Chapter 5

Parallel estimation of distribution algorithms

'I do not fear computers. I fear the lack of them.'

Isaac Asimov

5.1 Introduction

The reduction in the execution time is a factor that becomes very important when using many applications nowadays. In spite of the enhancement on computer hardware, this problem is always happening, as the increase in power is always related to the application of new methods that were also too time consuming in previous computer systems. Parallel programming techniques provide feasibility of solving new problems or longer size ones.

The computation time is sometimes forgotten or avoided in some research works, as the validity of new methods or algorithms among others is in many times the most important point to be considered in the first stages. However, if one wants to apply parallelism techniques in a real life application (or even to commercialize it) the fact of reducing the computation time of the program becomes an important aspect to take into account. When willing to make faster the execution of a program, specially for algorithms that are very computationally consuming, we have four main choices:

1. We can optimize the code so that it does not repeat tasks or that minimizes the time of accessing slow functions such as disk access.

2. We can improve the hardware on which the algorithm is executing. This is done sometimes by recording the whole program in a ROM-type chip, but also by buying a faster processor, adding more processors to the computer, or by increasing the size of RAM memory of the computer.

3. Compilers that will convert automatically a sequential source program into a parallel one have also been proposed, but nowadays they mostly work for easily parallelizable programs such as vector and matrix operations.

4. We can rewrite the code making use of parallelization techniques.

From all these solutions, the first one can improve considerably the computation time of a program, but its main drawbacks are that this reduction has a limit and that highly optimized code is normally very difficult to maintain. The second solution is always possible, but it requires an important economic cost depending on how much we want the hardware to be optimized. In addition, it is important to analyze which is the hardware resource that makes the program go slower, as many times users tend to invest money in adding more and faster processors when the fact is that most of the time the bottleneck is the lack of RAM memory[1]. The third solution implies the use of high performance compilers that are able to provide parallel versions of a sequential source code in order to allow execution in different processors at the same time. These compilers use techniques to detect loops within the code where matrix or vector operations are performed, and they apply special techniques for parallelization. Unfortunately, these compilers are not *intelligent* enough to parallelize every sequential program and to detect data communication and computation parts within them that could be parallelized, and as a result these will not be a solution in our case. Furthermore, these compilers are usually very hardware dependent and appear to be very expensive. The interested reader can find more information about this subject in [Polaris, 1994, Wolfe, 1996].

Taking all the aforementioned reasons into account, we will concentrate on the last solution, the application of parallelization techniques. These are very powerful and effective for most of the cases, and they are often very easy to be applied to existing sequential programs. In addition, the proliferation of dedicated parallel libraries makes the application of parallelization techniques to be quite easy for a beginner on this field. This chapter intends to explain the basics of the existent parallelism techniques, the state of the art of the parallelization field, as well as to give an introduction to the most important ones using as an example the EDA program itself.

5.2 Sequential programs and parallelization

5.2.1 The design of parallel programs

Different techniques and methodologies can be applied for creating parallel solutions. However, the best parallel solution that can be developed is usually quite different from the sequential one. It is of fundamental importance to choose the proper methodology for parallel design in order to obtain the best parallel approach, in which concurrency aspects are taken into account in depth before focusing on machine-dependent issues.

In [Foster, 1995] one of the many possible parallel design methodologies is introduced. This methodology contains four distinct stages that are performed one after another: partitioning, communication, agglomeration, and mapping. During the first and second stages the programmer focuses on concurrency and scalability, while in the third and fourth stages the attention is on locality and other performance-related issues. These four stages are illustrated in Figure 5.1, and are explained as follows:

Partitioning: the parts on the problem that can be parallelized are identified and selected. This task is performed independently of the hardware type available, and issues such as the number of processors available are not taken into account at this stage.

[1]The lack of enough RAM memory forces the computer to work into the hard disk. The disk access time is measured on the order of milliseconds, while RAM memory is in the order of nanoseconds. Therefore this factor implies a considerable delay on the overall execution time of the whole program and the operating system.

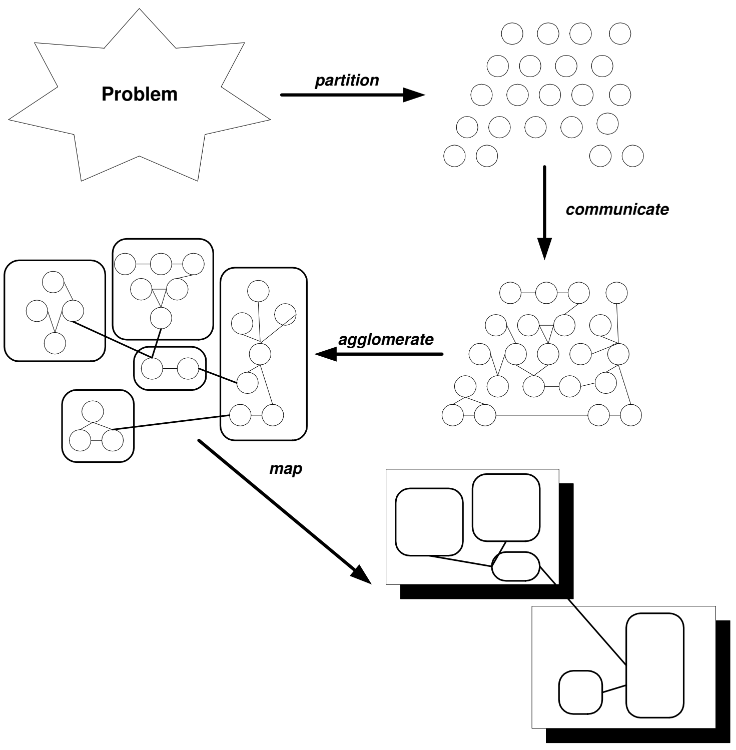

Figure 5.1: Parallel design methodology when parallelizing programs, as described in [Foster, 1995]. The four stages are illustrated: starting from a problem specification, (1) the problem is partitioned in smaller tasks that are divided in processes, (2) communication requirements between the different processes are determined, (3) processes are agglomerated, and finally (4) processes are mapped to processors.

Communication: the communication that is required to coordinate all the processes is identified and analyzed. Communication data structures and proper communication protocols are defined.

Agglomeration: this stage focuses on performance requirements and implementation costs. Some of the processes are combined into larger groups in order to improve performance and implementation costs.

Mapping: processes are organized regarding the number of processors available. The main objectives are to balance the work load of each process and to minimize communication costs.

Usually, parallel programs will create and destroy dynamically processes in order to obtain a balance in work load on amount of processes between processors. The design of parallel algorithms is presented here as sequential work, but often considerations at some stages require reconsidering previously designed aspects. Next, these four steps will be discussed in more detail.

Partitioning the problem

The main objective of the partition step is to divide the problem in the smallest possible tasks, in order to obtain what is called a *fine-grained* decomposition of the problem[2].

Partition is applied to both the whole computation job and the data associated to the processes. Usually, programmers focus first on the data partition (i.e. the way of partitioning the data is determined) and finally processes are associated to the partitioned data. This partitioning technique is known as *domain decomposition.*

Another approach is to focus first in partitioning the global job in processes and to work out afterwards which is the best way to partition the data. This is called *functional decomposition.*

These two techniques are complementary, and they can be applied to different parts of a single problem or even both can be applied at the same time to the whole problem in order to obtain different parallel algorithms to solve a single problem.

Process communication schemes in parallel programs

After partitioning the problem in sub-tasks assigned to processes, the programmer has to determine the way in which all these working processes will be coordinated and organized. As processes will require to receive information associated to another processes, the information flow between all the processes has to be analyzed with care. This task is performed in the communication phase of the parallel program design. For this, it is essential to take into account all the synchronization and communication aspects between all the processes.

A general idea to understand how to coordinate the different processes collaborating for a global job is the use of a producer-consumer scheme, which is a general approach of how two types of processes can be organized. The general case is the one in which there are a set of processes playing the role of the producers, where they produce elements that are stored in a buffer or an intermediate work pool. The rest of the processes will play the role of the consumer, which will read elements from the buffer or work pool and will use them for some determined task. Figure 5.2 illustrates this approach.

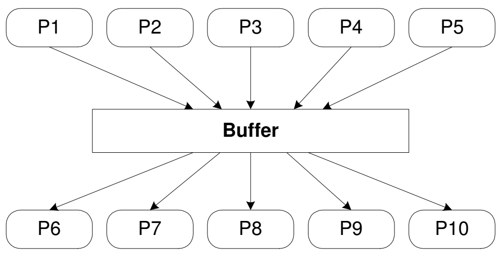

Figure 5.2: The producer-consumer approach.

However, there are also many other ways to organize the communication and synchronization between all the processes. In any case, it is always important to determine the nature of the communication channel to select the best way of communicating processes. In [Foster, 1995], these aspects are focused on the communication, of which different types are classified taking into account four orthogonal axes:

[2]A fine-grained decomposition represents the greatest flexibility possible to parallelize programs, as it divides the whole problem in the maximum number of tasks.

Local/Global: in local communication each process communicates with a small set of processes. In global communication it communicates with the rest of them.

Structured/Unstructured: in structured communication, all the processes form a regular structure when communicating. Regular structures can be a tree, a pipeline, or a grid for instance. On the other hand, in unstructured communication arbitrary graphs are formed.

Static/Dynamic: the identity of the communicating partner processes is always the same in static communication. When it is dynamic, the identity of the communicating partners can be different each time.

Synchronous/Asynchronous: in synchronous communication, there are producers and consumers of messages that execute in a coordinated way, with producer/consumer pairs cooperating in data transfer operation. In asynchronous communication, a consumer can obtain its data without the collaboration of the producer.

There are many working schemes that could be chosen for coordinating the execution of all the processes. None of them is better than the others for all the practical cases, and the programmer has to decide which is the one that best fits the organization of each parallel program.

Next, a short review of five typical and basic working schemes for organizing processes in parallel programming is given. Each of them is applied in those tasks in which their use has been shown to be the most effective. The programmer should be aware of all these possibilities and choose the one that he considers to be the optimum, as this choice as well as the information flow is of fundamental importance to obtain a satisfactory performance on parallel programs. A bad design could lead to a dramatic worsen on the performance of the whole program, resulting in some cases in even worse execution times than with a sequential program.

Generally speaking, the different basic working schemes are the following: phase parallel, divide and conquer, pipeline, master-slave, and work pool.

Phase parallel. A parallel program that follows this method will be divided in a series of two main steps. In the computation step, each of the processes will perform an independent computation in parallel. In the following step, all the processes will synchronize (either by using lock variables, semaphores or any other blocking communication method, see Section C.3 in Appendix C) and all the results will be gathered. Figure 5.3 shows this approach.

Divide and conquer. This algorithm for parallel programming is very similar to its sequential homologous as it can be appreciated in Figure 5.4. A parent process divides its computing weight in many smaller parts an it assigns them to a number of child processes. The children processes will proceed similarly, and then they will gather their children's results and send them back to their parent. This set of division on the job and gathering and sending of results is made recursively. The main drawback of this method is that a balance is required for an equitable division of tasks among processes. When using this scheme it is important to consider the dynamic nature of the problem, as well as to analyze the possibility of fully parallelizing it.

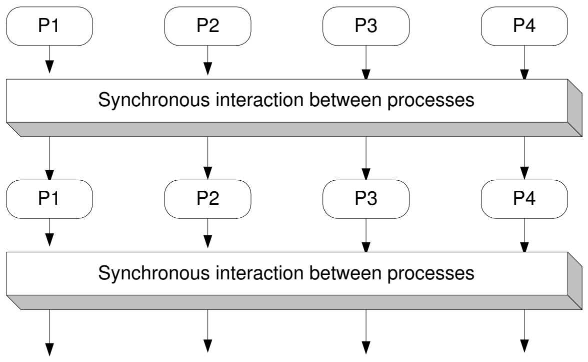
P1
P2
P3
P4
Synchronous interaction between processes
P1
P2
P3
P4
Synchronous interaction between processes

Figure 5.3: The phase parallel approach.

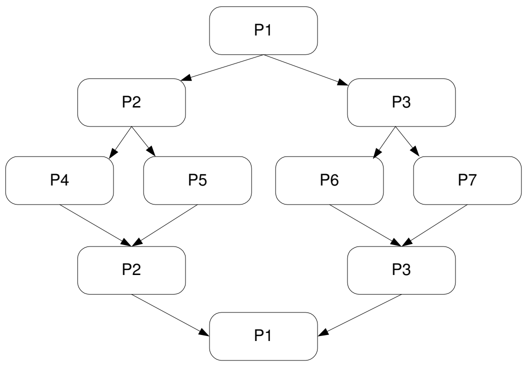

Figure 5.4: The divide and conquer approach.

Pipeline. In this approach, all the processes form a chain as shown in Figure 5.5. This chain is feed with a continuous flow of data, and processes execute the computations associated to each different step on the pipeline, one after another for each data-unit, but all work at the same time on different data-units. This behavior is similar to the execution of instructions in a segmented processor, in which it is important to take into account the dependence types of the input data.

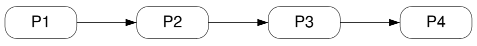

Figure 5.5: The pipeline approach.

Master-slave. As already explained, this working scheme –also known as *manager-worker* or *process farm*– is among the most applied. A process takes the role of the master or manager, executes parts of the global job that cannot be parallelized, and divides and sends to the rest of slave or worker processes the part of the global job that can be executed in parallel. This approach is illustrated in Figure 5.6. When a slave or worker process finishes its task, it sends back to the master the results obtained. Afterwards, the master is then sending more work to the slave. The main drawback is that the master process coordinates the whole information exchange, which in some cases results in a bottleneck.

In some particular problems, the master also performs part of the job instead of simply waiting for the rest of the workers, and therefore it also plays at the same time the role of a worker.

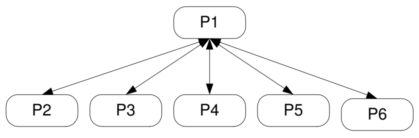

Figure 5.6: The master-slave approach.

Work pool. This method is often used with the model of shared variables. Figure 5.7 shows that this approach requires a global data-structure that is used as a work pool. Initially some basic amount of work has to be added to this pool. The main difference between this model and the previous master-slave one is that in this case all the processes are of the same type, as there is no master process on the scheme. All the processes that take part in the job can access it and produce: (1) no work, (2) a part of the work that will be placed on the pool, or (3) many parts of the work that will also be placed on the pool.

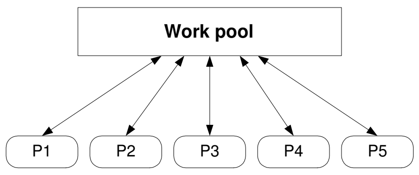

Figure 5.7: The work pool approach.

The parallel part of the program finishes when the work pool is empty. This method makes easier to balance the work load between all the processes. However, it is not easy to implement it when the message passing model is to be used (see Section C.3.5 in Appendix C) as this model does not allow an efficient access to the shared structure. The pool can be implemented as an unordered set, or an ordinary queue with or without priority.

Agglomeration

The agglomeration phase focuses on the performance of the system that has been designed so far: for instance, the fact of having many more processes than processors is highly inefficient in terms of execution time, as most of the time the system will not be able to keep executing all the processes at the same time. When agglomerating, we move from the theoretical design phase towards the realistic one, where the hardware resources available will be taken into account. The agglomeration is therefore the opposite step of the partitioning phase, in which the fine-grained solution is revised and the number of global tasks reduced. In addition to the reduction in the number of processes, in the agglomeration phase the replication of the data between different tasks will be determined to be worthwhile or not in terms of efficiency. In brief, the two objectives for the agglomeration phase is firstly to reduce the number of processes (by combining them and creating larger ones), and secondly to provide an appropriated number of them so that the processors can deal with, and to reduce communication costs.

Even if the number of tasks will be reduced in this step, it is usually the case that at the end more processes than processors will be scheduled. The reason for this is that processes will make use of other resources such as input/output devices: processes that are waiting for input/output devices will leave their processor free for a time, and in the meanwhile another process could make a profit and advance in its job.

Mapping

Finally, the mapping phase will distribute the processes among the available processors. The mapping phase does not arise on single-processor or on shared-memory computers that provide automatic task scheduling.

The goal of mapping algorithms is to minimize execution time. Two strategies are used for achieving this goal:

1. Processes that are able to execute independently are executed in different processors, in order to increase concurrency.

2. Processes that communicate frequently are assigned to the same processor, in order to increase locality.

Unfortunately, these two strategies sometimes conflict between them. In addition, the fact that usually there are more processes than processors is another factor that makes the mapping problem more complex.

There are several algorithms, such as load-balancing algorithms, that provide solutions to the mapping problem. This problem is known to be NP-hard, and discussion on this topic is out of the scope of this thesis. The interested reader is referred to [Foster, 1995] for more information on this topic.

5.2.2 Parallelizing an already existing sequential program

The algorithm to design parallel programs introduced in the previous section is suitable when an algorithm has to been redesigned from scratch. However, in many cases the programmer has already a sequential program to solve the problem that is executing too slowly just because of specific bottlenecks at different stages of the algorithm.

When a sequential version of the program is available, and we have access to its source code, it is not necessary to apply the four stages of the previous design model. Instead, a common practice is to identify the parts in the code that represent the most important bottlenecks in the program and to apply parallelization mechanisms to these parts.

Identifying bottlenecks in programs can be a difficult task if we do not know exactly how the program behaves. In addition, sometimes different input data could lead to very different CPU-time requirements of the different routines of the sequential algorithm. Hopefully, there are many tools to perform an analysis of the execution time rate of each of the routines on the program. An example of these tools is the *gprof*, which is a GNU tool that records all the required information for an exhaustive analysis. This tool is used together with the *gcc* ANSI C++ compiler, and an example of its application is shown later in Section 5.5.2. Such a tool helps the programmer to determine the routines that constitute the main bottlenecks in the program, which should be parallelized.

Once the routines to be parallelized have been identified, we could use one of the many communication and synchronization paradigms available. Many of these are described in

Section C.3.6 in Appendix C. For these, the task that the selected routines performs is divided in subtasks and, as well as in the previous section, communication and synchronization between them is determined and implemented.

An example of a way of coordinating the different processes collaborating for a global job is the use of a producer-consumer scheme, which was introduced in Section 5.2.1. Very often this approach is used for the case of a single producer and one or more consumers. It is important to mention that the client-server scheme is also a particular case of the producer-consumer: we could assume that the producers are clients that basically *produce* requests and store them in a structure such as a buffer, and the server will play the role of *consuming* requests by attending them.

Section 5.5 shows an example on how to apply these techniques for the case of having already a sequential program in which routines that represents bottlenecks are identified and parallelized.

5.3 Parallel architectures and systems

5.3.1 Parallel computer architectures

As the performance of any parallel program is heavily dependent on the hardware available, it is important to have a general understanding of the existing parallel machines in order to select the best one for our purpose. Obviously, one can always execute parallel programs on an ordinary PC or workstation with a single CPU and its particular memory, but the fact of having more than 2 or 4 processes or threads will lead to a competition between all of them for the use of the CPU rather than to the desired collaboration between them. The drastic reduction in computer prices in the last years has made possible for more users to acquire machines with more than a processor. Moreover, the existence of very fast local area networks allows single processor computers to communicate with each other with rates similar as communication within an internal bus of a machine.

We present in this section a classification of the different parallel systems. The classification is done based on the number of processors and the arrangement of CPUs and memory within the different parallel systems. One of the first classifications is the one based on Flynn's specification [Flynn, 1972]. An illustration of this classification is shown in Figure 5.8. Following this general classification, the main difference between computer systems is done regarding the number of processors and their type in the next way:

Single Instruction Single Data (SISD): This is an ordinary workstation system, where there is a single CPU and a single address map accessible by the CPU. This is not considered as a parallel system.

Single Instruction Multiple Data (SIMD): This type of systems can be considered as a first approach to parallel computing, and they are nowadays disappearing. In a typical SIMD machine we can have hundreds of CPUs, even thousands, all of them with a small private memory space. They all execute at the same time the same instruction over different data (at least, if the instruction makes it possible). These systems were thought for parallel computation of vector and matrix operations. When a CPU requires data stored in another's memory address map an explicit communication procedure has to be executed before. The main problems of these machines are their inflexibility as well as their high dependence on synchronization between all the CPUs of the system.

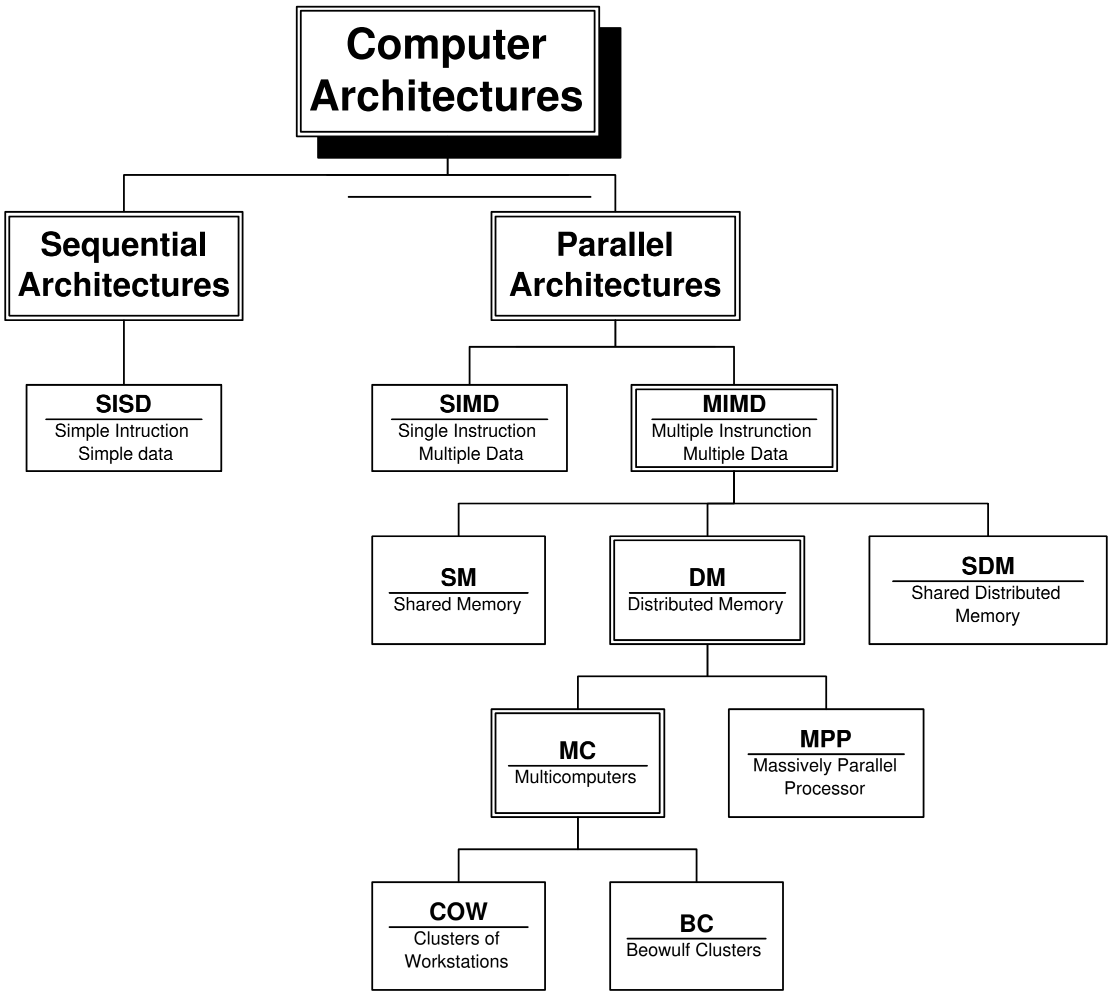

Figure 5.8: Illustration of all the different computer architectures. Here all the possible forms are shown, and many of them exist today. In some of these systems the number of processors is just one, but in other there can be thousands of them.

Multiple Instruction Multiple Data (MIMD): In this approach with many CPUs, all the processors have their particular memory space too, but the way of execution is very asynchronous and each processor can execute a different instruction, or even a different program. There is a complete independence in execution between all the CPUs. These systems are very convenient for nowadays's parallel systems. A diagram of the most used memory models is also presented in Figure 5.9. MIMD systems' performance shows a high dependence on the memory architecture, and depending on it we can classify these systems as follows:

Shared memory. The system has a single memory space, so that any computer can access any local or remote memory in the system, independently of the process that is the owner. Having a single memory space allows having shared memory for communication between all the executing workers or tasks, which makes it be an efficient communication mechanism. This is very convenient for parallel programs where the amount of data to process is very big, as the data requires no copying in order for all the CPUs to access it at the same time. In other

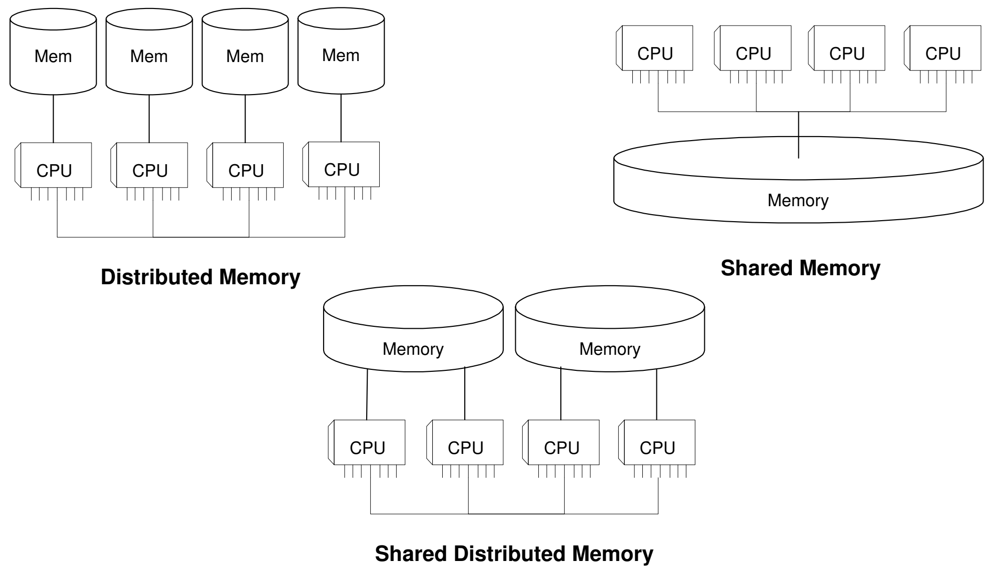

Figure 5.9: Illustration of the most used memory models for parallel computer architecture taxonomy.

cases, all this data should be exchanged between CPUs' address maps during the execution of the parallel program with the consequent lack on efficiency. An aspect to remember is that the disadvantage of using shared memory access is that shared variables lead to the existence of race conditions between parallel programs or routines, and therefore synchronization mechanisms have to be included in our parallel programs.

Distributed memory. In this case each processor has a particular address map that the rest cannot access at all. It is therefore essential that a process sends all the data to another if they have to collaborate to compute something. This communication operations are explicit (i.e. they have to be programmed). These systems have the advantage of being very easily scalable, but the lack of a shared address map adds a time penalty for each inter-process communication. An example of this type of computer systems are the so called Massively Parallel Processors (MPP).

Shared distributed memory. This hybrid model has the goal of having the advantages of both systems (i.e. the easy communication mechanism of shared memory and the scalability of distributed systems). These systems would contain their own memory address map, but the architecture is designed so that any node can access the memory of another with a slight time penalty. This type of architecture is a current trend on multiprocessor architectures.

In the recent years the development of two main fields in computing, such as the faster processors and the improvements of networks in communication speed, have also made available some new computer systems within the distributed memory MIMD model that could be regarded as *virtual parallel computers*. It is important to distinguish properly between three different possible machines that follow the MIMD model:

Network of workstations: in this model many very fast separated computers are connected through a fast Local Area Network (LAN). Each computer has its own console, can work as an independent machine, and occasionally they can all collaborate in a common task. This type of systems are known as NOW (Network Of Workstations).

Cluster: clusters are made of different machines, but there is a single console to control all the computers. However, in order to have full control of each of the nodes, it is necessary to do a connection through the console.

Multicomputers: in multicomputers all the different computers behave as a single one, and the user or programmer can control the whole execution of all the CPUs from a single node. No connection is necessary to a single node in order to control what is going on in the multicomputer.

This type of MIMD model machines are mainly built with clusters of workstations, where each workstation in the cluster acts as a separate computer. Processes on these systems cannot use shared memory, as the memory space of every workstation is physically separated from the others, and therefore message passing primitives are used for inter-process communication between processes executing in different workstations. In some special cases, these systems also are able to use virtual shared memory.

However, there are also special types of computer clusters composed of ordinary hardware architectures (i.e. ordinary PCs) with public domain software (i.e. Linux OS and dedicated libraries designed for fast message passing). A node plays the role of the server and controls the whole cluster, serving files to the rest of the nodes (i.e. the client-nodes). In addition, in some cases a node can also be a shared memory system (a multiprocessor). This particular type of clusters are known as Beowulf Clusters (BC) [Beowulf, 1994].

The main advantage of NOWs is their lower cost compared to other parallel machines, scalability, and code portability. The main drawback is the lack of available software specially designed for this type of systems in order to make the cluster behave as a single virtual machine. However, the existence of specialized public domain libraries such as Parallel Virtual Machine (PVM) and implementations of the Message Passing Interface (MPI) makes programmers easier to work with them (later in Section 5.3.3 these two libraries are analyzed in detail).

Finally, it is worth mentioning how typically all these parallel systems are combined with the different communication and synchronization methods. Typically, when shared memory is available the selected communication approach is to use shared buffers and variables that all execution units can access using as a result an explicit synchronization mechanism such as mutex semaphores. On the other hand, in distributed systems only message passing primitives are possible for communication between execution units, and synchronization is implicit on these primitives. Table 5.1 illustrates the way of combining communication paradigms (shared memory and message passing) with different architectures (multiprocessors and multicomputers). This table shows that the native communication models for multiprocessors and multicomputers are shared memory and message passing respectively, but that multiprocessors can also use message passing (which also can be efficient) and multicomputers can use shared memory mechanisms (although this latter solution in practice is not so efficient).

5.3.2 Comparison of parallel programming models

As explained before, parallel architectures as well as parallel software provide a way of dividing a task into smaller subtasks, each of them being executed on a different processor.

	Multiprocessors	Multicomputers
Shared memory	native	virtual shared memory library
Message passing	implemented over shared memory	native

Table 5.1: Table showing the combination of communication models and parallel architecture models. The native communication models for multiprocessors and multicomputers are shared memory and message passing respectively.

When programming this subdivision of tasks we can use several concurrent models, as described in Section 5.2.1, such as client-server or producer-consumer. Whichever the type of model chosen, two are the aspects that should be taken into account in order to compare two different parallel implementations of a computationally expensive sequential program:

Granularity: This is the relative size of each of the computation units (i.e. the amount of work for each of the workers) that execute in parallel. This concept is also known as coarseness, or fineness of task division.

Communication: This is relative to the way that execution units communicate to each other and how they synchronize their work.

It is important to take into account that the number of processes or workers is also an important factor to consider every time that the parallel program will be executed. This is very commonly a parameter of parallel programs. One might think that it is better to use as many processes as possible so that the workload of each is very low and each of them needs less time to complete its subtask, leading to a shorter time to complete the whole job. However, it is important to take into account that the number of CPUs is a limiting factor, as the maximum number of processes that can be running at the same time is equal to the number of CPUs (the rest will be in an idle state waiting for an executing process to pass to a blocked state and to take ownership of the freed CPU). In addition, it is also important to note that creating a new process is a procedure that also requires some additional time by the operating system, as well as the communication or synchronization procedures to coordinate them all. The latter is specially costly when processes are executing in different workstations connected through a network. That is why the number of processes has to be carefully chosen trying to find a balance between workload and cost for creating, communicating, and synchronizing these.

5.3.3 Communication in distributed systems: existing libraries

In the recent years some alternatives have been created in order to provide a programming interface when writing programs for multicomputers. Three of the most known ones are the following:

- Parallel Virtual Machine (PVM): PVM [Parallel Virtual Machine, 1989] is a software system that allows a heterogeneous network to be seen as a parallel computer. As a result, when a parallel program is to be executed, this is done over a virtual parallel machine –hence the name. PVM is maintained by the so called *Heterogeneous Network Computing research* project. This project is a collaboration of the Oak Ridge National Laboratory, the University of Tennessee and Emory University.

- OpenMP[3] application program interface: OpenMP [OpenMP, 1997] is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer. The design of OpenMP is based on threads that share a common memory space.
- The Message Passing Interface (MPI): MPI [Message Passing Interface, 1993] has been designed and standardized by the so called *MPI forum* which is formed by academic and industrial experts and is thought to be used in very different types of parallel computers. Nowadays there are many implementations of the standard available for many platforms. MPI is used now by the most important parallel computer vendors, as well as in universities and commercial companies.

PVM and OpenMP are standards where both the interface and implementation are defined (they are closed standards regarding these aspects), while in MPI only the interface is designed and many different implementations exist. These parallel programming alternatives were designed with the main objective of providing a solution to the problems of lack of standardization in parallel programming, that resulted in a lack of portability. Actually, different versions of these libraries exist for many different operating systems and architectures, which make them be portable enough for most of the existing computer systems.

5.4 Parallelism techniques in the literature applied to graph matching

Due to the complexity of many real graph matching problems, long execution times are required for dealing with all the search space as well as with all the data to evaluate each solution. As a result, parallelism techniques have been proposed in the literature to parallelize graph matching algorithms of very different types.

Many different parallelism techniques have been applied in the literature for faster computation of graph matching problems. Among them we have the use of linear combination and parallel graph matching techniques for 3D polyhedral objects representable by 2D line-drawings [Wang, 1999], and the use of a communication scheduling framework for communication algorithms [Bhat et al., 1999].

The subject of using parallel techniques to graph matching is also analyzed as a whole subject in some references. Examples of this are the description of the basic combinatorial, algebraic, and probabilistic techniques required for the development of fast parallel algorithms for graph matching problems and for closely related combinatorial problems [Karpinski and Rytter, 1998, Reif, 1993], and a study of different parallel algorithms as well as a proposal of a new one for attributed exact graph matching [Abdulkader, 1998].

On the other hand, there are also examples on applying graph matching techniques to improve parallelism in computer networks and operating systems. An example of this is tackling the problem of finding an optimal allocation of tasks onto processors of a distributed computing system [Tom and Murthy, 1999].

[3]The MP in OpenMP stands for *Multi Processing.*

All these references concentrate on the design of the parallel program, and in analyzing the complexity of the different graph matching approaches in order to obtain a simpler one. In addition, they are mainly based in shared memory approaches and threads, and practically no examples on combining fast networks and message passing standards for graph matching problems can be found.

5.5 Parallelization of sequential EDA programs

5.5.1 Computation needs in EDAs

Experiments applying EDAs to complex problems such as inexact graph matching –see Chapters 6 and 7– show that some of the EDA algorithms are very time consuming [Bengoetxea et al., 2000, 2001a,b,c,d, 2002a]. The execution time is directly proportional to the number of dependencies (i.e. maximum number of parents that each node can have) that the learning algorithm takes into account, as well as to the number of vertices of both the model and data graphs (i.e. the number of regions to recognize). This is specially evident in the case of EBNA for the discrete domain and EGNA for the continuous domain. Unfortunately, in real problems these graphs contain a lot of vertices due to imprecisions in image acquiring techniques, and therefore these two EDAs can even require many days to fulfill the stopping criterion.

Parallel programming techniques can be applied to improve these execution times by executing processes in parallel. As a first step before starting to modify the code, it is important to identify the steps on the algorithm that are suitable for parallelization, as some of its steps are inherently sequential and cannot be parallelized. An example of a step that one can think of to be easily parallelized is the simulation step. This step is used to create the R individuals that will form the next generation after the learning step in the probabilistic graphical model. These individuals can be created in parallel, as the simulation of each of these individuals has to be done without taking into account the generation of the previous ones.

Another important step that can be parallelized is the learning step in EDAs. EBNA and EGNA algorithms are among the most complex EDAs as they try to take into account all types of dependencies between the variables, and therefore the learning steps of these two are specially time consuming. Due to this reason, we will concentrate in parallelizing these two algorithms. However, before proceeding to any parallelization design, it is essential to know how the learning is performed on these. The learning procedures in EBNA and EGNA have been reviewed in Sections 4.3.4 and 4.4.4 respectively. The techniques described in Section 5.2.1 were applied for this purpose.

Whichever the procedure selected for parallelization, it is important firstly to analyze in the whole problem which of these steps is the one that consumes most of the CPU time of the overall execution time. This is an important factor since parallelizing a function that only supposes for instance a 5% in the global execution of the program will not have much influence in terms of reduction of execution time. The next section is a study of execution steps and procedures on EDAs.

5.5.2 Analysis of the execution times for the most important parts of the sequential EDA program

It is important to realize that the simple procedure of creating a new execution unit (either a process or a thread) requires some additional execution time, and therefore further study is required in order to be sure that the amount of work per process is big enough to justify the extra time-overhead of creating the process itself.

The different EDAs described so far for both the discrete and continuous domains have been executed and the time required for each of the internal steps and procedures has been measured. The tool used for this measurement is the widely known *gprof*, which is a GNU tool that records all the required information that we need. As already explained in Section 5.2.2, this tool is used together with the *gcc* ANSI C++ compiler. Table 5.3 shows the execution times in absolute values to obtain these results after execution in a two processor Ultra 80 Sun computer under Solaris version 7 with 1 Gb of RAM. It is important to note that these values do not correspond to ordinary execution times because the compilation options required for doing this analysis are different and debugging flags are active, thus making the execution much slower. After all, the results are given in statistical terms such as percentages of use of CPU, information that is enough for our purposes. That is also why execution in another machine such as a PC with Linux would return similar results in terms of execution time percentages.

Three experiments were carried out using the inexact graph matching problem with graphs generated randomly for Study 1 (the 10 & 30, 30 & 100 and 30 & 250 examples), and using the fitness function defined in Section 3.4.3. Some of the results obtained with these experiments are shown in Table 5.2.

In order to understand the results shown in Table 5.2, it is important to have a better understanding on the purpose of the procedures in the source code. There is a fitness function that is used to compute the value of each individual. This function is represented in the table as *Fitness Func.* and is the one defined in Equation 3.2. The fitness function is the same for both discrete and continuous EDA experiments, although in the latter algorithms there is an additional step in the continuous case as described in Section 3.3.2. These procedures and the way of carrying out the experiments are described in detail in Section 3.4.3 as well as in [Bengoetxea et al., 2001c,e, Mendiburu et al., 2002].

There is also a procedure that performs the learning for each of the EDAs, the learning of the probabilistic graphical model (Bayesian network or the Gaussian network in discrete and continuous domains respectively) expressed in the table as *Learning* for the discrete and continuous EDAs. This procedure constitutes the main difference between the different EDAs, and the relative execution time among the different EDAs of this procedure depends essentially on the complexity of the chosen algorithm. In the continuous domain, there are also two procedures called *Means and Covariances* and *Matrix operations* that are also part of the learning process of the algorithms (i.e. their times are included on the *Learning* part on continuous EDAs) that are on the table in order to show their relative significance in the execution time.

Finally, the last procedure to mention is *Simulation*, which finality is the generation of the R individuals of the next generation. This procedure is different in the discrete and continuous cases, but essentially they perform the same task.

The times given in Table 5.2 only reflect the time in percentages. This information shows clearly that in the EBNA$_{BIC}$ and EGNA$_{BIC}$ cases the fact of parallelizing the simulation step would not reduce drastically the overall execution time of the whole program, as in

EDA Algorithm	Procedure	Execution Time (%)	Procedure	Execution Time (%)
UMDA	Fitness Func.	9.6	Fitness Func.	81.9
	Simulation	7.4	Simulation	9.4
MIMIC	Learn Bayesian N.	7.7	Learn Bayesian N.	24.2
	Fitness Func.	12.89	Fitness Func.	63.7
	Simulation	6.4	Simulation	8.0
EBNA_{BIC}	Learning (BIC)	47.3	Learning (BIC)	85.7
	Fitness Func.	18.1	Fitness Func.	12.3
	Simulation	4.0	Simulation	1.4
UMDA_c	Means and Covariances	24.6	Means and Covariances	24.6
	Fitness Func.	14.3	Fitness Func.	34.7
	Simulation	2.1	Simulation	0.9
MIMIC_c	Learning	23.6	Learning	23.1
	Fitness Func.	13.7	Fitness Func.	34.1
	Means and Covariances	23.5	Means and Covariances	22.6
	Simulation	4.4	Simulation	3.5
EGNA_{BGe}	Learning	5.6	Learning	55.0
	Means and Covariances	21.4	Means and Covariances	7.2
	Fitness Func.	12.2	Fitness Func.	10.9
	Simulation	1.9	Simulation	0.3
EGNA_{BIC}	Learning (BIC)	53.1	Learning (BIC)	(*)
	Means and Covariances	3.5	Means and Covariances	(*)
	Fitness Func.	1.6	Fitness Func.	(*)
	Simulation	0.6	Simulation	(*)
EGNA_{ee}	Learning	34.2	Learning	(*)
	Means and Covariances	15.9	Means and Covariances	(*)
	Matrix operations	17.2	Matrix operations	(*)
	Fitness Func.	9.2	Fitness Func.	(*)
	Simulation	5.3	Simulation	(*)
EMNA_{global}	Learning	40.4	Learning	(*)
	Matrix operations	33.7	Matrix operations	(*)
	Means and Covariances	6.8	Means and Covariances	(*)
	Fitness Func.	5.4	Fitness Func.	(*)
	Simulation	10.3	Simulation	(*)

Table 5.2: Time to compute for two graph matching problems synthetically generated with sizes 10 & 30 (first column) and 50 & 250 (second column). All the figures are given in relative times, i.e. 100% = full execution time. The values with the symbol (*) would require more than a month of execution time to be properly computed.

the big case (last column) this step does not represent a significant execution time of the algorithm (only the 0.6% and 4% of the execution time). These two experiments show clearly that the most time consuming function is the *BIC* function, the one that is used to evaluate the different Bayesian and Gaussian networks in their respective versions.

It is also important to realize from the tables presented that the relative significance of the learning procedures in the execution time is more important when the size of the problem is also bigger. These data also show that the learning is not linear regarding the size of the problem, showing the already known NP-hard nature.

EDA Algorithm	10 & 30 example: execution time	50 & 250 example: execution time
UMDA	00:02:22	04:28:25
MIMIC	00:02:25	05:27:33
EBNA$_{BIC}$	00:04:44	37:01:45
UMDA$_c$	00:28:52	33:37:51
MIMIC$_c$	00:29:42	33:25:03
EGNA$_{BGe}$	00:34:14	146:43:30
EGNA$_{BIC}$	03:52:04	(*)
EGNA$_{ee}$	00:44:34	(*)
EMNA$_{global}$	01:46:34	(*)

Table 5.3: Time to compute the analysis in Table 5.2 for the 10 & 30 and 50 & 250 examples (hh:mm:ss). Again, the values with the symbol (*) required more than a month of execution time to be properly computed.

As discrete representations are mainly used in real graph matching problems we decided to parallelize the most CPU expensive discrete EDA that we defined: the EBNA$_{BIC}$. Therefore, this thesis concentrates mainly on parallelizing the BIC score, which implementation appears to be very important in the total execution time required by this EDA medium size problems.

5.5.3 Interesting properties of the BIC score for parallelization

Looking at the time required to execute algorithms such as EBNA in the discrete domain and EGNA in the continuous domain, it appears clear that parallel programming and concurrency techniques need to be applied in order to obtain shorter execution times. Some parallel algorithms have already been proposed in the literature for similar purposes [Freitas and Lavington, 1999, Sangüesa et al., 1998, Xiang and Chu, 1999], and also more concretely for the EBNA algorithm [Lozano et al., 2001].

In EBNA$_{BIC}$ and EGNA$_{BIC}$ the learning of the probabilistic graphical model is usually done by starting with an arc-less structure and by adding or removing step by step the arc that most increases the BIC score. This process is repeated until a stopping criterion is met, and the final result is a probabilistic graphical structure that reflects the interdependencies between the variables. As a result, both EBNA$_{BIC}$ and EGNA$_{BIC}$ are based on a score+search approach.

The BIC score is based on the penalized maximum likelihood. It can be written as:

$$BIC(S,D) = \sum_{i=1}^{n} BIC(i,S,D) \tag{5.1}$$

$$BIC(i,S,D) = \sum_{j=1}^{q_i}\sum_{k=1}^{r_i} N_{ijk}\log\frac{N_{ijk}}{N_{ij}} - \frac{1}{2}(r_i-1)q_i. \tag{5.2}$$

An important property of the BIC score is that it is *decomposable.* This means that the score can be calculated as the sum of the separate local BIC scores of each of the variables. Therefore, each variable X_i has associated a local BIC score –$BIC(i,S,D)$– as defined in Equation 5.2.

As a result, this allows us to compute the component that each variable adds on the global BIC score of the structure separately. It is important to remember that the arc adding or removing has to ensure that the structure will still be a DAG (Directed Acyclic Graph) in order to be accepted.

The structural learning algorithm has to find the best arc addition or removal in order to improve the BIC score. This task is accomplished by computing the corresponding BIC score for every arc modification. Therefore, as there are $n(n-1)$ possible arc modifications in a structure with n nodes, there are also $n(n-1)$ possible gains on the BIC scores to calculate each step arc modifications. The arc modification that maximizes the BIC score, whilst maintaining the DAG structure, is applied to S. Also, if the arc (j, i) is modified (i.e. added or removed), only the component $BIC(i, S, D)$ is affected. In the next step the rest of the $BIC(k, S, D) \;\; k \neq i$ do not change, and therefore only $n-2$ terms have to be computed.

All the computation of the different possible arc modifications can be computed separately. This task can be distributed to different processes that can be computing in parallel all the $BIC(i, S, D)$ in different processors.

5.5.4 Parallel techniques applied in this thesis

Regarding the different parallel architectures and systems shown in Section 5.3 where a parallel program can run and in order to offer the two possibilities of using shared memory or message passing, we have developed two different parallel versions of EBNA$_{BIC}$: one is based on using threads and shared memory, suitable for SM and SDM or multiprocessor machines in general, and was implemented using the *pthreads* library. The other is based on processes communicating using message passing, suitable for NOW, clusters, or multicomputers, implemented using the MPI interface. The reason for choosing these two parallelization standards is their proved performance, but also their portability and availability for different operating systems such as Windows, Solaris and Linux. Appendix D shows the main parts of the source code in EBNA, while an example of parallelizing the EBNA program in the way described is presented later in Section 6.4. Experimental results and conclusions are shown in the latter section for both using threads and MPI. These techniques are applied to the parallelization of the BIC procedure due to its computation cost in complex problems.