Chapter 4

Parallel Computational Fluid Dynamics

4.1 Why Parallel Computational Fluid Dynamics ?

The computing power (in terms of flops, RAM memory and disk space) available on the average desktop computer has exploded in the past few years. This is usually attributed to two reasons [148]. First, because microprocessor companies inherited the successful elements of supercomputer designs and second and perhaps more important, because of the emergence of a personal computer and business market with increasing power demands. With such a large (and competitive) market available, a huge research effort has been invested into the development of inexpensive high performance processors for the home market. A typical PC has performance exceeding that of a supercomputer of a decade ago, and similar to a current workstation (so nowadays the classification of small computers into PCs and workstations is probably nonsense).

For many applications, performance is no longer the issue: we can just wait for a few years, watching sequential computers get faster enough for our application and forget about parallel computing. This is not the case for scientific computing:

- It is not clear that this increase in computational power can be sustained for many years in single processor systems, due to both technical and economic reasons.

  Form a technical point of view according to different predictions the current clock rates are close to the theoretical maximum. For instance, according to [149], the clock rates will continue growing up to 2GHz and then level off[1].

  From an economic point of view, the competition in the single processor market might decrease due to the extremely high cost of developing new processors. The number of vendors of high-end microprocessors has continuously decreased in the last years.

- In order to accurately simulate the behavior of many relevant physical phenomena, we need to deal with systems of a huge number of freedom degrees. As an example, steady-state CFD problems with 2×10^6 nodes are being routinely solved in industrial contexts, for engineering purposes.

The second reason will be illustrated with an example. One of the most relevant examples of such physical phenomena is the simulation of turbulent flows. As discussed in section 1.3, DNS approach needs computational resources growing with at least Re^4. To deal with them, parallel computers are needed.

If we do not consider execution time, any computational task that can be done by a parallel computer, no matter how big, can also be done by a sequential computer. But, if we want it to be

[1]However, it seems that a new 4.5 GHz processor has recently been announced.

completed in a given time, assuming that any computational technology has a limit in the number of operations per time unit, we need parallel computers to do so.

With the algorithms currently available, the current sequential computers are that far from the resources needed for the application of DNS to all the problems where it would be of scientific or technical interest, that (as a working hypothesis) we can consider that our ambition is unlimited and that we actually want to solve problems with an *infinite* number of discrete unknowns.

Quoting from [92], "The solution of the very large sets of coupled algebraic equations that characterize wide-band-width systems, can burden the most powerful computers. Indeed, such calculations are always likely to be resource limited, since the required processing increases disproportionately with the system bandwidth B_W[2], and there is virtually no limit to the range of B_W that can be usefully exploited in the representation of natural systems". So, we can consider that we have to solve linear systems of *infinite* equations in a *finite* time.

To do so, any conceivable sequential processor would be too slow. However, parallel computing systems with P processors have the capability to do an *arbitrarily large* number of operations per time step, just increasing P.

Numerical algorithms that can use this power are needed. Going on with our example, such computers and algorithms would ideally allow the solution of a linear system of N unknowns, representing (for instance) a discrete approximation to pressure correction equation, or the problem model 2.7.3, using P processors in a time T_P, with

$$T_P = K\frac{N}{P} \tag{4.1}$$

where K is a constant (small enough for practical uses). To achieve this ideally scalable behavior, both an ideal algorithm and an ideal parallel computer would be needed.

To solve a problem with a given N, if there was enough interest (and money to pay for it), a system with a P large enough would be used. The constant K would decrease with the increase of computing power of each of the P processors and the enhancement of the algorithms.

An economic argument would led us to the same conclusion. The cost of the exceptionally fast sequential computers has always been very high as they need special hardware that has a very reduced marked compared with the global hardware market. Parallel computers, on the other hand, can be built using a large number *of the shelf* processors and network components.

Thus, the developers of algorithms for scientific computing must be able to reuse the relatively low cost computer components, developed for business and domestic markets and not necessarily optimized for their purposes. Many of the examples presented in this work have been computed with the JFF cluster of PCs (appendix B).

These seem to be the current trends in high performance computing. The most powerful computer in the world as the writing of this paragraph[3] is the ASCI Red at Sandia National Labs (USA). It is a distributed-memory system manufactured by Intel with 9472 Pentium processors, capable of 10^9 floating point operations per second (flops). Updated information of this system can be found at http://www.sandia.gov/ASCI/Red/.

However, in spite of the promises of PCFD, it is not yet a so common technology. There are different reasons to explain that (in order of increasing importance):

- Parallel computers are a relatively new technology that is still expensive, scarce and difficult to use. For instance, until quite recently, there have not been well established, hardware-independent programming environments allowing the development of truly portable parallel codes.

- Code for parallel computers is much more complex to develop, test and optimize than code for sequential computers.

[2]The symbol Q is used in the original.

[3]An updated list of the top 500 supercomputer sites is maintained at http://www.top500.org/

- Algorithms designed for sequential computers usually do not run efficiently on parallel computers. The development of new, parallel algorithms, is in some cases a very difficult task, if possible at all.

4.2 Overview of parallel computing technology

In this section a brief analysis of parallel computing technology and its effects over the development of efficient parallel algorithms will be given.

4.2.1 Hardware

Hardware taxonomy and memory distribution

With the evolution of parallel computing technology, some of the architectures that were designed and manufactured in the early days have become extinted by the market forces. As aforementioned, these forces do not necessarily direct evolution towards more efficient architectures for scientific computing but perhaps to better general-purpose systems than can be used in different applications. However, if only for historical reasons, it is worth considering a general classification. The standard[4] classification scheme of parallel computers is the *Flynn taxonomy*. It uses the relationship between the *instruction stream* and the *data stream* to classify the four different possible architectures:

- **SISD:** Single Instruction stream operating on a Single Data stream. This is a standard (Von Neumann) sequential computer, such as a Pentium-based PC.
- **SIMD:** Single Instruction stream operating on Multiple Data stream. A set of processors execute the same instruction on different sets of data. A SIMD machine can be emulated by a MIMD machine while the opposite is not true, so this architecture is going old of fashion.
- **MIMD:** Multiple Instruction streams operating on Multiple data Streams. These are the most versatile and currently popular parallel computers. They are essentially a set of different processors that can run the same or different programs with the same or different data sets. A sub-classification of MIMD computers will be given in next paragraphs.
- **MISD:** Multiple Instruction streams operating on a Single Data stream computers are included mainly for completeness as there are few, if any, commercial examples of this type.

As the practical totality of the parallel computers currently available for scientific applications are MIMD machines, we will concentrate in this type[5]. An important sub-classification of MIMD is performed according to their memory distribution:

- **Shared Memory** processors or *multiprocessors* share a global memory space. The key feature is the use of a single address space across the whole system, so that all the processors have the same view of memory. Any processor can directly access any address (for instance, any position of any array). Examples of shared memory machines include SGI Origin 2000 or Sun HPC servers. If the processors are not specialized in different tasks, such systems are referred to as *Symmetric Multi Processor* (SMP) systems.

  The kernel distributes dynamically the processes among the P processors, in order to balance the load. Both sequential and parallel programs can run together. From the point of view of the operating system, a parallel program is just a set of processes. Any process can dynamically ask for as much RAM as needed, up to the total available in the system.

[4]But considered *fairly rudimentary* by Tanenbaum [150].

[5]There is a debate about the classification of Shared-Memory Multiprocessors as MIMD or MISD, see [148], chapter 12. Here they are considered MIMD.

Access of memory becomes a sever bottleneck as the number of processors is increased. The cost of these systems increases more than linearly with the number of processors if the efficiency of the access to the memory has to be sustained. In practice, each processor of a shared memory system tends to be slower than the equivalent processor in a sequential system, even when running a sequential code.

- **Distributed Memory:** processors have their own private memory space. Access to other processors' data must be done through a network. If some of the P processors have more processes running than the average number of processes of the computer, there is no way to balance the load of the system without stopping some processes and restarting them again in other processors. There is an additional sub-classification of distributed memory systems:

  - Commercial distributed-memory parallel computers, or *multicomputers* like the IBM SP2 or the Cray T3E.
  - Clusters of sequential workstations with a dedicated network that are administered as a parallel computer [6]. The most important example of this class are the Beowulf (http://www.beowulf.org/) clusters of Linux-based PCs. In this line, the Computational Plant (Cplant) project at Sandia National Laboratories is developping a large-scale, massively parallel computing resource from a cluster of commodity computing and networking components [151].

    Note that a cluster of shared-memory systems is also possible and will probably be important in the future if small shared memory systems with 2-4 processors reach the domestic PC market and their cost is reduced. In order to exploit them efficiently, the groups of processes with more intercommunication are assigned to each shared-memory subsystem.

  For the same number of processors, the cost of distributed-memory systems is usually smaller than the cost of shared-memory systems. Its cost (except the network) grows linearly with the number of processors. However, they have disadvantages: a single process can only use the memory available in one processor; the communication performances are usually worse than in the shared memory systems (specially for the case of clusters of workstations); their administration is costly; they do not support easily the shared-memory programming model.

- **Shared Non-Uniform Memory Access** computers are a compromise between shared and distributed systems. In a NUMA (Non-Uniform Memory Access) system, the processors have direct access to all the memory located anywhere in the system. Processors can access directly to local memory but references to the memories on the other nodes must be sent across a network. Remote references take longer than local references. It seems that NUMA technology by SGI will soon become open-source so the concept would be implementable in Linux.

Networks

Hardware allowing the processors to communicate is a critical aspect of both shared and distributed memory parallel computers. From an abstract point of view, it does not make much difference whether we connect processors to memory or computers to one another. The role played by the network in a cluster of workstations is comparable to the role of the bus in a shared-memory computer.

Two different interconnection topologies will be shortly considered: The *bus architecture*, like in a network hub, is a broadcast interconnect that allows each component to watch each operation that occurs in the bus, providing just a single path to exchange data that has to be shared by all the processors. It is cheap but -like the bus of a shared memory system- does not scale with the number of processors. The *crossbar architecture*, like in a network switch, provides multiple independent paths to communicate the processors simultaneously. It is efficient but its cost grows (theoretically) more than linearly with the number of processors, so it can not be scaled to large systems.

[6] Both the availability of a dedicated network and administration of the system as a parallel computer are essential to achieve reasonable performances. The latter condition excludes all the iterative work with the workstations.

There are other technologies [150, 148] to support the interconnection of large systems, involving several steps in the path from source to destination.

High level programmers do not need to deal with the network topology. The network software and hardware of current parallel computers hide the low-level details, allowing the codes to directly talk to any processor in the computer. However, it is useful to consider two aspects of the networks that are relevant for the algorithm design:

- Data transfer rate. The standard network model involves two parameters: The *network latency* L (s), the time needed to initiate the connexion between two processors and the *network bandwidth* B ($\frac{bytes}{s}$), the rate at which data is exchanged after the connexion has been iniciated. So the time t_d to transfer b bytes of data is:

  $$t_d = L + \frac{b}{B} \tag{4.2}$$

  This is, it is clearly better to send one long message rather than a set of short messages, even if the total amount of data transferred is the same.

- Local and global communications. The different parallel programming models allow easy one-to-all, all-to-one and all-to-all communications. The lower-level software and hardware have to express these operations in terms of a number of one-to-one communications, or broadcasts, and each of them has a cost. A typical example is the dot product of two vectors $\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{N} u_i v_i$. If each of the P processors has a subset of the components of both vectors, a partial local sum can be done without communications but the final result involves necessarily global communications, that depending on P and L can be more expensive than the partial sums done in parallel.

  In the algorithms in sections 5 and 7, two types of communications are needed: Each-to-neighbours and all-to-all. In the first type, the term “neighbours” refers to the relative position of the processors in the virtual topology used to decompose the domain.

Loosely coupled and tightly coupled parallel computers

Let F be the sustained number of floating point operations that each of the processors of a parallel computer can do[7]. If the code only involves long messages, where we can neglect the latency, the ratio F/B, can be used to classify the parallel system[8]:

- *Loosely coupled* systems, such as clusters of workstations, have a high computing power compared to their communication performances (too high $\frac{F}{B}$).
- *tightly coupled* systems such as the Cray T3E, have good communication performance compared to their computing power (apropiated $\frac{F}{B}$).

The terms tightly and loosely coupled are also used in a more restrictive sense (for instance [69]) to refer to shared and distributed memory systems respectively.

Better parallel efficiencies can be obtained in tightly coupled systems. However, note that this is a relative concept that is not directly dependent on F or B but on their ratio, so the parallel computers that will be available in the medium term might have worse F/B ratios than the current systems. One of the recipes to obtain the best parallel efficiencies with your network is to choose the slowest processors available. Doing so, you will have a better (lower) $\frac{F}{B}$ ratio, and better speed-ups (but of course, this is cheating).

As an example, consider the aforementioned Sandia/Intel ASCI RED machine (the largest parallel computer as the writting of this paragraph). The design specification was 1 byte per peak (not

[7] F is usually only a fraction of the peak performance of the processor, which depends on the code being executed.

[8] The usual definition of loosely and tightly coupled systems (as for instance [150]) does not consider the ratio between the communications and computation capabilities but just the communications.

sustained) FLOPS rate of a single node. With the current technology this ratio can not be achieved using standard components.

Depending on the architecture of the parallel computer considered, the most scarce resource can be the network. In the design of CFD algorithms for loosely coupled systems, the effort has to concentrate in the reduction of comunications (the total data transferred *and* the number of the messages), as years ago it was typically concentrated in the reduction of the core memory and later in the reduction of the CPU time.

4.2.2 Software. Parallel Programming Models.

Parallel programming methodologies (or *models*) can be classified according to three criteria:

1. **Shared versus distributed memory**

   The *shared memory parallel programming models* are characterized by the existence of a global memory that can be directly read from and written to by every process involved in a computation. This memory holds the contents of certain variables (typically global variables) of the code while others are private to each processor. To transfer data between two processors, the sender just writes it in a global variable and the receiver reads it. It has two main advantages: *(i)* It is easier to use than the message passing programming model; *(ii)* The parallelization of a sequential code can be done step-by-step, starting with the CPU intensive areas (usually less than 20% of the code), while in the non-CPU intensive areas are executed only by one of the processors while the others wait for it.

   But it also has some inconvenients: *(i)* Until very recently, there has not been a standard protocol to guarantee portability [152]; *(ii)* Although it is in principle the easiest method from the programming point of view, it does not seem to encourage clean parallel designs (as allows a progressive migration from sequential to parallel codes and does not emphasize the need to minimize the data transference that certainly have a cost even in NUMA or shared memory systems). Additionally, subtle bugs can be easily generated.

   In a *distributed memory programming model* or *message passing programming model* each process is assumed to have only a local memory. No other processes can directly read from or write to it. To exchange data, processors must use messages. The codes tend to be of a lower-level than their equivalents in shared memory, but the programmers have the full control over the number and type of communications.

   The difference was originated according to the underlying computer architecture, but current technology allows other possibilities. The four possible combinations between hardware and programming models are summarized in Table 4.1.

	Shared memory parallel computers	**Distributed memory parallel computers**
Shared memory programming model	Natural	In principle possible, but difficult
Distributed memory programming model	Does not exploit all the hardware features but scales well	Natural

Table 4.1: Compatibility between parallel computing hardware and programming models.

2. **Explicit versus implicit programming languages.** The parallelism in the computation can be explicitly expressed by the programmer or it can be implicit and then automatically identified by the compiler. The last option is of course attractive but it is a challenge for compiler technology. Sometimes the programmer has to help the compiler to exploit the potential parallelism and this can be more difficult than to express it explicitly.

3. **Data-parallelism versus control-parallelism.** Parallelism can be expressed at the level of the operators (data-parallelism) of the language or at the level of the control structures (control-parallelism). As an example, in the former case, the language can provide an intrinsic function for computing the addition of two vectors in parallel. With control-parallelism, the programmer would specify (or the compiler would find by itself) that the iterations of a loop can be executed in any order and thus it can be done in parallel. Their borderline is diffuse: libraries providing functions that emulate data parallelism can be implemented to deal with specific data structures of a particular problem. This is discussed in section 4.6.

Only three of the eight possible families of programming models are in wide use. From higher level (easier to human beings) to lower level (easier to computers) they are:

1. Shared-memory implicit data-parallel models, like High Performance Fortran (HPF) or Vienna Fortran are attractive for their abstract representation of parallel computing. However, codes written using this model are very demanding on the hardware (i.e., tightly coupled parallel computers) and on the compiler. They are not freely available in all the platforms.

2. Shared-memory explicit models: Compiler directives and library functions that can be used to specify parallelism in Fortran and C programs. They are intended to combine the ease of use of the previous group with the functionality of the next group. However, they are restricted to shared-memory systems (and thus, they can not be universally ported) and until recently the lack of a standard has made the portability of this model very difficult, even among different shared-memory platforms.

3. Distributed-memory model with explicit parallelism or *message passing* programming model only allow the processes to read and write to their respective memories. They exchange data and syncronise by calling library functions, available for different languages (even Java versions will be soon available). This is the most efficient approach, even for shared-memory machines [153] but also the most involved. The difference between message-passing models and shared-memory models has been compared with the difference between assembler language and high-level languages. To implement algorithms in which it is essential to control the number and size of the data transferred by the processors, like the DDV or DDACM cycles, message passing is probably the best option.

   There are two main message passing libraries, PVM [154] and MPI [155]. MPI is generally accepted to be a better option, specially for homogeneous parallel computers, as it inherited all the experience acquired with PVM. Codes using MPI are totally portable to any conceivable platform including single-processor computers[9] .

4.3 Parallelism and scalability

An iterative equation solver will be used as a model to present different aspects related with parallel algorithms. Consider the solution of a linear equation system with N unknowns, such as the one presented in section 2.7.1:

$$Ax = b \tag{4.3}$$

using an iterative algorithm that, starting with an initial guess x^0, generates an iterative sequence of approximations to the solution x^k with $x^k = f(x^{k-1})$ and $x = A^{-1}b = \lim_{k\to\infty} x^k$. The iterations are stopped when the convergence criteria has been achieved,

$$||r^k|| = ||Ax^k - b|| < \epsilon \tag{4.4}$$

The number of iterations needed to achieve the criteria for a given ϵ, x^0, A and b is called n_ϵ.

[9]As an example, some of the parallel codes used in this work have been compiled and executed in parallel systems ranging from a three-processors Linux workstation cluster with a 10Mbit network (two of them 486 PCs with 8 Mbytes RAM) to a Cray T3E using 64 processors.

4.3.1 Dependencies

When in a computer program event A must occur before event B, we say that B is *dependent* on A. If data resulting from the first event should be used in the second, they are *data dependent.* In our model, as $x^k = f(x^{k-1})$ evaluation of x^k and x^{k+1} are data dependent. If different actions are to be selected in the second event depending on the results of the first event, they are *control dependent.* A control dependency example can also be found in our example: iterations go on or stop depending on $\|r^k\|$. Note also that if, as it is usual, the algorithm is parallelized by means of a decomposition of vector x, processors must communicate in order to evaluate $\|r^k\|$ and distribute this value to all of them.

Our goal in designing a *parallel algorithm* is to have as less dependencies as possible so that several different computations can be done at once by the different processors [10].

4.3.2 Iterative algorithms dependent on the number of processors

In general, parallel algorithms should be *independent* of the number of processors. This is, all the relevant intermediate results should not P. For the particular case of iterative algorithms, this means that the number of iterations needed to achieve the convergence criteria, n_ϵ, is the same for any number of processors (e.g., conjugate gradient without preconditioning). Although not totally essential, this is a desirable property, not only for computing efficiency but also because it helps debugging.

In certain algorithms, for the sake of computational efficiency, we may introduce changes so that intermediate results are a function of the number of processors, provided that the final result has to be the same. For instance, we can accept an algorithm in which, after iteration k, the intermediate result x^k is a function of P, the number of processors. This is the case of the DDACM algorithm discussed in section 7. We will refer to this parallel iterative algorithm as *dependent on the number of processors.* Under these conditions, if we stop the iterations when a convergence criteria is satisfied, the number of iterations needed, n_ϵ is a function of P. The same holds for the solution.

The second feature might seem alarming but it is essentially the same effect obtained when using a sequential method and different iterative algorithms: the solutions obtained are slighlty different, but as all them satisfy our convergence criteria, we shall consider that they are solutions of the problem. Of course $\|x_{P_1} - x_{P_2}\|$, where P_1 and P_2 are two different numbers of processor, can be made as small as wished just by decreasing ϵ (but this is not enough for chaotic flows, where the instantaneous maps would be a function of the number of processors).

4.3.3 Performance measures of a parallel iterative computation

Different measures of the efficiency are usually considered:

- The *numerical efficiency* of a parallel iterative algorithm dependent on the number of processors is defined as the quotient of the number of iterations done with one (n_1) and with P processors (n_P):

$$E^{num} = \frac{n_1}{n_P} \tag{4.5}$$

  The parallel iterative algorithms can be degradated with the number of processors, but not enhanced: $E^{num} \leq 1$. Otherwise, using one processor we could emulate the behavior of the P processors, call n_1' the new (smaller) number of iterations and redefine E^{num} accordingly.

[10] When describing parallel algorithms, it is usual to abuse the language, talking about *processors* when actually the term *processes* would probably be more correct. A process is an abstract concept that models the activation of a single program on a processor. Modern operating systems allow each processor to execute consecutively each of the members of a set of processes. Some of the processes of a parallel program may be executed alternatively by different processors. All them can even be executed by the same processor in a sequential computer, as it is usual when developing codes.

For non iterative algorithms (as for instance in section 6), can be analogously defined as the quotient of the number of operations done with one and with P processors. Note that E^{num} does not consider execution times but only numerical aspects. Thus, it is not influenced by computer hardware.

- The *parallel efficiency* is a measure of the performance of the parallel computation, not influenced by numerical efficiency aspects. Let t_P^i be the *wall time* needed by an iterative algorithm to perform i iterations with P processors and t_P the total time needed to do all the iterations needed to solve the problem.

  Wall time instead of CPU time is used in parallel computing. This is due to two reasons: First, each processor is assumed to be used only by one process (the opposite is nonsense in parallel computers), so both times should be roughly equal. Second, the time spent waiting for other processors to finish their jobs (due to load imbalance) and transferring data has to be accounted as well.

  The parallel efficiency is defined as:

$$E^{par} = \frac{t_1^1/P}{t_P^1} \tag{4.6}$$

  In general, $E^{par} < 1$ due to the time spent in the communications and to the load imbalance of the processors. It is independent of the number of iterations needed to finish the algorithm and of their unitary cost. Thus, it measures the hardware (network performance) and the algorithm (load balance), but not the overall efectivity of the algorithm.

- The *speed-up* (S) and the *efficiency* (E) are *global* measures of the performance of an algorithm running in a given parallel computer :

$$S = \frac{t_1}{t_P} = \frac{n_1 t_1^1}{n_P t_P^1} = E^{num} E^{par} P \tag{4.7}$$

$$E = \frac{t_1/P}{t_P} = E^{par} E^{num} \tag{4.8}$$

  Other authors, for instance [54, 156] express the total efficiency as the product of additional factors. These elaborated definitions are needed to work out *a priori* models to predict the behavior of the algorithms.

  Frequently, numerically inefficient sequential algorithms (i.e., with a long t_1 compared with other sequential algorithms) can be parallelized with high E^{par} and E^{num}. In these cases, high S and E values are obtained, but they do not actually reflect the behaviour of the algorithm. As an example, consider a Red-Black Gauss-Seidel smoother.

  Better measures for S and E could be obtained using t_{seq} instead of t_1, where t_{seq} is the wall time needed with *the best sequential algorithm available* ($t_{seq} \le t_1$). This is unpractical as a definition as there is generally no agreement about which is such algorithm. However, if there is a sequential algorithm known to perform better, but not parallel, it is our duty to mention it.

- Increasing the problem size. The previous measures allow us to quantify how good is our algorithm (and hardware) to reduce the time t to solve a problem with N unknowns to a precision ϵ. However, this is not usually our main goal when using parallel computers. Imagine that we have assumed that we have to wait for t seconds. What if our intention is to be able to increase N, without increasing t ?.

  Let $N^t_{max(P)}$ be the number of unknowns of the largest problem that can be solved with P processors in less than t seconds.

Using 1 processor, the maximum problem that we would be able to solve would have $N^t_{max(1)}$ unknowns. Thus, in the same time, the parallel computer allows us to solve a problem κ_P times bigger, where:

$$\kappa_P = \frac{N^t_{max(P)}}{N^t_{max(1)}} \tag{4.9}$$

Speedup and efficiency are the standard measures. However, in certain cases, the factor of increase in the problem size (here denoted κ_P) can be more useful.

4.3.4 Scalability

The term *scalability* is used to describe the capability of an algorithm to maintain values of E close to 1 when P is increased [11]. For non-trivial problems, scalability is an elusive property. This is due to different reasons:

- Amdahl's Law. Suppose that the total execution time of a program could be separated into a sequential and a parallel part with execution times t_{seq} and t_{par}. Then the execution time on P processors would be given by:

$$t_P = t_{seq} + \frac{t_{par}}{P} \tag{4.10}$$

  and the speedup by:

$$S = \frac{t_{seq} + t_{par}}{t_{seq} + \frac{t_{par}}{P}} \leq \frac{t_{seq} + t_{par}}{t_{seq}} \tag{4.11}$$

  So if, for instance there is only 10% of sequential tasks, the best speedup we can hope is 10 regardless of the number of processors available.

  This pessimistic result is known as *Amdahl's Law*. While it is not *wrong*, there is an element missing. When you double the size of the problem, the serial part increases usually less than the parallel part so by making a problem larger it becomes more parallel and larger speedups are possible. This is observed in practice with many problems (i.e., section 7).

- Load imbalance. Even for a perfectly parallel algorithm, the total work to be done can not be expected to be divisible by P. For a problem with n operations to be done in parallel, the difference between the number of operations in charge of the different processors increases with P. For instance, if $n = 100$ and $P = 3$, we have $100 = 33 \cdot 2 + 34$ and $34/33 \approx 1$, while for $P = 99$ we have $100 = 1 \cdot 98 + 2 \cdot 1$, so there is one processor that has double work than the others. The time to do the operations would be the same as if $P = 50$. Again, for problems of increasing size, the load balancing is easier.

- Communications overhead. Independently of the degree of parallelism, the processors have necessarily to exchange data. Depending on the number of operations to do between two messages and on the amount of data exchanged in each message, even a perfectly parallel algorithm can be inefficient on a loosely coupled system. Algorithms with a high number of messages need hardware with low latency and algorithms with long messages need hardware with high bandwidth. The first problem is specially critical.

A useful concept related to the tolerance of the algorithms to low network performances is the *grain* of the algorithm. *Coarse grain* algorithms have large portions of computations that can be done

[11] The term is also used to refer to the capability of a hardware architecture to be extended to a large number of processors.

locally, followed by communications, while *fine grain* algorithms only have small portions of computations between messages. Coarse grain algorithms can run efficiently even on relatively loosely coupled parallel computers, while fine grain algorithms need tightly coupled parallel computers. An example of a coarse grain algorithm can be found in section 4.7.

4.4 Control-volume based PCFD

4.4.1 Overview

Designing algorithms with a high parallel fraction *and* good load balance *and* exchanging a reduced number of short messages *and* doing as little operations as possible[12], is a challenge. Standard numerical analysis algorithms are not, in general, parallel. Usually, the better algorithms from the parallel point of view are inefficient from the numerical point of view. This is probably the main difficulty associated with scientific parallel computing.

From a parallel computing point of view, a possible classification of the numerical algorithms for the simulation of fluid flow phenomena is:

- Particle methods such as Lattice Boltzman or Direct Montecarlo Simulation Method, that use a limited number of "computational particles" - that can not be identified with real particles - to predict fluid behavior. An overview of different techniques, in the context of PCFD, is given in [69]. Particle evolution is concurrent and does not couple distant zones of the domain in each time step, so they seem to be better suited for parallel computers. Traditional applications included rarefied flows [157] and reactive flows. As an example of application to other flow types, the reader is referred for instance to [158, 159], where Direct Simulation Montecarlo Method is respectively used to solve Rayleigh-Benard convection and a driven cavity problem.
- Methods based on the numerical integration of the PDE governing equations [13]:
  - Totally explicit methods. The *domain decomposition* approach provides a natural way to parallelize explicit algorithms: the spatial domain to be solved is divided into a number of blocks or subdomains which can be assigned to different processors. The only data to be exchanged are the "inner boundary conditions" that each processor needs from its neighbours to proceed to next iteration.
  - Totally implicit or implicit/explicit methods, such as SIMPLEC or projection methods. It is more difficult to achieve an efficient parallel implementation of the implicit solution methods. This is due to the bottle neck caused by the solution of the linear equations. Unfortunately, as discussed in section 1.2, for incompressible flows, even for time-accurate solutions, there is at least one equation that has to be solved implicitly. There are different methods for the parallel solution of sets of PDEs using implicit techniques.
    - *Functional decomposition* is based on assigning different tasks to different processors. Here, the tasks are the different PDEs [160]. In the cases where additional unknowns have to be solved, like in reactive flows, this method is more attractive due to the higher number of scalar equations to be solved. However, as the different unknowns are coupled, after each time step all the fields have to be transferred to be used by all the processors. From the communications overhead point of view this is inefficient. Additionally, the continuity constraint (perhaps expressed in terms of a pressure correction) couples all the velocity components and has to be solved using another parallel approach.
    - *Domain decomposition in time* [161, 162] is based on the simultaneous solution of the discrete non-linear equations for different time steps in different processors. It also implies massive data transference after each iteration. An interesting possibility

[12]and doing something useful such as solve Navier-Stokes equations.

[13]Only local discretization techniques (section 2.3) have been considered here.

is to use it in combination of spatial domain decomposition in clusters of shared memory machines. Each shared memory system would use its processors for a domain decomposition in time solution of its spatial subdomains. In this way, the massive data transfers would only be done within each shared memory system.

∗ *Spatial domain decomposition* [163, 156] is perhaps the best strategy also for implicit solvers (or at least it is the standard approach), so it is considered with more detail in next section.

For the case of implicit CFD, the target of this work, in order to work efficiently on a parallel computer, the crucial point is the choice of the numerical method for the solution of the sparse linear systems. The parallelization of the other components of the method (i.e., coefficient evaluation), is straightforward. This is why, except section 4.7, the rest of the work is devoted to this problem.

4.4.2 Spatial domain decomposition for the solution of elliptic PDEs

Domain decomposition method is a generic name that is used to describe a variety of algorithms. The common aspect of all of them is that the spatial domain to be solved is divided into a number of blocks or *subdomains* which can be assigned to different processors. As the PDEs express local couplings, domain decomposition is perhaps the most natural strategy for this situation.

Different classifications of the domain decomposition method can be found in the literature. Subdomains can be overlapping or non-overlapping, parallel or rotated, etc. [164]. An important classification criteria is according to the stage where the domain decomposition is carried out:

- Approach A. Perform the decomposition before the discretization. Use overlapping subdomains and treat them as independent *continuous* problems with their own boundaries. Each of the subdomains uses as bo undary conditions information generated by the other subdomains where necessary. These internal boundary conditions are updated after each iteration. In this way, a sequence of solutions of the individual subdomains is constructed, that converges to the global solution of the problem. This approach is essentially a numerical version of the Schwartz Alternating Method (SAM). A history of the method can be found in [165].
- Approach B. Perform the decomposition after the discretization. In this approach, a single continuous domain is considered. Each processor is used to generate the discrete equations related with its part of the domain. Then, to solve the discrete equations, a parallel algorithm is used. Thus, the core of the PCFD problem becomes the efficient parallel solution of the linear equation systems. This is considered in section 4.5.

Numerical efficiency considerations apart, the first approach has two important advantages: *(i)* It is a coarse grain approach; as linear equations are not solved in parallel, it requires less communication between the processors, and only after each outer iteration and *(ii)* It allows to reuse almost all the sequential code without changes.

The first advantage is specially important if the code is to be used on loosely coupled systems. However, it has to be kept in mind that the PDEs to be solved in CFD are, in general, elliptic: local couplings are propagated to the entire domain. Quoting from Gropp and Keyes [166], "The domains of dependence of resolvents of elliptic operators, such as the spatial terms of the momentum and energy equations of (subsonic) fluid mechanics, are global, though there is a decay with the distance between source and field points. The global dependence implies that data must travel across the grid from each point to all others during the solution process (for the satisfaction of sensible accuracy requierements).". Thus, the SAM method does not scale well with the number of processors, unless the special circumstances of the flow help the convergence process.

Domain decomposition and multigrid. An important variant of the Schwartz method uses geometrical MG algorithms, doing a multi-level domain decomposition to enforce *global* convergence (as for instance in [167]). A clear discussion of the two possible methods to combine MG and domain decomposition is given in [168]. As these considerations are important for this work, an extract of it is reproduced here: In the first approach (method I in [168]), "...the domain decomposition acts as the outermost shell. For any block in the composite grid, the appropiate boundary conditions are obtained at the finest grid level and a multigrid cycle is then used to solve the governing equations within the block. This procedure is carried out for each block in the composite grid. In the second approach (method II), the multigrid component acts as the outermost shell. With this approach, for each level in the multigrid cycle, the appropiate composite grid is solved, with internal boundary conditions for each block being determined from neighbouring blocks within the same level. Therefore, within each multigrid cycle for the composite grid, internal boundary information is exchanged multiple times...".

The approach I is similar to the approach A previously discussed. In both cases, no technique is used to reinforce global convergence. Thus, due to the non-local nature of the equations solved (only the pressure correction equation for time-accurate flows or all the NS set for steady-state flows), approaches II and B are (in general) expected to be more efficient from a numerical point of view, but more difficult to parallelize efficiently on loosely coupled machines. The main aim of sections 5,6, and 7 is to find a way to do so.

As a counterexample, Schwartz Alternating Method (approach I or A) is known to behave well with parabolic flows: for each subdomain, as the guessed values at the downstream region have no effect over the domain, the information generated at the upstream region is quickly propagated from the first to the last subdomains [164]. This important feature of parabolic flows is exploited in the algorithm presented in section 4.7.

4.5 Parallel algorithms for the solution of linear equations

As it has been argumented in previous sections, the numerical simulation of complex fluid dynamics and heat transfer phenomena relies on the efficiency of parallel CFD codes. In turn, PCFD (or at least PCFD based on discretization of Navier-Stokes equations) relies on the efficient solution of the huge banded linear equations systems arising from the continuous governing equations.

Among the different families of algorithms used to do so, Block-Jacobi and Krylov subspace algorithms will be shortly considered. Schur Complement algorithm and Parallel multigrid will treated in detail in the next chapters.

4.5.1 Block-Jacobi algorithm

Overview

Block-Jacobi algorithms [104] are essentially the same idea as Schwartz iterative method [165, 169], but for the solution of the discrete equations rather than the original PDE.

The matrix A in $Ax = b$ is partitioned into submatrices as:

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1P} \\ \vdots & & \vdots \\ A_{P1} & \cdots & A_{PP} \end{bmatrix} \tag{4.12}$$

Each row of the block matrix is assigned to one of the P processors. Then, as it would be done with a scalar equation in a Jacobi algorithm, the unknowns outside the main diagonal are treated explicitly, using the values from the previous iteration:

$$A_{ii}x_i^{k+1} = -\sum_{i\neq j} A_{ij}x_j^k + b_i \tag{4.13}$$

Where k is the iteration. To evaluate the right hand side, processors must exchange data. Then, each processor solves its own subproblem, usually using a *local* iterative algorithm. Global iterations are stopped according to a convergence criteria.

When dealing with equations arising from structured meshes, the natural way to partition the matrix is to partition the mesh. In terms of the mesh, expression (4.13) means that each processor obtains its vector x^{k+1} as the solution of its subdomain, using the values provided by the neighbouring processors as if they where already correct.

Benchmark

Block-Jacobi algorithm is dependent on the number of processors. A numerical experiment has been done to illustrate its numerical efficiency. A steady-state, one-dimensional convection-diffusion equation, with different $Pe = \frac{\rho u L}{\Gamma}$ numbers has been solved with P processors, using a uniform mesh with N nodes in the domain $[0, 1]$, with boundary conditions $\phi(0) = 0$, $\phi(1) = 1$. The initial guess used has been $\phi = 0$. A uniform velocity field u, parallel to x axis is imposed. This problem is discussed in [57], section 5.2-3. Iterations are stopped when $||b - Ax|| < 10^{-6}$.

The first test has been performed using a direct solver (TDMA) for each of the subdomains. The number of block Jacobi iterations done for different numbers of processors is presented in Table 4.2.

	$N = 60$			$N = 600$		
P	$Pe = 0$	$Pe = 40$	$Pe = -40$	$Pe = 0$	$Pe = 40$	$Pe = -40$
1	1	1	1	1	1	1
2	429	9	68	4346	1	542
5	881	45	99	9045	205	789
10	1679	69	148	17279	335	1140
15	2471	93	203	25445	429	1523
30	4805	169	380	49507	739	2774

Table 4.2: Number of block Jacobi iterations using a direct solver for each subdomain.

The number of iterations increases dramatically with P, while the CPU time would only decrease linearly, as TDMA is a $O(N)$ algorithm. In a multidimensional domain, using MG instead of TDMA, the result would be similar, as MG is also roughly $O(N)$. Convection dominated problems have a better behavior but not still good enough to be of practical interest.

The difference between the number of iterations for positive and negative Pe numbers is due to the difference between the solution, and the initial guess. For positive Pe numbers, the left boundary condition $\phi = 0$ is transported through a large part of the domain. Thus, the solution is close to the initial map. The opposite situation occurs for negative Pe numbers, where the solution is close to 1 in a large portion of the domain.

Next test has been done using an iterative (Gauss-Seidel) algorithm as a local solver, with a fixed number of nodes $N = 60$. Different numbers of local iterations n_{loc} have been considered. E^{num} (here the ratio between the number of points relaxed with a single processor and with P processors) is presented in Table 4.3,

	$n_{loc} = 1$			$n_{loc} = 10$			$n_{loc} = 100$		
P	$Pe = 0$	$Pe = 40$	$Pe = -40$	$Pe = 0$	$Pe = 40$	$Pe = -40$	$Pe = 0$	$Pe = 40$	$Pe = -40$
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	0.966	1.0	0.980	0.608	0.941	0.500	0.108	0.222	0.058
5	0.921	0.924	0.934	0.410	0.326	0.360	0.054	0.044	0.040
10	0.854	0.846	0.872	0.265	0.231	0.259	0.027	0.028	0.027
15	0.797	0.788	0.822	0.192	0.175	0.197	0.019	0.021	0.019
30	0.665	0.661	0.700	0.097	0.094	0.105	0.009	0.011	0.010

Table 4.3: Numerical efficiency of block Jacobi iterations using Gauss-Seidel for each subdomain.

E^{num} is reasonably high only with a very low number of local iterations, where the algorithm behaves like a point Gauss-Seidel. This implies a fine-grain algorithm with a huge number of each-to-neighbours communications (as an example, 4805 with $N = 60$, $P = 30$ and $Pe = 0$). This is too costly for a loosely coupled computer, as will be seen in section 7.

Block-Jacobi algorithm can be enhanced with the use of *overlapping* domains. Each processor solves its subdomain plus a part of the adjacent subdomains. In the variant considered here, only the boundaries of the extended subdomains are transfered, so the communications cost is like in the non-overlapping version of the algorithm. The overcost is only due to the extra nodes to be relaxed. The previous experiment has been repeated with an overlapping area of 4 nodes. The results are presented in Table 4.4. As it can be seen, the results are only slightly better. An enhanced version of this technique is used as smoother in sections 5 and 7.

	$n_{loc} = 1$			$n_{loc} = 10$			$n_{loc} = 100$		
P	$Pe = 0$	$Pe = 40$	$Pe = -40$	$Pe = 0$	$Pe = 40$	$Pe = -40$	$Pe = 0$	$Pe = 40$	$Pe = -40$
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	0.977	0.882	0.929	0.849	0.882	0.860	0.499	0.588	0.352
5	0.870	0.815	0.831	0.626	0.579	0.606	0.255	0.186	0.186
10	0.841	0.826	0.837	0.444	0.382	0.432	0.092	0.075	0.069

Table 4.4: Number of block Jacobi iterations using an iterative solver for each subdomain with an overlapping area.

As a consequence of the low numerical efficiency of the plain Block-Jacobi (as well as Schwartz) iterations, they are not suitable, in general, as a parallel strategy for the numerical solution of Navier-Stokes equations. An important exception to this assert is exploited in section 4.7.

4.5.2 Krylov subspace algorithms

Krylov subspace algorithms [170] or *Non-stationary* [76] iterative algorithms, such as CG (Conjugate Gradient) or GMRES (Generalized Minimal Residual) are an important family of iterative algorithms for PCFD applications.

CG, restricted to symmetric positive definite matrices, is an illustrative example. A readable description of CG can be found in [104]. From a software engineering point of view, CG (like the other algorithms of the family) has the important advantage that it can be implemented without accessing to individual elements of the matrix A. Only matrix-vector product operation is needed. This means that a general CG algorithm can easily be implemented for different types of matrices. Additionally, their parallelization is straightforward. Each CG iteration only needs an each-to-neighbours communication (arising from a matrix-vector product) and an all-to-all communication (arising from a vector-vector product).

Their main inconvenient is the need of a preconditioner. The convergence properties of Krylov subspace algorithms strongly depends on spectral properties of matrix A. For realistic applications, *preconditioners* are always needed. There is little theoretical guidance for the selection of the best preconditioner for each application. However, the main problem is that the best preconditioners cancel both advantages: they need access to the matrices and they are not parallel.

Nevertheless, Krylov subspace algorithms are probably the most popular approach for parallel CFD [170, 163, 171, 172, 173, 174, 175].

4.6 Software engineering aspects of PCFD

As discussed in the introduction of this Section, one of the problems associated with parallel computing is the difficulty of developing and debugging the parallel codes. To reduce the programming effort, there are different options such as use shared-memory environments and parallelize only the most CPU intensive areas of the code (or use automatic compilers for the non-critical areas). These strategies are very useful for programmers facing the parallelization of scientific computing applications, possibly developed by other teams of people. But they may fail to provide the necessary portability among different platforms, and their performance in clusters of workstations may be limited.

If the applications to develop or port are restricted to a specific area, such as PCFD with structured meshes, another approach is to develop a layer of software, providing the necessary interface for higher level algorithms. Then, the programming and maintenance of the higher level applications can be done with a similar effort as in the case of the sequential codes. This has been the approach of this work, aimed to provide a first version of this lower level layer.

There are two possible working modes:

- **Subdomains mode** (as in section 4.7). There are a number of subdomains, that computationally are treated as separated entities, that exchange information through their boundary conditions, and cooperate to evaluate global information (such as the maximum residual). They are divided among the processors. Each processor can own one or several subdomains.

- **Single domain mode** (as in sections 5, 6 and 7). There is just one domain, shared by all the processors. Each processor can access legally to its part of the domain and to the adjacent areas (*halos*). Processors exchange local and global information as in the previous example. The main difference is that here, the processors can cooperate to solve a single equation system.

In future implementations, both modes could be combined: there would be a set of subdomains, each of them in charge of a set of processors.

The main principle followed is: **In the sections where each processor can work locally with its subpart of the problem, the parallel code should be as similar as possible to a sequential code**. This applies for instance to the evaluation of coefficient matrices (an important fraction of all the lines, and also the section where most of the maintenance effort is concentrated).

The calls to MPI functions and the points where the behavior of the code depends on the index of the processor or on the number of the processors have been concentrated in the lower layer of the software. In subdomains mode this is achieved quite naturally. The subdomains are distributed at the beginning of the execution according to the indications given by the user: automatic load balancing is not possible as the amount of computational work is not a function of the number of nodes. In single domain mode, global indices that sweep always all the fields independently of the number of processors have been used to access the scalar fields (velocity components, pressure, etc).

Each processor maps its area of the global field, plus the halos, to its memory according to a macro or an inline function, allocating only the memory space needed to do so. To simplify the code, processors adjacent to the external boundaries of the domain can access an "artificial" halo of values lying outside the domain. All the data to evaluate this mapping is stored using a data structure. High level functions are not allowed to access a field without using this protocol. A range-check control can be enabled during debugging.

The lower layer of software also provides other functionalities:

- A set of parallel I/O functions, that redirect standard output to a different file for each processor and implement a debbuging mode in which all the output is flushed immediately after each print.

- A module to catch exception signals and flushes the buffers before stopping the code.

- A function to do basic tests to verify that all the processors have the same CPU performance and verify that the network performance is as expected.

- Interfaces to implement the communication modes needed by the higher level functions.

- A module to measure execution (CPU and/or network) time of the different functions.

4.7 Implementation of a Schwartz Alternating Method to solve reactive flows

In this section, a first parallel implementation of the code DPC will be presented as an example of a large grain parallel algorithm that can run efficiently on both loosely coupled and tightly coupled parallel computers [169]. Currently, this code is routinely used to simulate flames. It is an algorithm designed for the solution of parabolic or quasi-parabolic flows that uses Schwartz Alternating Method. The main limitation of this approach is that its numerical efficiency is restricted to the simulation of determinate flow types, such as parabolic flows. Two examples are presented. In one of them, the flow is parabolic and the number of iterations does not increase with the number of subdomains, while in the other there are recirculations (elliptic flow areas) that cause the numerical efficiency to decrease.

4.7.1 Physical model

The aim of this model is to advance in the simulation of laminar flames using finite rate kinetics. Although industrial combustors work under turbulent conditions, studies focused on laminar flow conditions are a common issue in their design. Furthermore, a good understanding of laminar flames and their properties constitute a basic ingredient for the modeling of more complex flows [176]. The numerical integration of PDE systems describing combustion involves exceedingly long CPU times, especially if complex finite rate kinetics are used to describe the chemical processes. A typical example is the full mechanism proposed by Warnatz [177], with 35 species and 158 reactions. In addition of the momentum, continuity and energy equations, a transport convection-diffusion equation has to be solved for each of the species, as well as the kinetics. These detailed simulations are a key element to model combustion kinetics using less expensive models.

The governing equations for a reactive gas (continuity, momentum, energy, species and state equation) can be written as follows[14]:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{4.14}$$

$$\rho \frac{\partial \mathbf{v}}{\partial t} + (\rho \mathbf{v}.\nabla)\, \mathbf{v} = \nabla \cdot \tau - \nabla p + \rho g \tag{4.15}$$

$$\frac{d\,(\rho h)}{dt} = \nabla \cdot (k \nabla T) - \nabla \cdot \left(\rho \sum_{i=1}^{N} h_i Y_i \,(\mathbf{v}_i - \mathbf{v}) \right) \tag{4.16}$$

$$\frac{\partial\,(\rho Y_i)}{\partial t} + \nabla \cdot (\rho Y_i \mathbf{v}_i) = w_i \tag{4.17}$$

$$\rho = \frac{pM}{RT} \tag{4.18}$$

where t is time; ρ density; $\mathbf{v}$ average velocity of the mixture; τ stress tensor; p pressure; g gravity; N total number of chemical species; h specific enthalpy of the mixture; h_i specific enthalpy of specie i; T absolute temperature; k thermal conductivity of the mixture; M molecular weight of the mixture; R gas universal constant. The diffusion velocities are evaluated considering both mass diffusion and thermal diffusion effects:

$$v_i = v - D_{im} \nabla Y_i - \frac{D_{im}^{T}}{\rho} \nabla\,(\ln T) \tag{4.19}$$

[14]This model is not a part of this work, it has been done by other collegues of the Laboratory. Detailed information can be found in [178].

where D_{im} and D^T_{im} are respectively the mass diffusivity and the thermal diffusivity of the i specie into the mixture. The evaluation of the net rate of production of each species, due to the J reactions, is obtained by summing up the individual contribution of each reaction:

$$w_i = M_i \sum_{j=1}^{J} \left(\nu^{''}_{i,j} - \nu^{'}_{i,j}\right) \left[k_{f,j} \prod_{i=1}^{N} [m_i]^{\nu^{'}_{i,j}} - k_{b,j} \prod_{i=1}^{N} [m_i]^{\nu^{''}_{i,j}}\right] \tag{4.20}$$

Here, $[m_i]$ are the molar concentrations and M_i the molecular weight of the i specie, ν_{ij}, $\nu^{''}_{ij}$ the stoichiometric coefficients appearing as a reactant and as a product respectively for the i species in the reaction j, and $k_{f,j}$, $k_{b,j}$ the forward and backward rate constants. The transport and thermodynamic properties have been evaluated using CHEMKIN's database. More information of the model can be found in [178, 7].

4.7.2 Numerical aspects

In this implementation of the domain decomposition method, the meshes of the individual subdomains are overlapped and not necessarily coincident. The second feature allows more geometrical flexibility that, for instance is useful to refine the mesh in the sharp gradient areas at the edges of the flames. However, the information to be transferred between the subdomains has to be interpolated. This has to be done in a way that the interpolation scheme and the disposition of the subdomains adopted should not affect the result of the differential equations. For instance, methods that would be correct for one second order PDE [179] are not valid for the full Navier-Stokes set. If this condition is not satisfied, more outer iterations are needed and slightly wrong solutions can be obtained. Here, conservative interpolation schemes that preserve local fluxes of the physical quantities between the subdomains are used [180, 181, 182][15].

The governing equations are spatially discretized using the finite control volume method. An implicit scheme is used for time marching. A two-dimensional structured and staggered Cartesian or cylindrical (axial-symmetric) mesh can be used for each domain. High order SMART scheme [98] and central difference are used to approximate the convective and diffusive terms at the control volume faces. It is implemented in terms of a *deferred correction approach* [72], so the computational molecule for each point involves only five neighbours. Solution of the kinetics and the transport terms is segregated. Using this approach, kinetic terms are an ODE for each control volume, which is solved using a modified Newton's method with different techniques to improve the robustness [183]. To solve the continuity-momentum coupling, both segregated and coupled ACM can be used.

4.7.3 Illustrative results

Illustrative results obtained with the code are presented in Fig. 4.1. Two examples are presented: A flat flame (top) and a micro slit burner (bottom). The contour lines of the streamfunction[16] and the temperature (K) are at the left and the CO concentration at the right. In both cases, the laminar combustion of a stoichiometrical mixture of methane and air is studied. Their main characteristics are in Table 4.5. Symmetry boundary conditions are used and only one half of the domain is simulated. The dimensions and mesh size of the *computational domain* are indicated.

From the point of view of the fluid flow, their main difference is the presence of a recirculation area in the case of the micro slit burner.

[15]This critical aspect has not been considered in this work. This implementation in DPC is due to other collegues of the Laboratory.

[16]The iso-value lines of the streamfunction are tangent to the velocity field. Arbitrary iso-values have been plotted in order to show the main features of the flow.

	Flat flame	**Slit burner**
General description	Premixed CH4/Air (N2+O2) laminar flame	
Burner type	Flat flame on a multiple slit burner	Single confined flame on a micro-slit burner
Domain dimensions	0.075×0.4 cm	0.6×0.8 cm
Mass flow rate	2.6250×10^{-4} Kg/s	1.6564×10^{-3} Kg/s
Inlet temperature	$298K$	
Inlet composition	Stoichiometrical CH4/Air mixture	
Inlet velocity profile	Parabolic	
Outlet boundary conditions	$\frac{\partial(\rho v, Y_i, T)}{\partial n} = u = 0$	
Lateral boundary conditions	$\frac{\partial(\rho v, Y_i, T)}{\partial n} = u = 0$	Center: $\frac{\partial(\rho v, Y_i, T)}{\partial n} = u = 0$ Solid wall: $\mathbf{u} = 0$, $T = 298K$, $\frac{\partial Y_i}{\partial n} = 0$
Mesh size	60×80	
Reduced kinetics mechanism	Jones & Lindstedt: 4 reactions and 7 species [184]	

Table 4.5: Main characteristics of the illustrative examples of reactive flows.

4.7.4 Parallel performance

Before starting the parallel implementation, the sequential version of the code was used to evaluate its numerical efficiency for this problem, from one to ten subdomains (than can be assigned to different processors). For the case of the examples considered, the results are in Fig. 4.2. For the case of the flat flame, the number of iterations is almost independent of the number of subdomains ($E_{num} \approx 1$), while significative variations are found for the case of the slit burner. This is due to the presence of the recirculation area. In general, the number of iterations increases if the recirculation areas are divided into one or more subdomains.

Another important consideration is the cost per iteration of each of the subdomains. For the case of a flat flame[17], it was measured using a case with 10 subdomains, with the same number of control volumes per subdomain. A sequential computer was used. As it can be seen in Fig. 4.3, the cost is slightly different for each domain. This is due to the presence of solid areas in the subdomains at the bottom and to the different number of overlapping areas (two for the inner subdomains and one for the subdomains adjacent to the boundaries). Similar results would be obtained for the slit burner simulation.

Different parallel computers have been used to benchmark the code. For the case of a flat flame, where the number of iterations are almost independent of the number of subdomains, the results are presented in Fig. 4.4 *(i)* O2000: SGI Origin 2000, a shared memory system with R10000 processors; *(ii)* IBM SP2, a distributed memory system with thin 160 nodes; *(iii)* A Beowulf Cluster of Pentium II (266 MHz)[18].

For the benchmark, each subdomain has one processor (the code also allows each processor to solve a set of subdomains). The speed-ups obtained in the different systems (evaluated in relation to the respective times for one processor) are in Fig. 4.4. It is remarkable that, for the cases where the numerical efficiency is close to one, they are very similar in the different platforms.

This is because the algorithm requires little communication. For instance, for the most unfavorable situation (O2000 with 10 processors) only a single message of about 11.25 Kbytes is exchanged between neighbouring processors approximately every 0.15 seconds. So, the decrease on the efficiency is mainly due to the load imbalance and to the extra work done (overlapping areas and interpolations).

[17]The examples of this and the next paragraph are from [169], where the simulation of a flat flame, similar to the case described in Table 4.5 is used.

[18]The system at Daresbury Laboratory (UK) was used. All the information can be found at http://www.dl.ac.uk/TCSC/disco/Beowulf/config.html.

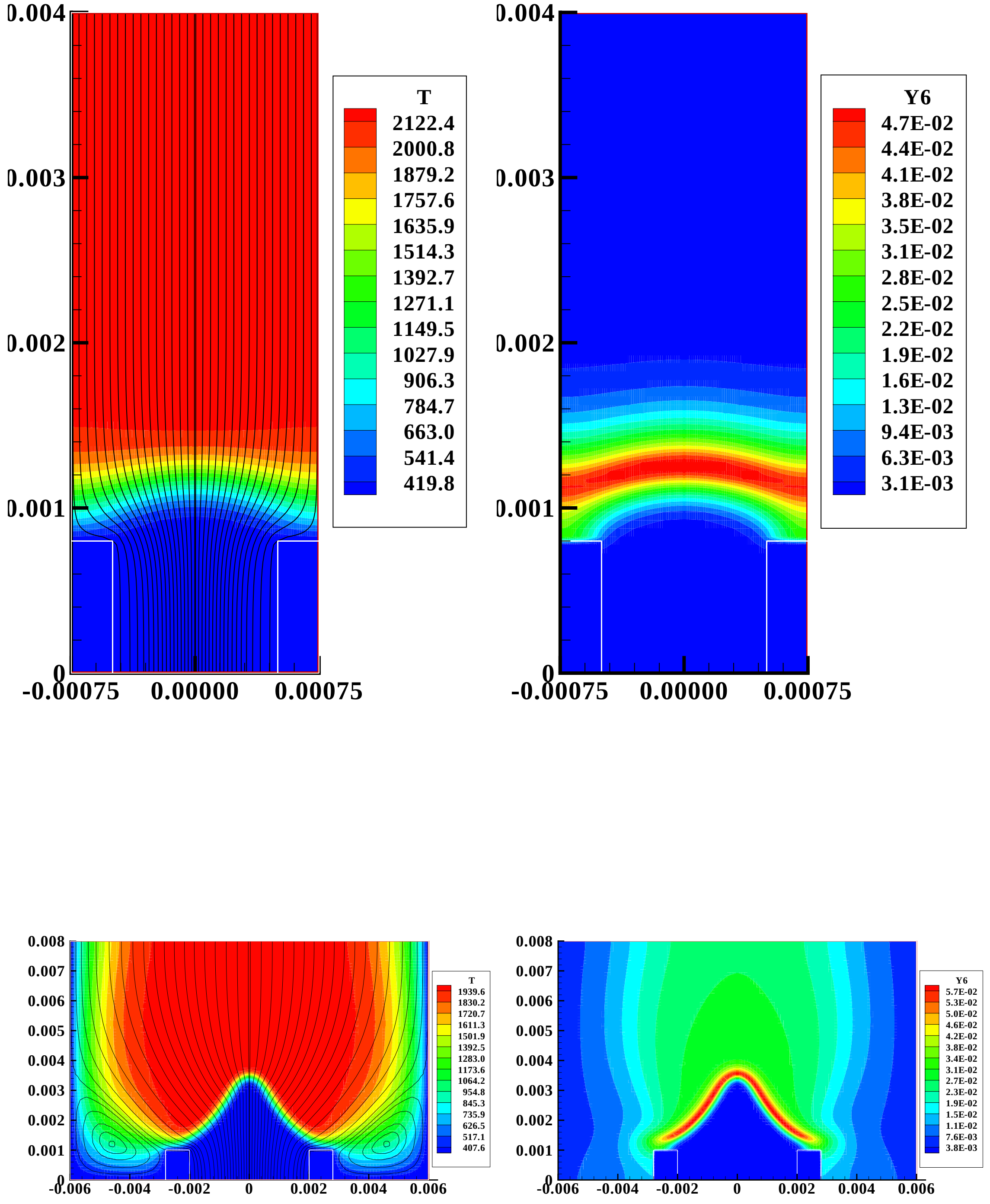

Figure 4.1: Illustrative results obtained with the parallel code for reactive flows. Top: flat flame. Bottom: Micro slit burner.

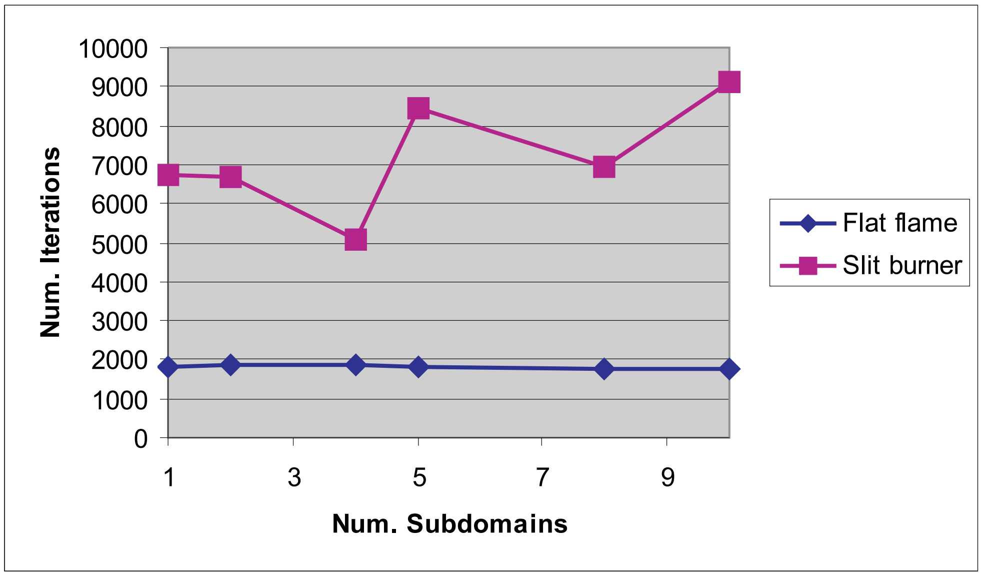

Figure 4.2: Iterations needed by the parallel code for reactive flows as a function of the number of subdomains.

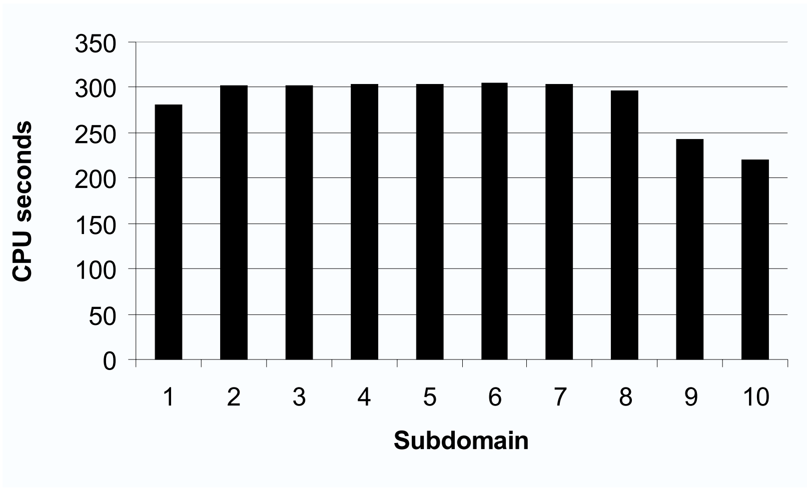

Figure 4.3: Time to solve each of the subdomains for a flat flame problem.

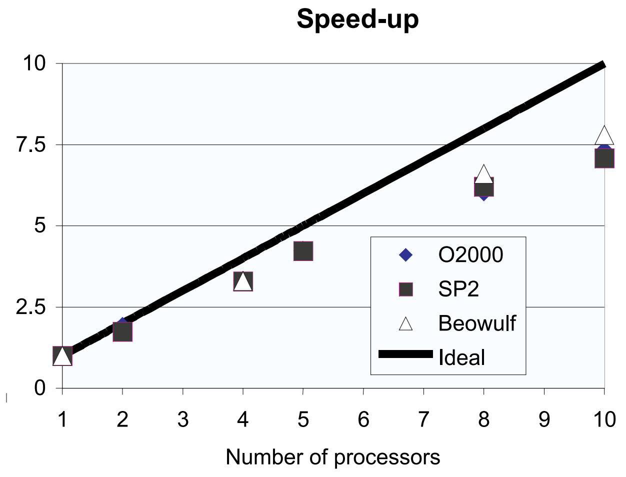

Figure 4.4: Speed-ups of the code for reactive flows (flat flame) in the different systems.

4.8 Nomenclature (4.1-4.6)

A	coefficient matrix
b	number of bytes
B	network bandwidth
E	efficiency
E^{num}	numerical efficiency
E^{par}	parallel efficiency
F	sustained number of floating point operations per second
K	constant
L	network latency
N	number of unknowns
$N^t_{max(P)}$	maximum number of unknowns that can be solved with P processors in a time t
n	number of iterations
n_ϵ	number of iterations to achieve the convergence criteria
Pe	Peclet number
Re	Reynolds number
B_W	bandwidth of an Eq. system
P	number of processors
r	residual vector
t_d	time to transfer data
S	speedup
t^i_p	wall time do to i iterations with p processors
u, v	generic vectors
x	unknown vector

Greek symbols

ϵ	precision
κ	increase of problem size

Subindices

1,2,3	Cartesian components
i	vector component
p	processor
par	parallel
P	obtained with P processors
seq	sequential

Superindices

k	iteration

4.9 Nomenclature (4.7)

t	time
ν	average velocity of the mixture
ρ	mass density
τ	stress tensor
h	specific enthalpy of the mixture
p	pressure
g	gravity
N	number of chemical species
T	temperature
k	thermal conductivity of the mixture
M	molecular weight of the mixture
R	gas universal constant
D_{im}	diffusivity
D^T_{im}	thermal diffusivity
J	number of reactions
$[m_i]$	molar concentrations
ν_{ij}	stoichiometric coefficients for specie i in reaction j (reactant)
$\nu^{''}_{ij}$	stoichiometric coefficients for specie i in reaction j (product)
$k_{f,j}$	forward rate constants
$k_{b,j}$	backward rate constants
$\mathbf{v}$	velocity vector

Subindices

i	referent to specie i