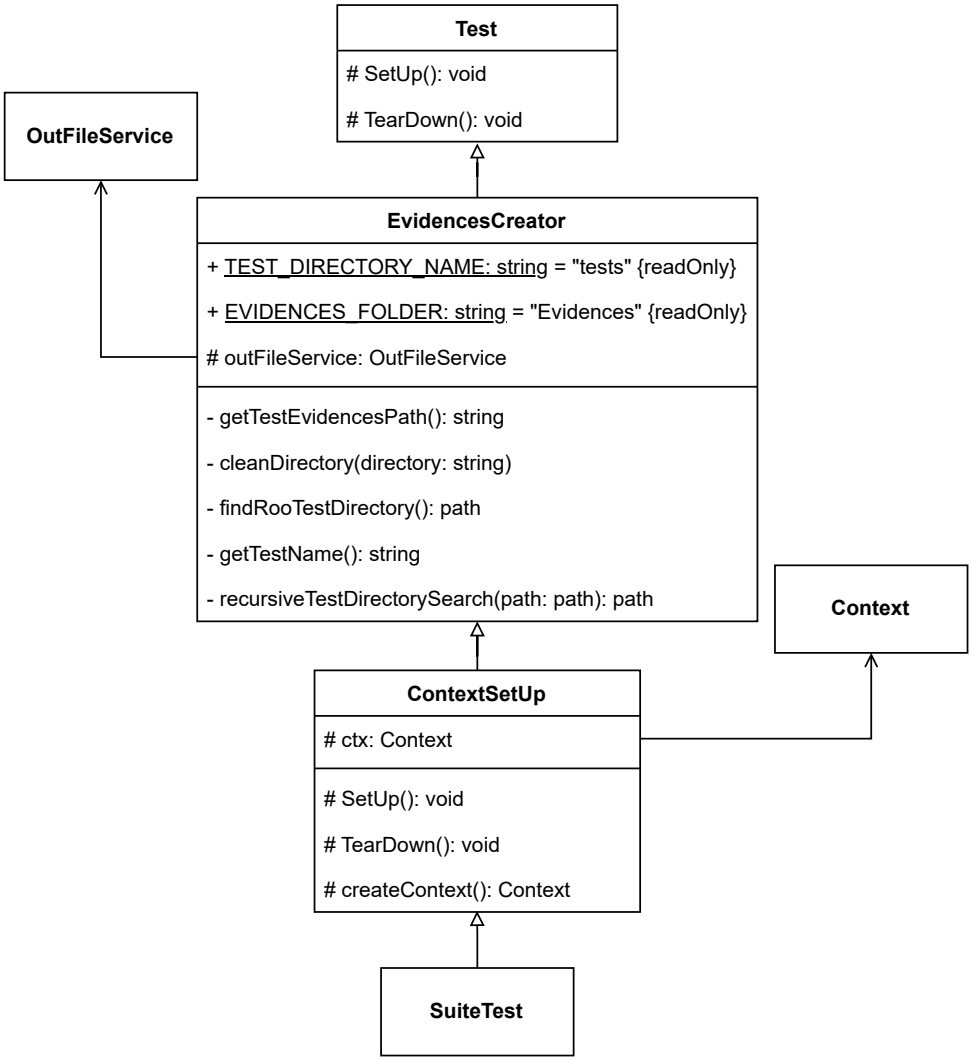


Figure 5.14: UML diagram of the uni testing framework developed for the library validation.

The **EvidenceCreator** class is an intermediate class whose objective is to create a specific evidence directory for the current running test. First, it searches recursively among the parents directories until it finds the "tests" directory. Once this directory is found, it creates a directory called "Evidences" at the "tests" level, if "Evidences" does not exist previously. The last step consists in using the Test superclass features, it queries the suite and test names and creates another directory, down the "Evidence" one, with those

names. Furthermore, it uses the latter directory path, the one with the suite and test names, to create an instance of the OutFileService class to save there all the file-base evidences.

ContextSetUp is another intermediate class, only used by integrator test suites because they all share some steps. Those steps are the Context creation, with their associated Mesh and BaseLogger instances, at the beginning of the test run, and clean the integrators at the end. To do so, this class overrides the "`SetUp(): void`" and "`TearDown(): void`" base methods. Lastly, the bottom classes of the hierarchy are the **SuiteTest** classes that contain their corresponding test cases.

The list of the suite and test cases that exists in this first version of the library is:

- Logger Suite:
  - "ConcurrencyManagementInFileLogger"
- Point Suite:
  - "FullBuild"
  - "MinBuild"
  - "BadBuild"
  - "BadConstruction"
  - "BadSpecies"
- Mesh Suite:
  - "AddPoints"
  - "Clear"
  - "OutOfIndexAccess"
  - "AddBadPoint"
  - "ChangeSpeciesVector"
- CsvReader Suite:
  - "ReadGoodFile"
  - "ReadBadFile"
  - "ReadNotExistingFile"
  - "ReadTxtInvalidFile"
- CsvWriter Suite:
  - "WriteNormalResults"
- CanteraIntegrator Suite:
  - "BasicIntegration"

  - "SerialVSOMP"
- CVodeIntegrator Suite:
  - "BasicIntegration"
  - "ValidationVSCantera"
  - "SerialVSOMP"
- CVodeIntegratorGPU Suite:
  - "ValidationVSCantera"
  - "FiniteDiffJacobian"

A total of 23 test cases contained in 8 test suites.

5.7 A C++ application example

This section is a step-by-step tutorial of how to use the library and take the most of it in a real C++ application. The section contains many pieces of code, but now that the reader has gone through the exhaustive explanation of the ODEIntegratorLibrary, it will be easy to follow because most of the function calls are part of the library's API, which should be familiar.

Main function

```
1  void runODEApplication(string configFileName)
2  {
3    auto application = ODEApplication(configFileName);
4    auto logger = application.getLogger();
5
6    logger->info("Starting ODEApplication...");
7
8    application.readData();
9    application.integrate();
10   application.writeResults();
11
12   logger->info("End of ODEApplication");
13 }
```

Listing 5.12: ODEApplication trigger function.

Listing 5.12 shows how in a few steps the highest level function performs the integration. The function makes use of the library through the ODEApplication class, whose UML diagram is found in the Figure 5.15. The input of this main function is a string that contains the path of the YAML configuration file. Then it creates an instance of the ODEApplication class, using the latter argument as a parameter in the class' constructor. For convenience, the ODEApplication instance allows the programmer to access the BaseLogger instance through its API. Finally, "runODEApplication" function calls

sequentially the methods "`readData()`", "`integrate()`" and "`writeResults()`", which their names are self-explanatory. Let's dive into those ODEApplication's methods.

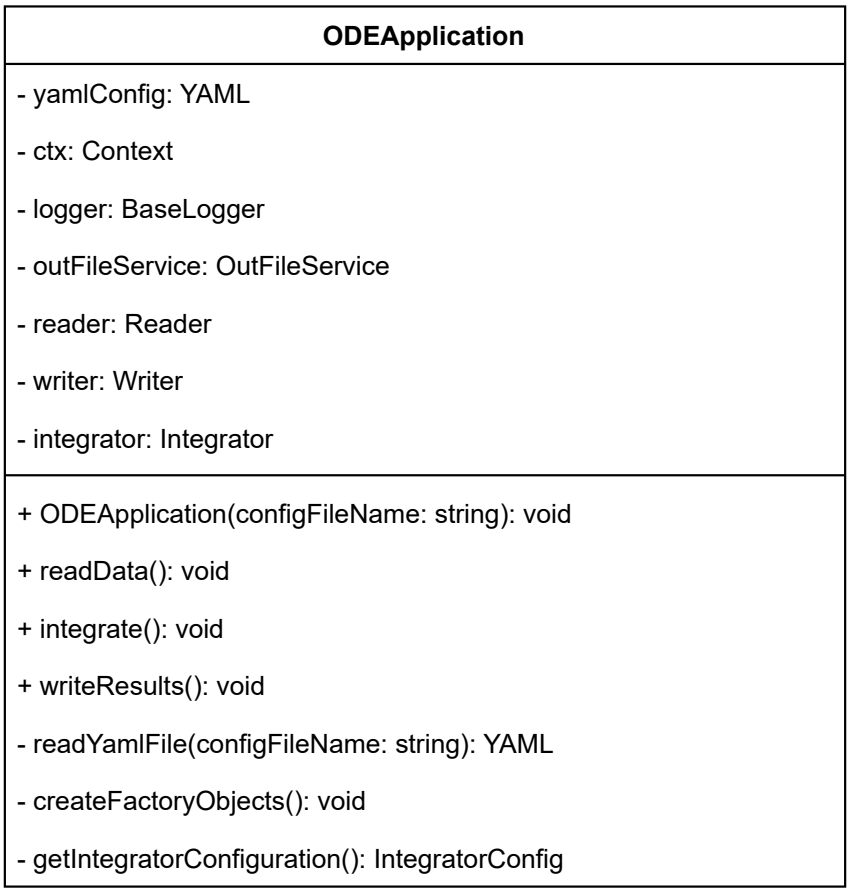


Figure 5.15: UML diagram of the ODEApplication class.

ODEApplication's Constructor

At the moment of creating an instance of the ODEApplication class, its constructor calls two functions (see Listing 5.13). The first one reads the YAML configuration file and keeps it internally as an instance property. The second function, line 5 of Listing 5.13, use the ODEIntegratorFactory class in order to instantiate the objects that correspond with the YAML specification. Listing 5.14 shows the factory in action.

```
1 ODEApplication(string configFileName)
2 {
3   this->yamlConfig = this->readYamlFile(configFileName);
4
5   this->createFactoryObjects();
6 }
```

Listing 5.13: ODEApplication's contructor method.

```
void createFactoryObjects()
{
  auto factory =
          make_unique<ODEIntegratorFactory>(this->yamlConfig);
  this->ctx = factory->createContext();
  this->logger = this->ctx->getLogger();
  this->outFileService = this->ctx->getOutFileService();
  this->reader = factory->createReader(this->ctx);
  this->integrator = factory->createIntegrator();
  this->writer = factory->createWriter(this->ctx);
}
```

Listing 5.14: ODEApplication's factory method.

Read and write functions

The read and write methods are just wrappers of the internal library API with a log message for debugging purposes. Notice that the "reader" and the "writer" instances are just calling the abstract method of their superclasses, so the code of Listings 5.15 and 5.16 does not need to be changed as long as the instances inherit from Reader and Writer abstract classes.

```
void readData()
{
  this->logger->info("Reading input data");
  this->reader->read();
}
```

Listing 5.15: ODEApplication's read method.

```
void writeResults()
{
  this->logger->info("Writting results");
  this->writer->write();
}
```

Listing 5.16: ODEApplication's write method.

Integrate function

Finally, the "`integrate()`" method consists in taking from the YAML object the configuration parameters, constructing the IntegratorConfig object, and use that latter object, plus the integration time range, to call the three integrator methods of the library's API: "`init(...)`", "`integrate(...)`" and "`clean()`". Again, notice that any type of integrator that inherits from the Integrator abstract class will be able to fit in this function calls without breaking.

```
void integrate()
{
  this->logger->info("Integrating systems");

  IntegratorConfig integratorConfig =
                    this->getIntegratorConfiguration();
  double dt = yamlConfig["integrator"]["dt"].as<double>();

  this->integrator->init(this->ctx, integratorConfig);
  this->integrator->integrate(dt);
  this->integrator->clean();
}
```

Listing 5.17: ODEApplication's integrate method.

```
IntegratorConfig getIntegratorConfiguration()
{
  IntegratorConfig integratorConfig;
  integratorConfig.reltol = this->yamlConfig["reltol"]
  integratorConfig.abstol = this->yamlConfig["abstol"]
  integratorConfig.mechanism = this->yamlConfig["mechanism"]
  integratorConfig.jacobianStrategy =
                              this->yamlConfig["jacobianStrategy"]
  integratorConfig.pressure = this->yamlConfig["pressure"]
  integratorConfig.ompConfig = this->yamlConfig["omp"];
  return integratorConfig;
}
```

Listing 5.18: ODEApplication's method that queries the integrator configuration.

5.8 High-level Python application for data collection

Before concluding this chapter, some brief comments about a high-level application will be given in the current section. Throughout the chapter we have seen that the library has many options to configure. From the input data, through the integrator parameters to the type of concrete instances, such as logger, integrator, etc. Analyzing all possible configurations to gain insight into library performance and scalability can be a difficult task. In order to solve this problem, a high-level Python application has been developed with the goal of collecting configuration data in an automated way and saving this data into a relational database. So, the Python application is just a wrapper of the C++ application presented in the previous section.

Figure 5.16 illustrates the entity-relationship diagram of the local database developed for archiving runtime configurations. Chapter 6 presents results that come from this database after some processing. The technology used for the database is **SQLite** that is a Structured Query Language (SQL) database engine written in C language [53]. SQLite

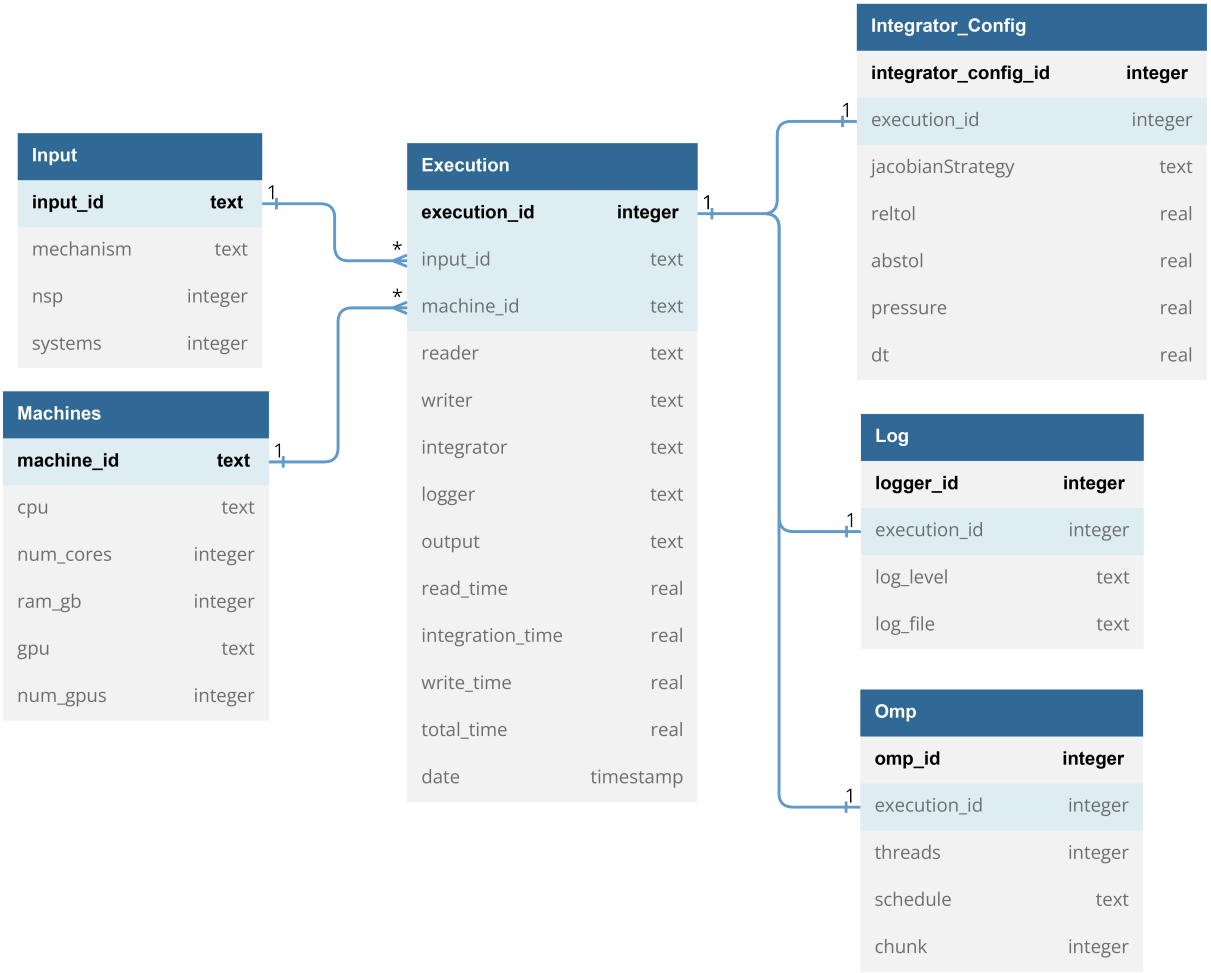


Figure 5.16: Database's diagram of the high-level Python application.

presents a Python interface, **sqlite3**, that comes with the standard package installation. The application database consists of the following tables:

- **Input table**. This is where the different CSV input file parameters are stored. It has a one-to-many relationship with the Execution table, because many runtime configurations can have the same input file.
- **Machines table**. It saves the information of the compute node where the execution has been performed. As well as the previous table, it has a one-to-many relationship with the Execution table.
- **Execution table**. This table stores the information of the type of instances, execution time and date timestamp.
- **IntegratorConfig table**. As its name suggests, this table stores the integrator parameters used for the integration of the ODE systems. It has a one-to-one relationship with the Execution table, because each runtime configuration can have only one IntegratorConfig row associated.
- **Log table**. It has the information related to the logging system, such as the

LogLevel option chosen and the log file content, if any. As well as the previous table, it has a one-to-one relationship with the Execution table.

- **Omp table**. This table only receives new entries when a parallel OpenMP integrator is chosen, such as CanteraIntegratorOMP or CVodeIntegratorOMP. It stores the OpenMP runtime configuration and it has a one-to-one relationship with the Execution table.

Chapter 6

Results

6.1 Analysis of parallel algorithms

The current chapter presents the results regarding computing performance. The objective of parallel computing is reducing the compute time of an algorithm. This involves the concept of maximizing the degree of use of a parallel computer, which usually answer the question of how the algorithm scales with the problem size or/and the number of processor units. To analyze the performance of a parallel algorithm, two types of parameters are used:

- **Absolute** parameters. These parameters give information about the real cost of an algorithm and enable the calculation of relative parameters. The most important parameters are the serial and parallel execution times.
- **Relative** parameters. These parameters enable comparing the performance of a parallel algorithm among multiple implementations, which, among others, give information about the degree of use of the parallel computer.

$$S(n,p) = \frac{t(n)}{t(n,p)} \tag{6.1}$$

$$E(n,p) = \frac{S(n,p)}{p} \tag{6.2}$$

Eqs.(6.1) and (6.2) are the formulas to calculate the speedup, $S(n,p)$, and the efficiency, $E(n,p)$, of a parallel algorithm, where $t(n)$ is the serial execution time to solve the problem of size n and $t(n,p)$ is the parallel execution time using p threads/processes. However, those formulas give the performance analysis of what is known as *strong scaling*, which is how the solution time varies with the number of processors for a **fixed total problem size**. Another scalability analysis is the *weak scaling* that consists in study how the solution time varies with the number of processors for a **fixed problem size per processor**. Ideally, perfect weak scalability is achieved when the time remains constant when the number of processor and problem size scale proportionally[1]. Eq.(6.3) shows how

[1]This is only true if the computational cost of the algorithm scales linearly with the problem size, otherwise, the analysis is more complex. For instance, when performing a weak scalability test for different

to calculate the *weak speedup*. $C(n,p)$ is the theoretical parallel cost, n_0 is the problem size that each thread/process has to solve and k is the proportional constant with which threads/processes and problem size grows.

$$S(n,p)_{weak} = \frac{C(n,p)}{t(n,p)} = \frac{k\,p_0\,t(n_0)}{t(k\,n_0, k\,p_0)} \tag{6.3}$$

The following sections show the obtained performance results of the ODEIntegratorLibrary, using execution time, speedup and efficiency parameters. The computation has been performed with the **hapy.dsic.upv.es** machine that pertains to the *Departamento de Sistemas Informáticos y Computación*, whose specifications appear in the Table 6.1.

Machine Specifications	
CPU	Intel(R) Core(TM) i9-10940X CPU @ 3.30GHz
Cores	14
Logical processors	28
RAM capacity	15 GB
GPU	NVIDIA RTX A2000 12GB
Number of GPUs	1

Table 6.1: **hapy.dsic.upv.es**'s machine specifications.

6.2 Serial performance

6.2.1 Cantera and standalone CVODE

Figure 6.1 shows the integration time taken by the serial version of Cantera and CVODE integrators of the ODEIntegratorLibrary. It can be appreciated that the serial CVodeIntegrator outperforms Cantera's version. As described in the previous chapter, CanteraIntegrator uses the CVODE solver under the hood, which indicates that the difference in performance is due to the overhead generated by the multiple layers that Cantera has. Cantera calculates other related chemical properties that are not needed for the current application. Therefore, even though Cantera gives a lot of functionality and information about the thermochemical state of the solution, this represents a performance penalty that has no beneficial outcome because those extra functionalities are not needed.

Furthermore, Figure 6.2 gives more insights about the results from Figure 6.1. The speedup achieved when using CVodeIntegrator, instead of CanteraIntegrator, is almost a x5 and it seems to be independent of the number of systems to solve. This means that the overhead generated by Cantera does not grow with the problem size, but it is constant.

combustion mechanisms, in order to maintain the same load on each thread, one must take into account not only the number of systems to solve but also the cost of each mechanism's algorithm, which generally scales quadratically or cubically with the number of species.

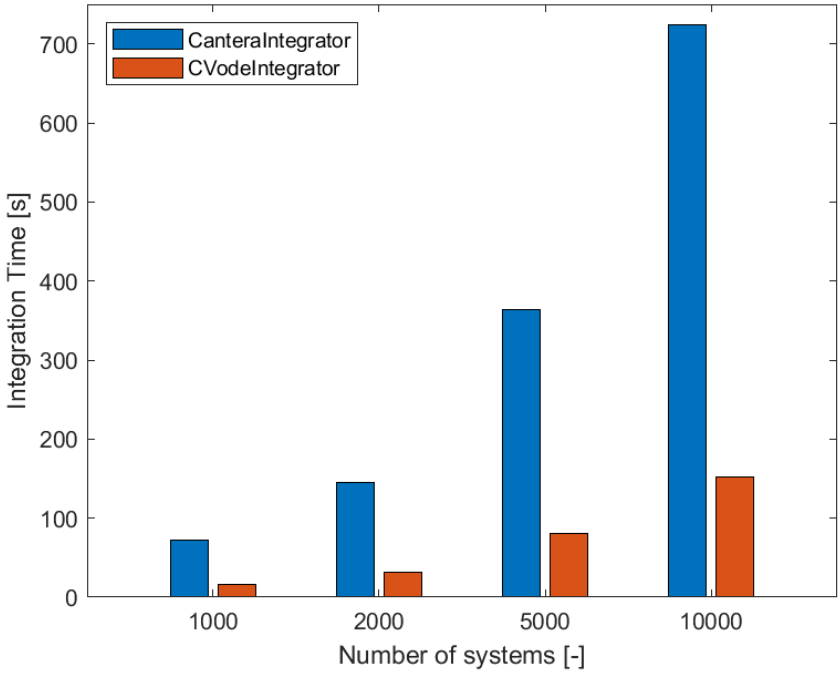


Figure 6.1: Integration time of CanteraIntegrator and CVodeIntegrator when solving various number of systems.

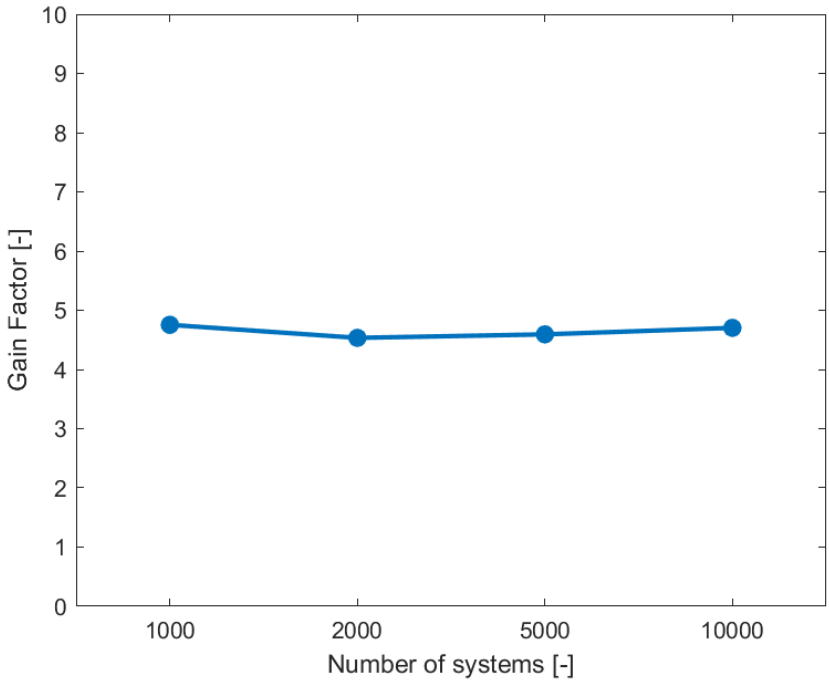


Figure 6.2: CVodeIntegrator's integration time over CanteraIntegrator's from the data of Figure 6.1.

For these results the GRI3.0 combustion mechanism [54] has been used, with the integrator configuration of Table 6.2[2].

Integrator Configuration	
JacobianStrategy	analytical
reltol [-]	1.0e-6
abstol [-]	1.0e-10
pressure [Pa]	15 GB
dt [s]	1.0e-3

Table 6.2: Integrator configuration for serial integration with CanteraIntegrator and CVodeIntegrator.

6.2.2 Jacobian matrix calculation strategy

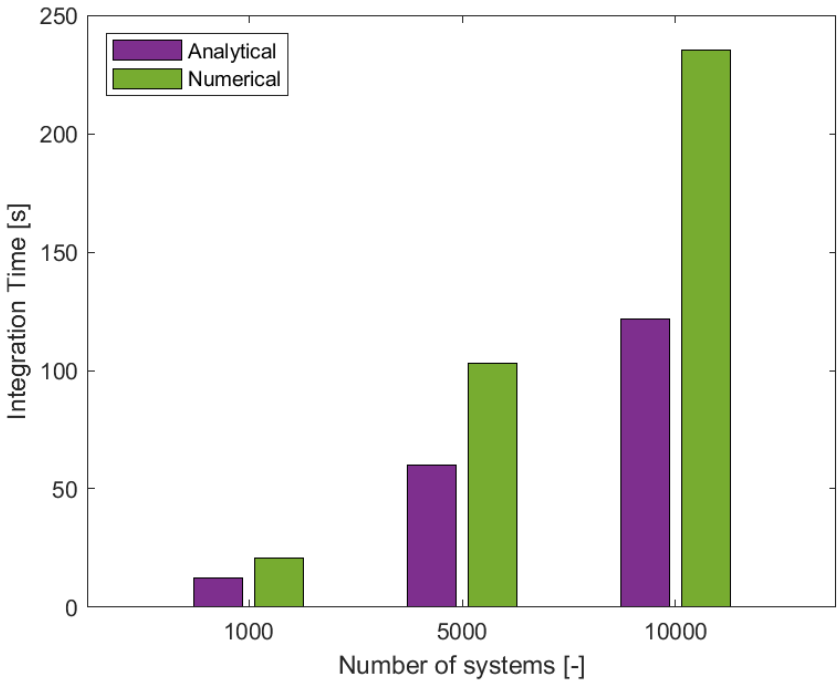


Figure 6.3: CVodeIntegrator's integration times with analytical and numerical approaches for the Jacobian matrix evaluation.

Continuing the analysis with the CVodeIntegrator, Figure 6.3 illustrates the integration times taken to solve different number of systems with two Jacobian matrix strategies. The analytical evaluation uses the functions provided by pyJac package. The numerical

[2]The JacobianStrategy option does not affect CanteraIntegrator, Cantera's library manage the Jacobian matrix evaluation in its way, without giving option to the user.

calculation is handled internally by CVODE solver with finite difference approximation. For every number of systems to solve, the analytical evaluation presents better performance due to the fact that finite difference approaches require more function evaluations to construct the matrix. In Figure 6.4, it can be seen the integration times with a numerical over analytical strategy, resulting the latter strategy approximately two times faster (constant for the number of systems and GRI3.0 combustion mechanism).

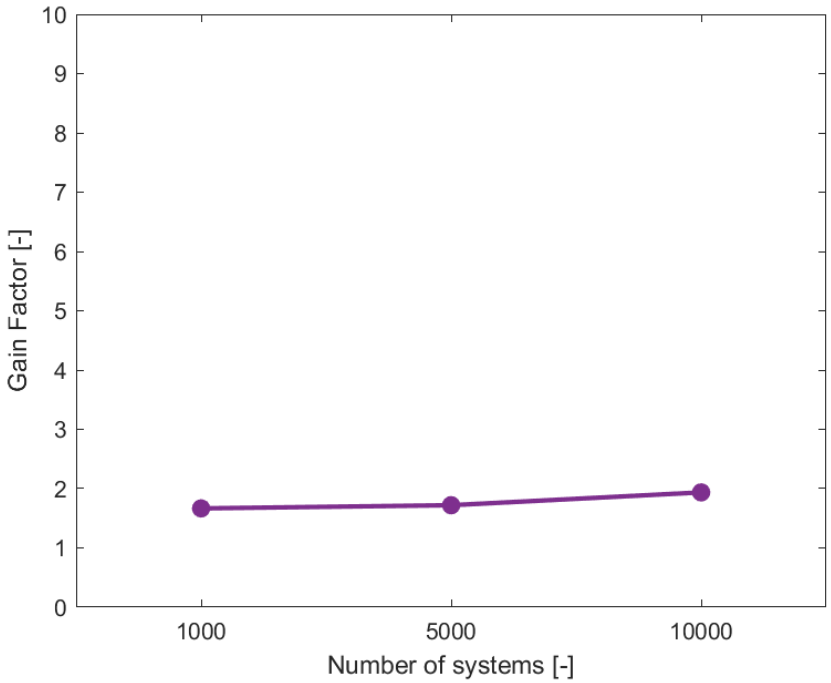


Figure 6.4: CVodeIntegrator's integration time with numerical over analytical Jacobian matrix evaluation strategy.

From now on, an analytical Jacobian matrix evaluation will be the default configuration for CVODE integrators.

6.3 OpenMP runtime configuration

This section shows the results of the analysis carried out looking for the best OpenMP's schedule configuration. There are two parameters to configure, the schedule strategy and the size of the chunk when `for` loops' iterations are distributed among OpenMP's threads. Figure 6.5 illustrates the integration times when solving 10000 ODE systems with the CanteraIntegratorOMP, using 14 OpenMP's threads. It is easy to appreciate that the smaller the chunk size, the better performance is achieved. This phenomenon indicates load imbalance among systems, which means that the times to solve individual ODE systems are different. Therefore, with smaller chunk sizes, the overall load is balanced among the threads. That is the reason why the schedule strategy has almost no effect in small chunk sizes, but when the chunk size is big, strategies such as "dynamic" or "guided" work better because threads dynamically request work. When using "static"

schedule for big chunk sizes, the load is distributed before starting computation, so if there are load imbalances among iterations, it cannot be compensated.

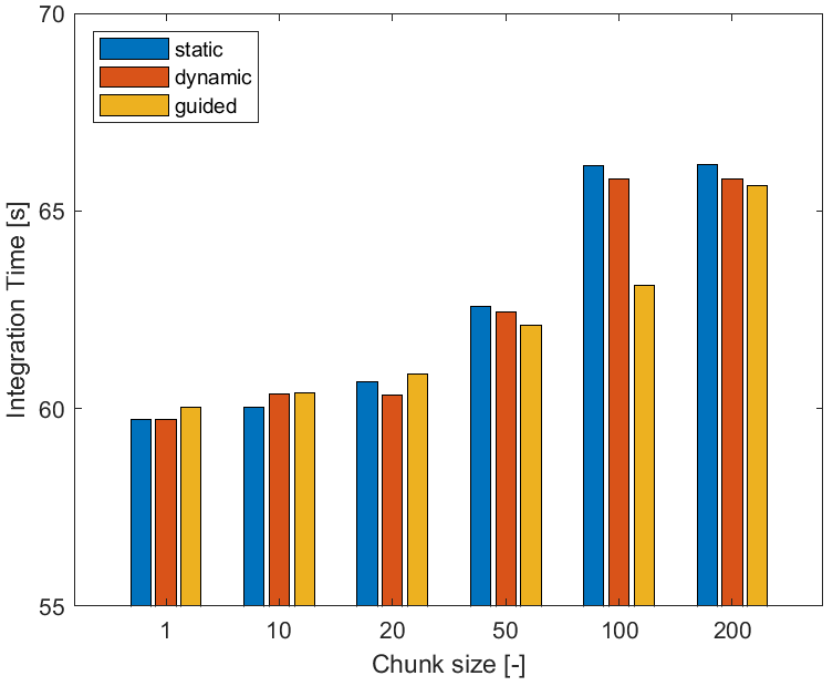


Figure 6.5: Integration time of CanteraIntegratorOMP for different schedule configuration when solving 10000 systems.

Figure 6.6 indicates that the pattern discussed above is repeated for the CVodeIntegratorOMP, but obviously with different integration times, as we have seen in the previous section that the CVodeIntegrator is almost five times faster than Cantera's. So, again, for CVodeIntegratorOMP smaller chunk sizes performs better. The difference in the integration times with different chunk sizes seems to be more pronounced for the CanteraIntegratorOMP case, but Figures 6.7 and 6.8 prove this thinking wrong. These Figures illustrate the same data but from a dimensionless perspective. In Figure 6.7, the Y-axis represents the integration time over the minimum integration time with respect to the data of Figure 6.5, and in the same way for Figure 6.8. So, notice that both Figures, 6.7 and 6.8, are almost identical (same Y-axis range) but it represents the results from CanteraIntegratorOMP and CVodeIntegratorOMP. This illustrates that the performance penalty in the worst case, i.e., 200 of chunk size with "static" schedule, is around a 10% for both integrators. In summary, the best OpenMP's schedule configuration is:

- Schedule: "dynamic".
- Chunk size: less than 10.

However, the penalty in performance of choosing another scheduling strategy is low.

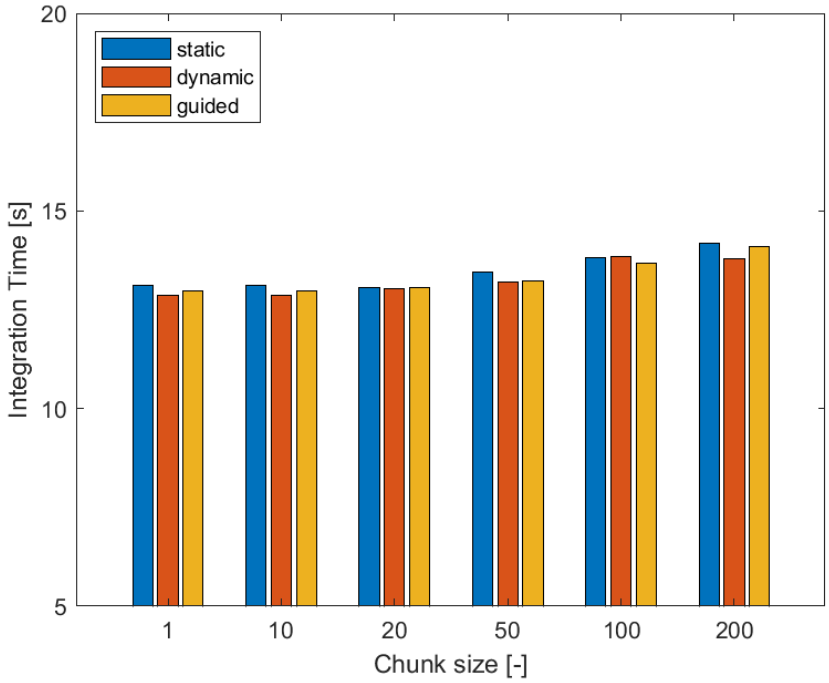


Figure 6.6: Integration time of CVodeIntegratorOMP for different schedule configuration when solving 10000 systems.

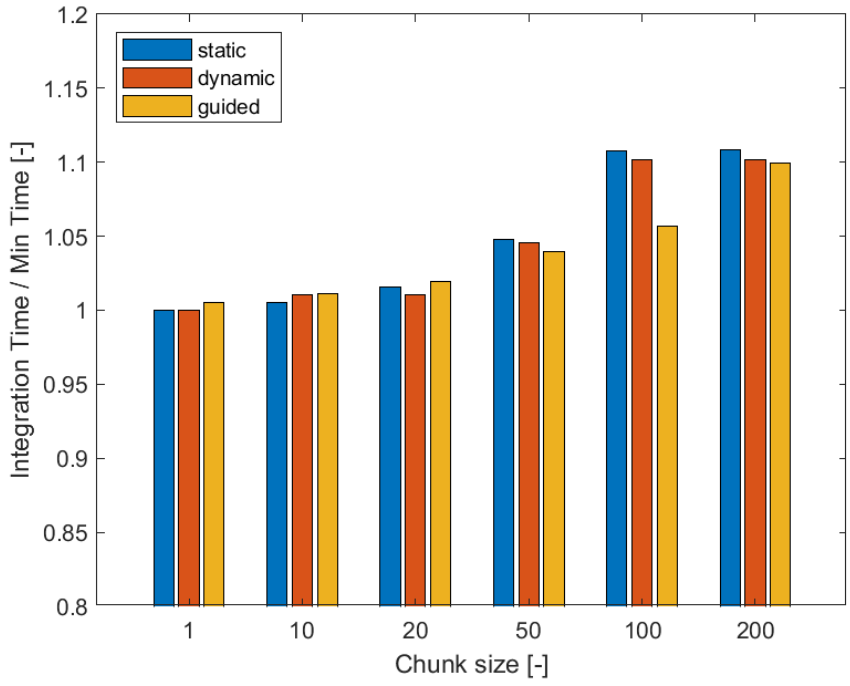


Figure 6.7: Integration time over the minimum time (among times of Figure 6.5) of CanteraIntegratorOMP for different schedule configuration when solving 10000 systems.

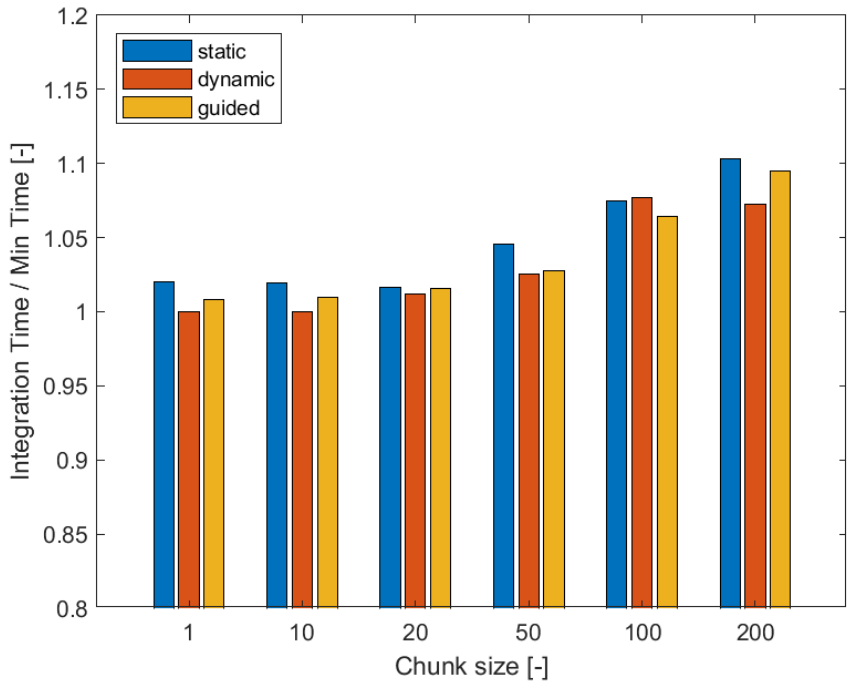


Figure 6.8: Integration time over the minimum time (among times of Figure 6.5) of CanteraIntegratorOMP for different schedule configuration when solving 10000 systems.

6.4 OpenMP Integrators' scalability

Now that we have discussed the best OpenMP's schedule strategy, we can analyze the scalability of OpenMP's integrators. As explained in the beginning of the chapter, two approaches have been taken for this task:

- **Strong scalability**
- **Weak scalability**

The integrator configuration used is the same as for the serial analysis (see Table 6.2), and Table 6.3 shows the OpenMP schedule configuration.

OpenMP Configuration	
Schedule	dynamic
Chunk size	1

Table 6.3: OpenMP's schedule configuration.

6.4.1 Strong scalability

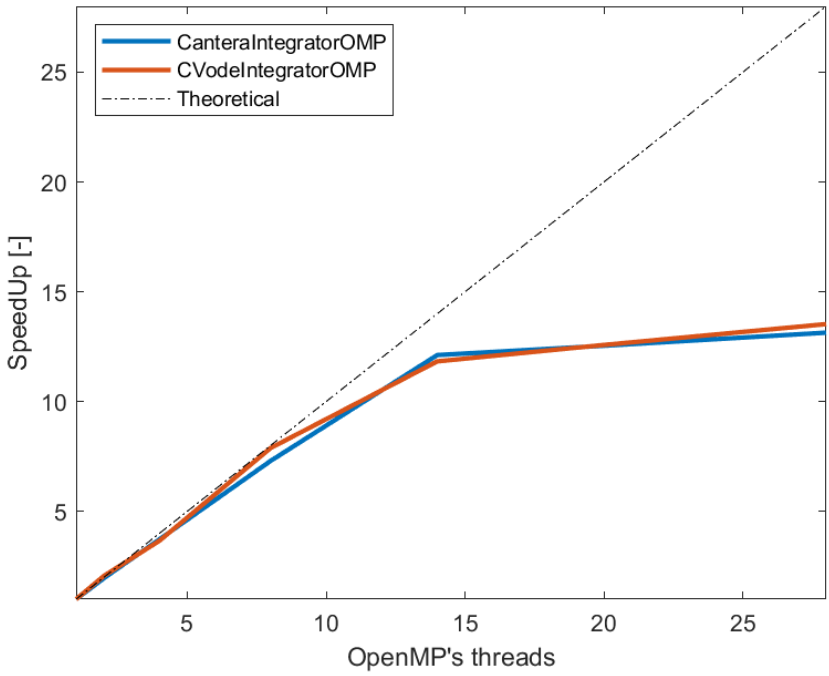


Figure 6.9: Strong speedup of CanteraIntegratorOMP and CVodeIntegratorOMP when solving 10000 systems.

For the strong scalability analysis, 10000 ODE systems has been chosen as the fixed problem size. Figure 6.9 shows the speedup of both, CanteraIntegratorOMP and CVodeIntegratorOMP, integrators. Both integrators present almost identical results, which are

really promising for the first half of the plot. For less than 15 OpenMP's threads the speedup is very close to the theoretical maximum. According to Amdahl's law [55], this indicates that the intrinsically serial part of the algorithm is relatively small compared with the part that can be parallelized. This is not only true for the CVODE solver but for the high-level layers that Cantera has above it. On the second half of the plot, above 14 OpenMP's threads, the speedup suffers a decrease on its rate of growth. This is because the number of physical cores that the **hapy.dsic.upv.es** machine has, is 14. Then, when configuring OpenMP with more threads than the latter number, the thread scheduler is using the hyperthreading capability that Intel's processors have. Intel's processor manage two logical processor units per physical processor, so from the software perspective it may seem that the machine has more capability, but under the hood those two logical processors have to share the same hardware. In applications with low compute intensity, for instance, a web server, this characteristic increases the maximum clients that the server can handle at once. However, for applications with high compute intensity, such as the present one, this does not perform as expected.

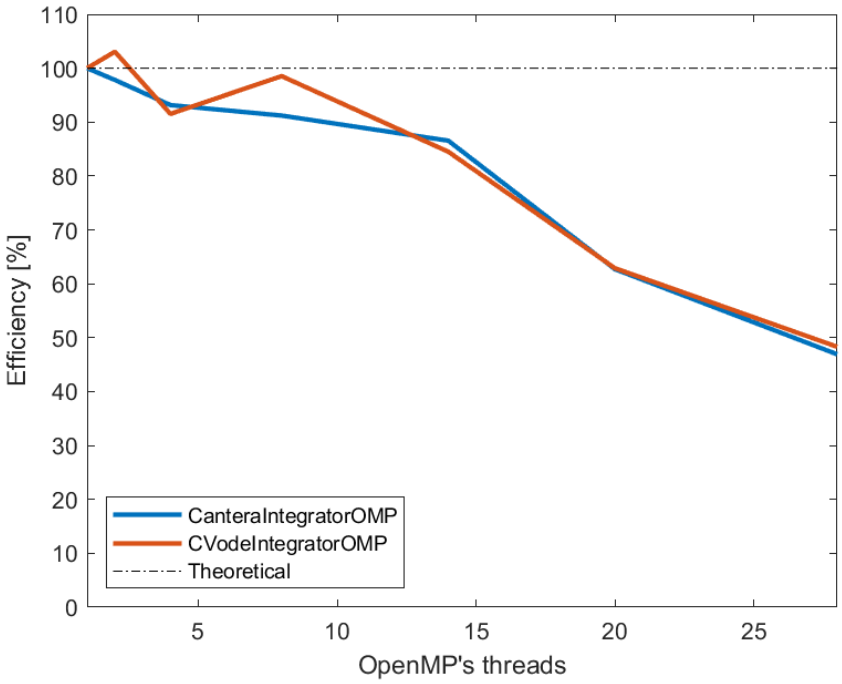


Figure 6.10: Strong parallel efficiency of CanteraIntegratorOMP and CVodeIntegratorOMP when solving 10000.

Figure 6.10 confirms the deductions described above. Here, the parallel efficiency is around 90% for less than 15 threads. But, it drops to 50% when hyperthreading comes into play.

6.4.2 Weak scalability

Figures 6.11 and 6.12 shows the results of speedup and efficiency of the weak scalability test. Here, the number of systems that each thread solves is 1000. Therefore, in the Eq.(6.3), $n_0 = 1000$, $p_0 = 1$ and $k = 1, 2, \ldots, n$.

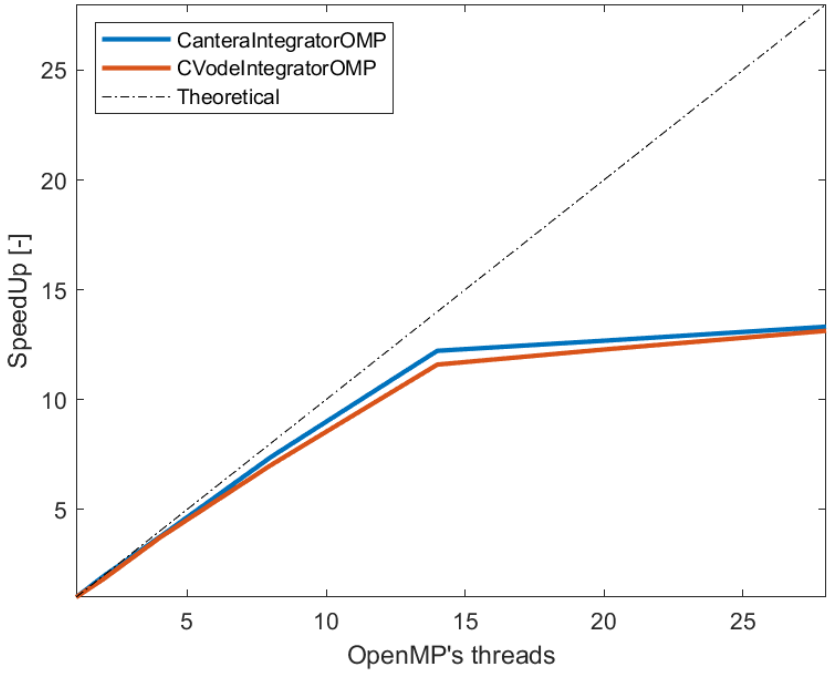


Figure 6.11: Weak speedup of CanteraIntegratorOMP and CVodeIntegratorOMP to a fixed load of 1000 systems per thread.

The *weak efficiency* is calculated using the Eq.(6.2) but using the *weak speedup* in the numerator. It can be appreciated that the weak scalability of the CanteraIntegratorOMP and CVodeIntegratorOMP have the same trend as in the strong scalability, meaning that the time to solve bigger problems, increasing the number of threads proportionally, is almost the same as solving in serial. The performance starts to drop when the number of threads is greater than the number of physical cores of the machine, as well as we have concluded in the strong scalability test. In summary, OpenMP's integrators of the ODEIntegratorLibrary show good scalability properties both for weak and strong scalability tests when the number of threads are equal or less than the number of physical processors.

From another perspective, we can assume that the integration time in the hyperthreading regions corresponds to the full capability of the physical cores, which means that a speedup of 13 with 28 OpenMP's threads actually is a x13 for 14 cores working in parallel, an almost perfect parallelization.

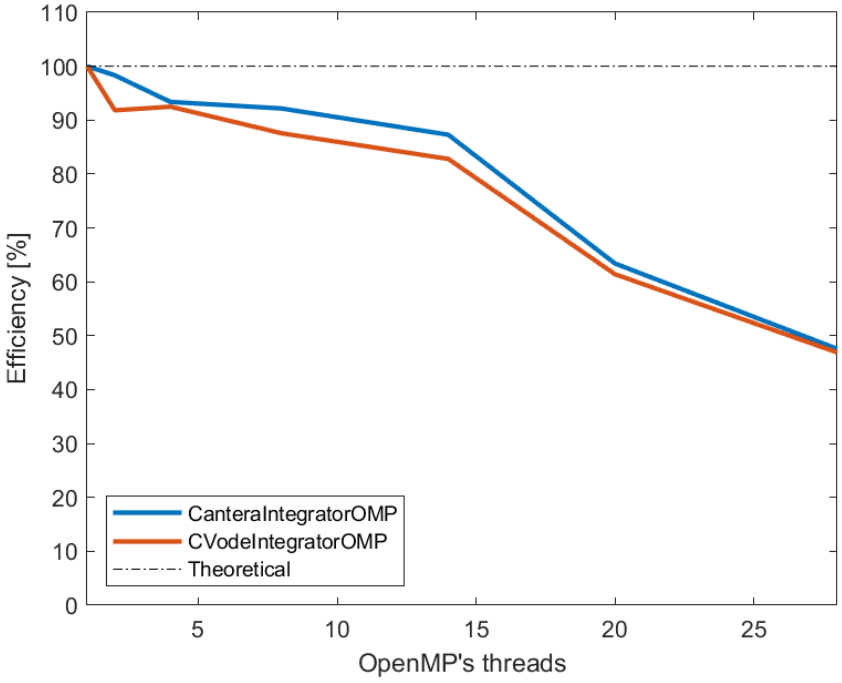


Figure 6.12: Weak parallel efficiency of CanteraIntegratorOMP and CVodeIntegratorOMP to a fixed load of 1000 systems per thread.

6.5 GPU integrator

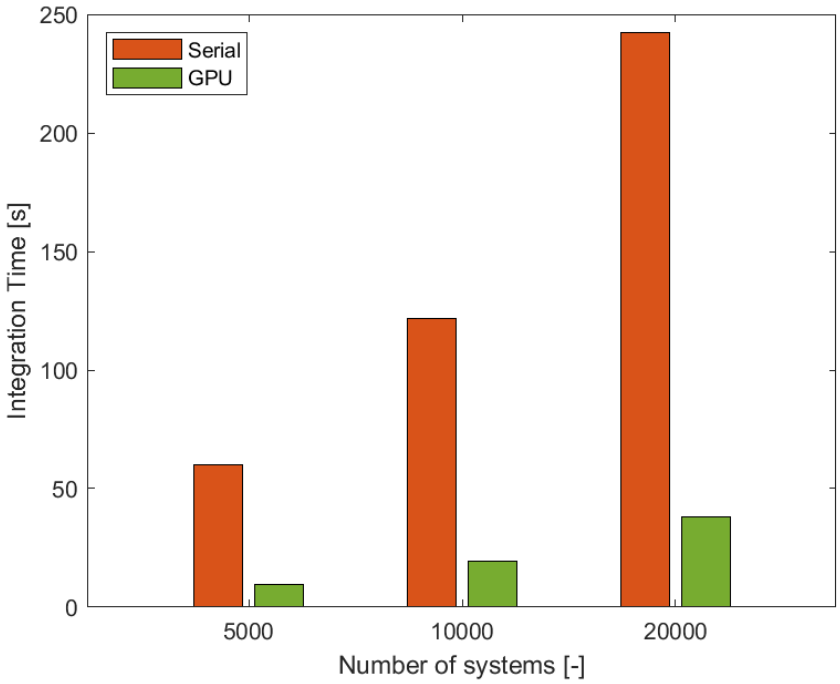


Figure 6.13: Integration times of serial and GPU implementations of CVODE integrators.

Figure 6.13 displays the integration times of the serial and GPU CVodeIntegrators for different number of systems. The integrator configuration is the same as used before, see Table 6.2, and the combustion mechanism is the GRI3.0. Figure 6.14 shows that, for the number of systems of the Figure 6.13, the GPU implementation is six times faster than the serial integration. From the results of the previous OpenMP's analysis, it can be confirmed that OpenMP's integrators have better performance while the number of physical cores is greater than six.

There are multiple reasons of why the GPU does not outperform OpenMP's implementations. First of all, pyJac and CVODE do not share the same vector and matrix layout for GPU data structures. Therefore, for every pyJac's kernel call, CVODE's GPU data must be copied to the pyJac's layout, and the other way around before finishing the kernel, as shown in Listing 6.1.

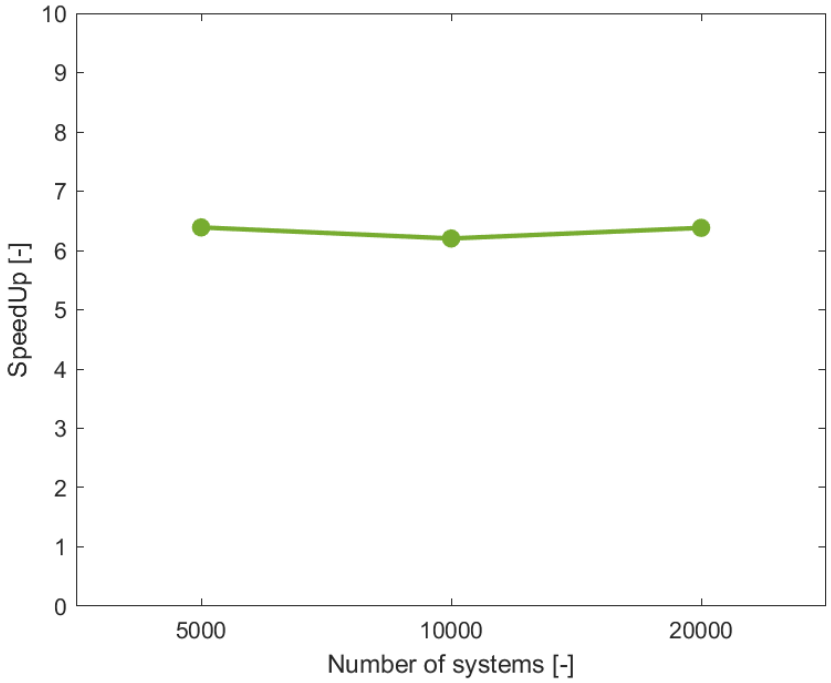


Figure 6.14: Serial integration times over GPU's for different number of systems.

```
__global__ void kernel_dydt(...)
{
    if (T_ID < systemsToProcess)
    {
        ...
        // Reorder data for PyJac
        sun_to_pyjac_Y(ySun, yPy);

        dydt(t, P, yPy, dyPy, d_mem);

        // Reorder data back to Sundials
        pyjac_to_sun_Y(dyPy, dySun);
    }
}

__device__ void sun_to_pyjac_Y(double *ySun, double *yPy)
{
    int threadID = threadIdx.x + blockIdx.x * blockDim.x;
    // PyJac index -> #define INDEX(i) (T_ID + (i) * GRID_DIM)
    for (int i = 0; i < NSP; i++)
    {
        yPy[INDEX(i)] = ySun[threadID * NSP + i];
    }
}
```

Listing 6.1: dy/dt CUDA kernel.

Secondly, pyJac's CUDA parallel approach is "'per-thread", rather than "per-block" [47]. A "per-thread" GPU parallelization model means that each GPU thread calculates an entire system, both function and matrix evaluations, while in "per-block" approaches, threads in a block cooperate for the evaluation. The former approach is similar to the OpenMP's, i.e., coarse-grained parallelism. This may cause thread divergence among CUDA "warps" (also known as the Single Instruction Multiple Thread (SIMT) CUDA model) [41].

Lastly, Sundials' current version (6.3.0) explicitly warns in its documentation that the Sundials-Magma modules are still experimental, so, we must wait for future stable versions in order to properly analyze the performance. In summary, the current GPU implementation merits further investigation for the purpose of enhancing the kernels. Next chapter proposes multiple lines of research for this case.

6.5.1 Combustion mechanism influence

Before moving on to the conclusion chapter, one last analysis has been performed, which is the influence of the combustion mechanism on the CVodeIntegrators. Figure 6.15 illustrates integration times using the combustion mechanisms of the Table 6.4 (notice the Y-axis logarithmic scale). It can be appreciated that the mechanism strongly determines the time to solution.

Mechanism	**Number of species**
H2O2	9
GRI3.0	53
ESTiMatE	214

Table 6.4: Combustion mechanisms and its associated number of species.

The computational cost scales almost cubically with the number of species. Even though, both OpenMP (using 14 threads) and GPU's integrators perform around one order of magnitude faster than the serial version, being OpenMP's the best implementation. However, results from the Figure 6.15 are very promising for the GPU. There, the GPU seems to be less performant for small mechanism sizes, but it really scales for large ones, approximating to the same performance that OpenMP presents for the *ESTiMatE* mechanism. Notice that OpenMP's performance degrades for large detailed chemical mechanisms, showing an inflection point where GPU integrator comes into play to replace the OpenMP version.

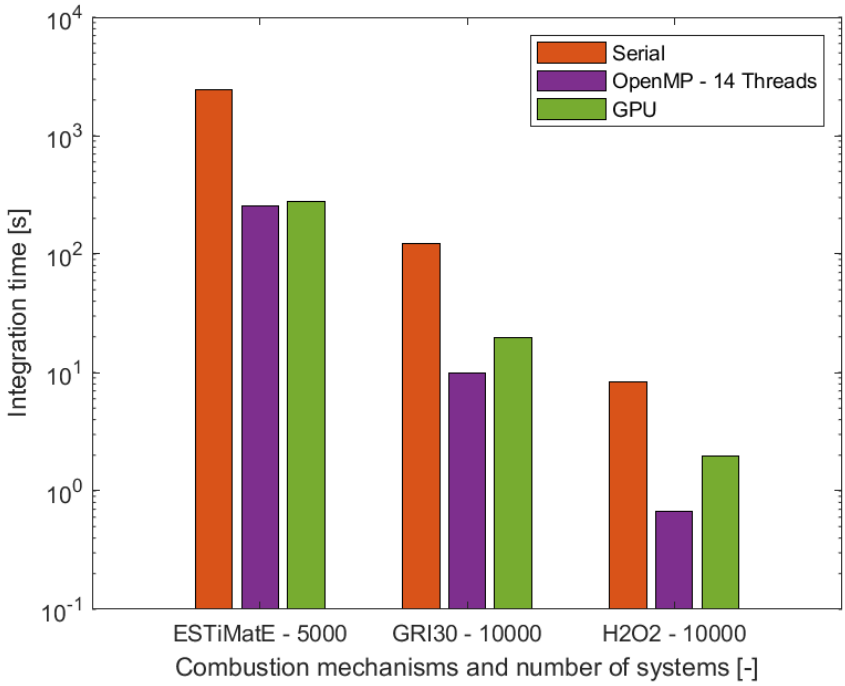


Figure 6.15: Integration times for different combustion mechanisms.

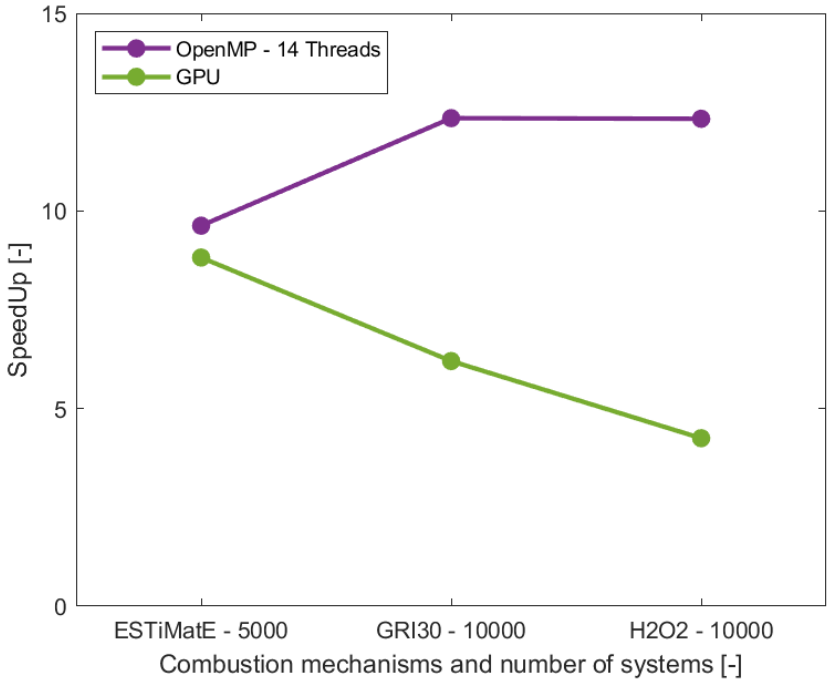


Figure 6.16: Speedup of OpenMP and GPU's integrators for different combustion mechanisms.

Chapter 7

Conclusions and future work

7.1 Conclusions

In the present work we have gone through the physics phenomena involved in the simulation of reacting flows, along with the mathematical splitting scheme that permits the integration of the chemical source term, that appears as a result of the combustion of air-fuel mixing. The equation that drives to the calculation of the temperature and species is identified as an Ordinary Differential Equation system. In Chapter 3 an exhaustive analysis of how to solve these ODE systems, when the stiffness characteristic is involved, has been performed. Then, in Chapter 4 we have described the state of the art of the High Performance Computing tools and techniques, focusing on the ones that enable parallelization at node-level with heterogeneous architectures. With all this context, we have developed a library that speeds up the integration time of the temperature and species that form the ODE system. The integrators that compound the library present different parallelization approaches:

- Serial integrators (no parallelization).
- Fork-join thread parallelization model using OpenMP.
- NVIDIA-GPU parallelization via CUDA language.

To support all these types of integrators, the library gathers multiple coupled systems that work together in order to provide the essential functionality that the integrators need. Those systems are in charge of providing functionality such as the data ingestion, logging, internal data management, working with the OS' file system and more. From the Chapters 5 and 6 we can enumerate the following conclusions:

- We have investigated the open-source ODE solvers that are available and, with those, we have configured and validated a base case using the GRI3.0 combustion mechanism with a constant pressure condition in a single processor unit.
- We have identified the parallelization opportunity when integrating multiples ODE systems.
- A working configuration of the Sundials-CVODE solver with the pyJac package has

been successfully performed.

- Two OpenMP's fully functional integrators have been developed that present great strong and weak scalability characteristics.
- A first GPU parallelized version of the CVODE integrator has been developed achieving moderate acceleration for small combustion mechanism, but that seems to outperform the OpenMP's versions for detailed combustion reactions.
- We have found the best OpenMP schedule configuration for the CPU parallel integrators.
- We have demonstrated that the analytical Jacobian matrix evaluation performs two times faster than using finite difference approximations for the serial and OpenMP integrators.
- The combustion mechanism strongly determines the time to solution where the computational cost scales cubically with the mechanism size.
- We have developed a high-level Python library with a SQLite database to automate the data collection pipeline for parametric studies.

7.2 Future work

The current work introduces the state of the art of the combustion phenomena and its associated mathematical problem, as well as the HPC tools to speed up its calculation. Therefore, this thesis can be used as the entry point for future students who want to enter this line of research. Still, many things deserve further investigation. Now that the basic infrastructure has been established, there is an opportunity to work in the following areas:

- The CVodeIntegratorGPU requires further research. On the one hand, The current kernel implementations can be tuned to increase the performance, specially the GPU finite difference kernel for the Jacobian matrix evaluation, whose current implementation is not functional. With the proper use of the NVIDIA's profiling tools this can be achieved. On the other hand, the GPU versions of pyJac and CVODE do not share the same data storage strategy, consequently, a lot of data movement occurs during runtime in order to link the two libraries. Therefore, multiple solutions can be taken to tackle this problem. To name a few, the pyJac source code can be modified to have the same matrix layout as CVODE, or it can be developed a standalone implementation of a non-linear solver that fits both, CVODE and pyJac, APIs.
- This first version of the library uses CMake [56] as the build system. However, the implementation needs to be reviewed. It would be convenient to add more options in the configuration step, such as compiler flags, accept external dependency binaries and the option to build without compiling GPU sources.
- The current logging system supports standard output (console) and file logging functionality for the serial and OpenMP's integrators. A great improvement would

be to adapt the current interface to also support logging from the GPU.

- Only CSV concrete implementations of the Reader and Writer abstract class are currently available. In-memory concrete classes should be implemented to be able to use the current library as a plug-in for other libraries or applications.
- The results presented in the Chapter 6 come from a compilation with the open-source GNU C/C++ compilers (`gcc` and `g++`), except for the ".cu" sources that are compiled with the NVIDIA CUDA compiler, `nvcc`. It would be convenient for the library to support other C/C++ compilers, such as Intel, CLANG or NVIDIA's. Besides, it will drive to further performance analysis with discussions about what compiler performs better for this application.
- Only one ODE solver is included in the library. But the library architecture can easily incorporate other solvers, both with stiff or non-stiff solving capabilities.
- Researching about the possibilities of changing the combustion mechanism using the runtime configuration. Currently, a compilation step is needed to change the combustion mechanism for CVODE's integrators.

Bibliography

[1] Vamsikrishna Undavalli et al. "Recent advancements in sustainable aviation fuels". In: *Progress in Aerospace Sciences* 136 (2023), p. 100876. DOI: 10.1016/j.paerosci.2022.100876.

[2] International Civil Aviation Organization (ICAO). *Aviation and Climate Change*. Technical Report. ICAO, 2016.

[3] *Paris Agreement*. United Nations Framework Convention on Climate Change. 2015. URL: https://unfccc.int/process-and-meetings/the-paris-agreement.

[4] IATA. *Net-zero carbon emissions by 2050*. 2021. URL: https://www.iata.org/en/pressroom/pressroom-archive/2021-releases/2021-10-04-03/.

[5] NASA. *Space applications of hydrogen and fuel cells*. 2021. URL: https://www.nasa.gov/content/space-applications-of-hydrogen-and-fuel-cells.

[6] Ahmad Baroutaji et al. "Comprehensive investigation on hydrogen and fuel cell technology in the aviation and aerospace sectors". In: *Renewable and Sustainable Energy Reviews* 106 (2019), pp. 31–40. DOI: 10.1016/j.rser.2019.02.022.

[7] J. Hoelzen et al. "Hydrogen-powered aviation and its reliance on green hydrogen infrastructure - Review and research gaps". In: *International Journal of Hydrogen Energy* 47.5 (2022). Hydrogen Energy and Fuel Cells, pp. 3108–3130. DOI: 10.1016/j.ijhydene.2021.10.239.

[8] Center of Excellence in Combustion (CoEC). 2020. URL: https://coec-project.eu/.

[9] Irvin Glassman, Richard A Yetter, and Nick G Glumac. *Combustion*. Academic press, 2014.

[10] Wikipedia contributors. *Premixed flame*. 2022. URL: https://en.wikipedia.org/wiki/Premixed_flame.

[11] Roger Temam. *Navier-Stokes equations: theory and numerical analysis*. Vol. 343. American Mathematical Soc., 2001.

[12] Uriel Frisch. *Turbulence: the legacy of AN Kolmogorov*. Cambridge university press, 1995.

[13] Denis Veynante and Luc Vervisch. "Turbulent combustion modeling". In: *Progress in Energy and Combustion Science* 28.3 (2002), pp. 193–266. DOI: 10.1016/S0360-1285(01)00017-X.

[14] John David Anderson. *Hypersonic and high temperature gas dynamics*. Aiaa, 1989.

[15] Daniel Mira et al. "HPC-enabling technologies for high-fidelity combustion simulations". In: *Proceedings of the Combustion Institute* (2022). DOI: 10.1016/j.proci.2022.07.222.

[16] D Anderson John Jr. "Computational fluid dynamics: the basics with applications". In: *Science/Engineering/Math. McGraw-Hill Science* (1995).

[17] Gilbert Strang. "On the construction and comparison of difference schemes". In: *SIAM journal on numerical analysis* 5.3 (1968), pp. 506–517. DOI: 10.1137/0705041.

[18] Zhuyin Ren et al. "Dynamic adaptive chemistry with operator splitting schemes for reactive flow simulations". In: *Journal of Computational Physics* 263 (2014), pp. 19–36. DOI: 10.1016/j.jcp.2014.01.016.

[19] Sergio Blanes Zamora, Damián Ginestar Peiro, and María Dolores Roselló Ferragud. *Introducción a los métodos numéricos para ecuaciones diferenciales*. Editorial Universitat Politècnica de València, 2014.

[20] Richard L Burden, J Douglas Faires, and Annette M Burden. *Numerical analysis*. Cengage learning, 2015.

[21] Wilhelm Werner. "Polynomial interpolation: Lagrange versus Newton". In: *Mathematics of computation* 43.167 (1984), pp. 205–217.

[22] The MathWorks Inc. *MATLAB version: 9.14.0 (R2023a)*. Natick, Massachusetts, United States, 2023. URL: https://www.mathworks.com.

[23] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*. Vol. 14. Jan. 1996. DOI: 10.1007/978-3-662-09947-6.

[24] J.R. Dormand and P.J. Prince. "A family of embedded Runge-Kutta formulae". In: *Journal of Computational and Applied Mathematics* 6.1 (1980), pp. 19–26. DOI: 10.1016/0771-050X(80)90013-3.

[25] Lawrence F. Shampine and Mark W. Reichelt. "The MATLAB ODE Suite". In: *SIAM Journal on Scientific Computing* 18.1 (1997), pp. 1–22. DOI: 10.1137/S1064827594276424.

[26] Robert K Brayton and Charles C Conley. "Some results on the stability and instability of the backward differentiation methods with non-uniform time steps". In: *Topics in numerical analysis* (1972), pp. 13–33.

[27] R.K. Brayton, F.G. Gustavson, and G.D. Hachtel. "A new efficient algorithm for solving differential-algebraic systems using implicit backward differentiation formulas". In: *Proceedings of the IEEE* 60.1 (1972), pp. 98–108. DOI: 10.1109/PROC.1972.8562.

[28] Ernst Hairer, Syvert Norsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Vol. 8. Jan. 1993. DOI: 10.1007/978-3-540-78862-1.

[29] Hemanth Kolla and Jacqueline H. Chen. "Turbulent Combustion Simulations with High-Performance Computing". In: *Modeling and Simulation of Turbulent Combustion*. Ed. by Santanu De et al. Singapore: Springer Singapore, 2018, pp. 73–97. DOI: 10.1007/978-981-10-7410-3_3.

[30] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. "A complexity theory of efficient parallel algorithms". In: *Theoretical Computer Science* 71.1 (1990), pp. 95–132. ISSN: 0304-3975. DOI: 10.1016/0304-3975(90)90192-K.

[31] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. 1st. USA: CRC Press, Inc., 2010. DOI: 10.1201/EBK1439811924.

[32] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104.

[33] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[34] M.J. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.

[35] Daniel Kusswurm. "Advanced Vector Extensions (AVX)". In: *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*. Berkeley, CA: Apress, 2014, pp. 327–349. DOI: 10.1007/978-1-4842-0064-3_12.

[36] Steven Fortune and James Wyllie. "Parallelism in Random Access Machines". In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC '78. San Diego, California, USA: Association for Computing Machinery, 1978, pp. 114–118. DOI: 10.1145/800133.804339.

[37] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[38] James Reinders. *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[39] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[40] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2007.

[41] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C programming*. John Wiley & Sons, 2014.

[42] NVIDIA. *CUDA Toolkit*. URL: https://developer.nvidia.com/cuda-toolkit.

[43] Russ Miles and Kim Hamilton. *Learning UML 2.0: a pragmatic introduction to UML*. " O'Reilly Media, Inc.", 2006.

[44] David G. Goodwin et al. *Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes*. https://www.cantera.org. Version 2.6.0. 2022. DOI: 10.5281/zenodo.6387882.

[45] Alan C Hindmarsh et al. "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 363–396. DOI: 10.1145/1089014.1089020.

[46] David J Gardner et al. "Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* (2022). DOI: 10.1145/3539801.

[47] Kyle E. Niemeyer, Nicholas J. Curtis, and Chih-Jen Sung. "pyJac: Analytical Jacobian generator for chemical kinetics". In: *Computer Physics Communications* 215 (2017), pp. 188–203. DOI: 10.1016/j.cpc.2017.02.004.

[48] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. "Towards dense linear algebra for hybrid GPU accelerated manycore systems". In: *Parallel Computing* 36.5 (2010). Parallel Matrix Algorithms and Applications, pp. 232–240. DOI: 10.1016/j.parco.2009.12.005.

[49] Edward Anderson et al. *LAPACK users' guide*. SIAM, 1999.

[50] Chuck L Lawson et al. "Basic linear algebra subprograms for Fortran usage". In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[51] Azzam Haidar et al. "Framework for Batched and GPU-resident Factorization Algorithms to Block Householder Transformations". In: *ISC High Performance*. Springer. Frankfurt, Germany: Springer, July 2015.

[52] *GoogleTest User's Guide*. http://google.github.io/googletest/. Version 1.13.0. 2023.

[53] Grant Allen and Mike Owens. *The definitive guide to SQLite*. Apress, 2011.

[54] M Frenklach et al. *Gri-mech: An optimized detailed chemical reaction mechanism for methane combustion. topical report, september 1992-august 1995*. Tech. rep. SRI International, Menlo Park, CA (United States), 1995.

[55] John L Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[56] Ken Martin and Bill Hoffman. "An Open Source Approach to Developing Software in a Small Organization". In: *IEEE Software* 24.1 (2007), pp. 46–53. DOI: 10.1109/MS.2007.5.