



UNIVERSITÀ DEGLI STUDI DI CAGLIARI
FACOLTÀ DI INGEGNERIA E ARCHITETTURA
Corso di Laurea in Ingegneria Elettrica ed Elettronica

Parallel computing: a comparative analysis using Matlab

Relatore:
Prof. Giuseppe Rodriguez

Candidato:
Edoardo Daniele Cannas

Anno Accademico 2015/2016

Contents

Introduction	ii
1 Parallel computing: a general overview	1
1.1 Definition and classification	1
1.2 Flynn's taxonomy	2
1.3 Parallel memory structures	4
1.4 Grain size and parallelism level in programs	6
1.5 Parallel programming models	8
1.5.1 Shared memory models	9
1.5.2 Distributed memory models	9
1.5.3 Hybrid and high level models	10
1.6 Modern computers: accustomed users of parallel computing	11
2 Parallel MATLAB	14
2.1 Parallel MATLAB	14
2.2 PCT and MDCS: MATLAB way to distributed computing and multiprocessing	16
2.3 Infrastructure: MathWorks Job Manager and MPI	17
2.4 Some parallel language constructs: pmode, SPMD and parfor	21
2.4.1 Parallel Command Window: pmode	21
2.4.2 SPMD: Single Program Multiple Data	21
2.4.3 Parallel For Loop	25
2.5 Serial or parallel? Implicit parallelism in MATLAB	27
3 Parallelizing the Jacobi method	29
3.1 Iterative methods of first order	29
3.2 Additive splitting: the Jacobi method	31
3.3 Developing the algorithm: jacobi_ser and jacobi_par	33
3.4 Numerical tests	35

<i>CONTENTS</i>	2
4 Computing the trace of $f(\mathbf{A})$	41
4.1 Global Lanczos decomposition	41
4.2 Gauss quadrature rules and Global Lanczos	44
4.3 Developing the algorithm: <code>par_trexpgauss</code>	45
4.4 Numerical tests	50
5 Conclusion	56
Bibliography	58

Acknowledgmentes and thanks

I would like to thank and acknowledge the following people.

***Professor Rodriguez**, for the help and opportunity to study such an interesting, yet manifold, topic.*

***Anna Concas**, whose support in the understanding of chapter's 4 matter was irreplaceable.*

All my family and friends, otherwise "they'll get upset" [21].

All the colleagues met during my university career, for sharing even a little time in such an amazing journey.

Everyone who is not a skilled english speaker, for not pointing out all the grammar mistakes inside this thesis.

Introduction

With the term *parallel computing*, we usually refer to the symoltaneous use of multiple compute resources to solve a computational problem. Generally, it involves several aspects of computer science, starting from the hardware design of the machine executing the processing, to the several levels where the actual parallelization of the software code takes place (bit-level, instruction-level, data or task level, thread level, etc...).

While this particular tecnique has been exploited for many years especially in the high-performance computing field, recently parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors, due to the physical constraints related to frequency ramping in the development of more powerful single-core processors [14]. Therefore, as parallel computers became cheaper and more accessible, their computing power has been used in a wide range of scientific fields, from graphical models and finite-state machine simulation, to dense and sparse linear algebra.

The aim of this thesis, is to analyze and show, when possible, the implications in the parallelization of linear algebra algorithms, in terms of performance increase and decrease, using the **Parallel Computing Toolbox** of **MATLAB**.

Our work focused mainly on two algorithms:

- the classical Jacobi iterative method for solving linear systems;
- the Global Lanczos decomposition method for computing the Estrada index of complex networks.

whose form has been modified to be used with MATLAB's parallel language constructs and methods. We compared our parallel version of these algorithms to their serial counterparts, testing their performance and highlighting the factors behind their behaviour, in chapter 3 and 4.

In chapter 1 we present a little overview about parallel computing and its related aspects, while in chapter 2 we focused our attention on the implementation details of MATLAB's Parallel Computing Toolbox.

Chapter 1

Parallel computing: a general overview

1.1 Definition and classification

Parallel computing is a particular type of computation where several calculations are carried out simultaneously. Generally, a computational problem is broken into discrete parts, which are solvable independently, and each part is further broken down to a series of instructions. The instructions from each part are executed on different processors at the same time, with an overall mechanism of control and coordination that oversees the whole process (for an example see figure 1.1).

Needless to say, the problem must be breakable into discrete pieces of work, and we have some requirements on the machine executing our parallel program too.

In fact, our computer should execute multiple program instructions at any moment in time: thus, in other words, we need a **parallel computer**.

Usually, a *parallel computer* might be a single computer with multiple *processors* or *cores*, or an arbitrary number of these multicore computers connected by a network. In the next sections we will try to provide a *general classification* of these machines, basing our work of scheduling on¹ :

- types of instruction and data streams;
- hardware structure of the machine;

We will try then to make a general overview of the most common *parallel programming models* actually in use, trying to illustrate how, indeed, our

¹Our classification is mainly based on [3]: for a comprehensive discussion, see [25], chapter 8.

computers exploit parallel computing in various ways and in several levels of abstraction. In fact, parallelism is not only related to our *hardware structure*, but can be implemented in our *software* too, starting from how instructions are given to and executed by the processor, to the possibility of generate multiple streams of instruction and thus, multiple programs to be performed.

We will try to give a general insight to each one of these aspects, considering that modern computers exploit parallelism in several manners that may include one or more of the means previously cited.

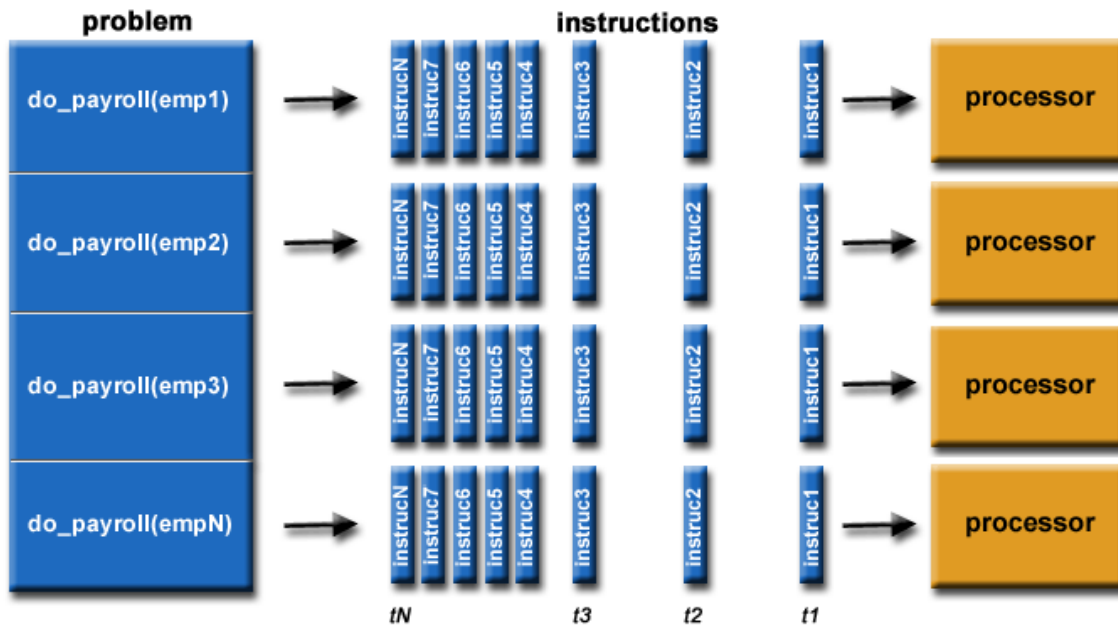


Figure 1.1: Example of a parallelization in a payroll problem for several employees of the same company

1.2 Flynn's taxonomy

Michael J. Flynn is a Stanford University computer science professor who developed in 1972 a method to classify computers based on the concept of *instruction* and *data* streams. An **instruction cycle** is a sequence of steps needed to perform a command in a program. It usually consists of an **opcode** and a **operand**, the last one specifying the data on which the operation has to be done. Thus, we refer to **instruction** and **data streams**, to the flow of instructions established from the main memory to the CPU, and to the

bi-directional flow of operand between processor and memory (see figure 1.2).

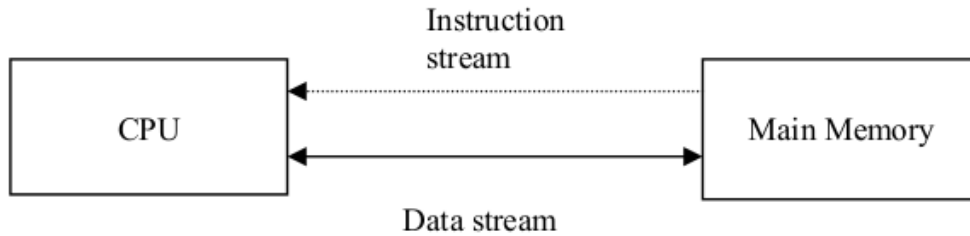


Figure 1.2: *Instruction and data streams*

Flynn's classification divides computers based on the way their CPU's instruction and data streams are organized during the execution of a program. Being I and D the minimum number of instruction and data flows at any time during the computation, computers can then be categorized as follows:

- **SISD (Single Instruction Single Data)**: sequential execution of instructions is performed by a single CPU containing a control unit and a processing element (i.e., an ALU). SISD machines are regular computers that process single streams of instruction and data at execution cycle, so $I=D=1$;
- **SIMD (Single Instruction Multiple Data)**: a single control unit coordinates the work of multiple processing elements, with just one instruction stream that is broadcasted to all the PE. Therefore, all the PE are said to be *lock stepped* together, in the sense that the execution of instructions is synchronous. The main memory may or may not be divided into modules to create multiple streams of data, one for each PE, so that they work independently from each other. Thus, having n PE, $I=1$ and $D=n$;
- **MISD (Multiple Instruction Single Data)**: n control units handle n separate instruction streams, everyone of them having a separate processing element to execute their own stream. However, each processing elements can compute a single data stream at a time, with all the PE interacting with a common shared memory together. Thus, $I=n$, $D=1$.
- **MIMD (Multiple Instruction Multiple Data)**: control units and processing elements are organized like MISD computers. There is only

one important difference, or rather that every couple of CU-PE operates on a personal data stream. Processors work on their own data with their own instructions, so different tasks carried out by different processors can start and finish at any time. Thus, the execution of instructions is asynchronous.

The MIMD category actually holds the formal definition of parallel computers: anyway, modern computers, depending on the instruction set of their processors, can act differently from program to program. In fact, programs which do the same operation repeatedly over a large data set, like a signal processing application, can make our computers act like SIMDs, while the regular use of a computer for Internet browsing while listening to music, can be regarded as a MIMD behaviour.

So, Flynn's taxonomy is rather large and general, but it fits very well for classifying both computers and programs for its understability and is a widely used scheme.

In the next sections we will try to provide a more detailed classification based on hardware implementations and programming paradigms.

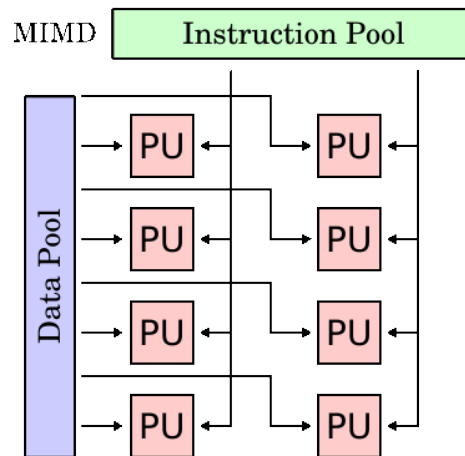


Figure 1.3: A MIMD architecture

1.3 Parallel memory structures

When talking about hardware structures for parallel computers, a very useful standard of classification is the way through which the memory of the

machine is organized and accessed by the processors. In fact, we can divide computers between two main common categories, the **shared memory** or **tightly coupled** systems, and the **distributed memory** or **loosely coupled** computers.

The first one holds all those systems whose processors communicate to each other through a shared global memory, which may have different modules, using several interconnection networks (see figure 1.4).

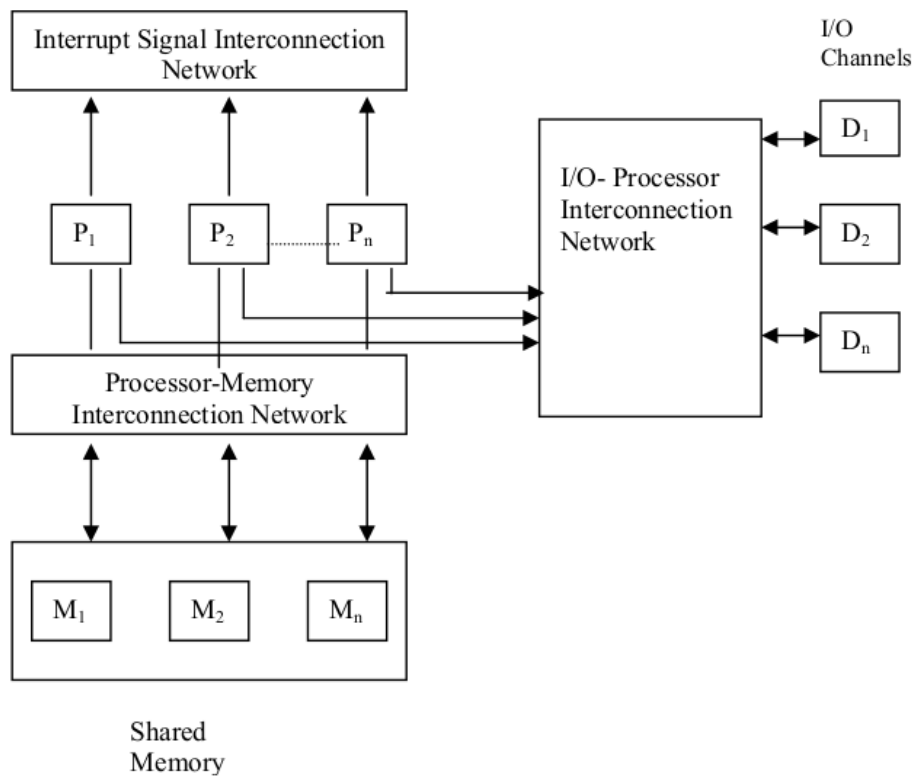


Figure 1.4: *Shared memory systems organization*

Depending on the way the processors access data in the memory, tightly coupled systems can be further divided in:

- **Uniform Model Access Systems (UMA)**: main memory is globally shared among all the processors, with every processors having equal access and access time to the memory;
- **Non Uniform Model Access Systems (NUMA)**: the global memory is distributed to all the processors as a collection of local memories

singularly attached to them. The access to a local memory is uniform for its corresponding processor, but access for data whose reference is to the local memory of other remote processors is no more uniform.

- **Cache Only Memory Access Systems (COMA)**: they are similar to the NUMA systems, being different just for having the global memory space formed as a collection of cache memories, instead of local memories, attached singularly to the processors. Thus, the access to the global memory is not uniform even for this kind of computers.

All these systems may present some mechanism of **cache coherence**, meaning that if one process update a location in the shared memory, all the other processors are aware of the update.

The need to avoid problems of memory conflict, related to the concept of cache coherence, which can slow down the execution of instructions by the computers, is the main motivation to the base of the birth of loosely coupled systems. In these computers, each processor owns a large memory and a set of I/O devices, so that each processor can be viewed as a real computer: for this reason, loosely coupled systems are also referred to multi-computer systems, or distributed multi-computer systems.

All the computers are connected together via message passing interconnection networks, through which all the single processes executed by the computers can communicate by passing messages to one another. Local memories are accessible only by the single attached processor only, no processor can access remote memory, therefore, these systems are sometimes called **no remote memory access systems (NORMA)**.

1.4 Grain size and parallelism level in programs

The term **grain size** indicates a measure of how much computation is involved in a process, defining *computation* as the number of instruction in a program segment.

Using this concept, we can categorize programs in three main sections:

- **fine grain programs**: programs which contain roughly 20 instructions or less;
- **medium grain programs**: programs which contain roughly 500 instructions or less;

- **coarse grain programs:** programs which contain roughly one thousand instructions or more;

The *granularity* of a program led to several implications in the possibility of parallelizing it. In fact, the finer the granularity of the program is, the higher degree of parallelism is achievable.

This not always means that parallelizing a program is always a good choice in terms of performance: parallelization involves demands of some sort of communication and scheduling protocols, which led to inevitable overhead and delays in the execution of the program itself.

Anyway, we can classify parallelism in a program at various levels, forming a hierarchy according to which the lower the level, the finer the granularity, and the higher the degree of parallelism obtainable is.

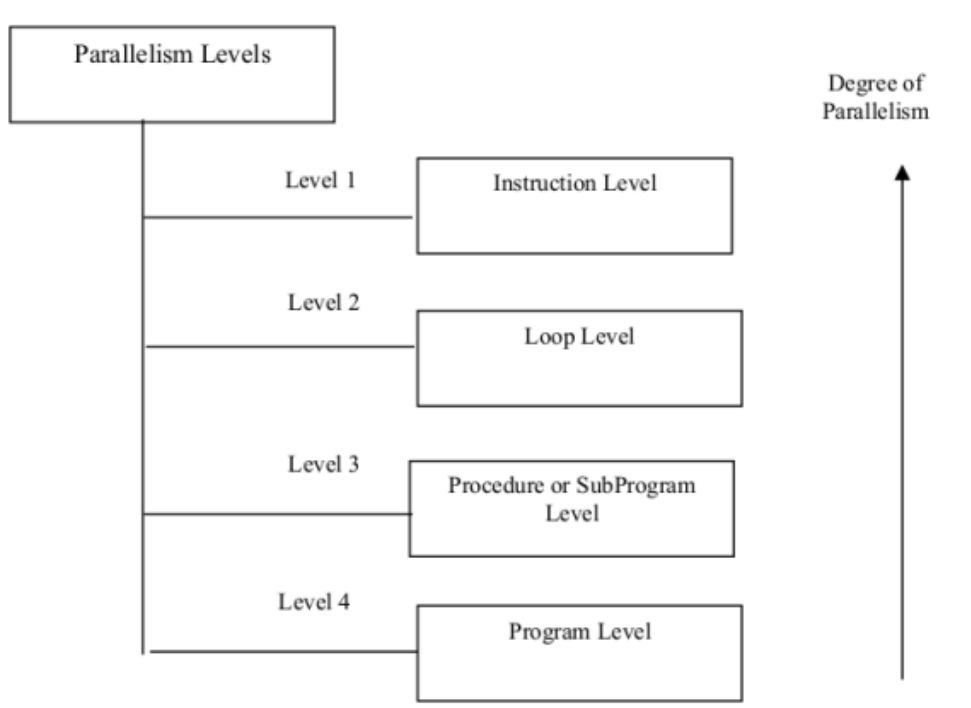


Figure 1.5: *Parallelism levels in programs*

Instruction level is the highest degree of parallelism achievable in a program, and takes place at the instruction or statement level. There are two approaches to instruction level parallelism [20]:

- **hardware approach:** the parallelism is obtained dynamically, meaning that the *processor* decides at runtime which instruction has to be executed in parallel;

- **software approach:** the parallelism is obtained statically, meaning that the *compiler* decides which instruction has to be executed in parallel.

Loop level is another high degree of parallelism level, implemented parallelizing iterative loop instructions (fine grain size programs). It is usually achieved through compilers, even though sometimes it requires some effort by the programmer to actually make the loop parallelizable, meaning that it should not have any **data dependency** to be performed in parallel.

Procedure or subprogram level consists of procedures, subroutines or subprograms of medium grain size which are usually parallelized by the programmer and not through the compiler. A typical example of this kind of parallelism is the *multiprogramming*. More details about the software paradigms for parallel programming will be given in the next section.

Finally, **program level** consists of independent program executed in parallel (multitasking). This kind of parallelism involves coarse grain programs with several thousand of instruction per each, and is usually exploited through the operative system.

1.5 Parallel programming models

A **programming model** is an abstraction of the underlying system architecture that allows the programmer to express algorithms and data structures. While programming languages and application programming interfaces (APIs) exist as an implementation that put in practice both algorithms and data structures, programming models exist independently from the choice of both the programming language and the API. They are like a bridge between the actual machine organization and the software that is going to be executed by it and, as an abstraction, they are not specific to any particular type of machine or memory architecture: in fact, any of these models can be (theoretically) be implemented on any underlying hardware [8] [4].

The need to manage explicitly the execution of thousands of processors and coordinate millions of interprocessors interactions, makes the concept of program model extremely useful in the field of parallel programming. Thus, it may not surprise that over the years parallel programming models spread out and became the main paradigm concerning parallel programming and computing.

There are several models commonly used nowadays, that can be divided in **shared memory** models and **distributed memory** models.

1.5.1 Shared memory models

The two main shared memory models are the ones using **threads** or not. The one which does not use threads, is probably the simplest parallel programming model existent. In this model, **tasks** (a logically discrete section of computational work, like a program or program-like set of instructions executed by a processor [4]), share a common address space, which to they read and write asynchronously, using mechanisms such as *semaphores* and *locks* to prevent deadlocks and race conditions.

The main advantage, which is also the biggest drawback of this model, is that all the tasks have equal access to the shared memory, without needing any kind of communication of data between them. On the other hand, it becomes difficult to understand and manage the **data locality**, or rather the preservation of data to the process that owns it from the access from other tasks, cache refreshes and bus traffic that are typical in this type of programming.

This concept may be difficult to manage for average users.

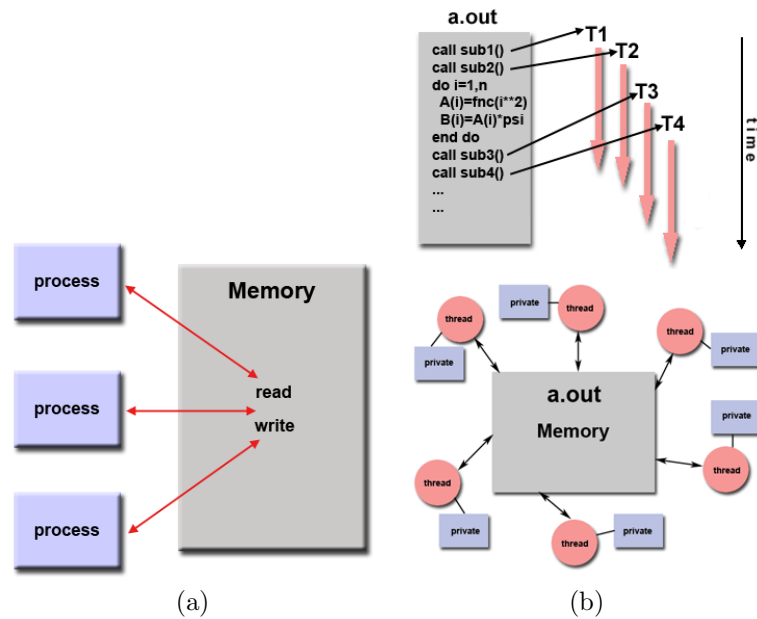
The **thread model** instead is another subtype of shared memory parallel programming. In this model, single *heavy* processes may have several *light* processes that may be executed concurrently. The programmer is in charge of determining the level of parallelism (though sometimes compilers can give some help too), and usually implementations allow him to do it through the use of a library of subroutines, callable from inside the parallel code, and a set of compilers directives embedded in either the serial or parallel source code.

Historically, every hardware vendor implemented its own proprietary version of threads programming, but recent standardization effort have resulted in two different implementations of these models, **POSIX threads** and **Open MP**.

1.5.2 Distributed memory models

All distributed memory models present these three basic features.

- computation is made of separate tasks with their own local memory, residing on the same physical machine or across an arbitrary number of machines;
- data is exchanged between tasks during the computation;
- data transfer require some sort of data exchange protocol.

Figure 1.6: *Shared memory model with and without threads*

For these last two reasons, distributed memory models are usually referred as **message passing models**. They require the programmer to specify and realize the parallelism using a set of libraries and/or subroutines. Nowadays, there is a standard interface for message passing implementations, called **MPI (Message Passing Interface)**, that is the "de-facto" industry standard for distributed programming models.

1.5.3 Hybrid and high level models

Hybrid and high level models are programming models that can be built combining any one of the previously mentioned parallel programming models.

A very commonly used hybrid model in hardware environment of clustered multi-core/processor machines, for example, is the combination of the message passing model (MPI) with a thread model (like OpenMP), so that threads perform computational kernels on local data, and communication between the different nodes occurs using the MPI protocol.

Another noteworthy high level model is the **Single Program Multiple Data (SPMD)**, where all the tasks execute the same program (that may be threads, message passing or hybrid programs) on different data. Usually, the program is logically designed in such a way that the single task executes only a portion of the whole program, thus creating a logical multitasking compu-

tation. The SPMD is probably the most common programming model used in multi-node cluster environments.

1.6 Modern computers: accustomed users of parallel computing

Having given in the previous sections some instruments to broadly classify the various means and forms through which and where a parallel computation can take place, we will now discuss about the **classes** of parallel computers. According to the hardware level at which the parallelism is supported, computers can be approximately classified in:

- **distributed computers:** distributed memory machines in which the preprocessing elements (or computational entities) are connected by a network;
- **cluster computers:** groups of multiple standalone machines connected by a network (e.g. the *Beowulf cluster* [24]);
- **multi-core computers.**

Multi-core computers are systems with **multi-core processors**, an integrated circuit to which two or more processors have been attached for enhanced performance, reading multiple instruction at the same time and increasing overall speed for executing programs, being suitable even for **parallel computations**.

Nowadays, multi-core processors are exploited in almost every average computer or electronic device, leading ourselves to deal with parallel computing even without awareness.

In figure 1.7, we can see how parallelism is exploited in various levels: a single system in a cluster (a standalone machine like commercial off-the-shelf computers) might have several processors, each one executing high performance applications that implement one of the parallel programming models discussed previously.

Every one of these processors might be a multi-core processor, each one of them executing a multithreaded program (see section 1.5.1), with every core performing a thread parallelizing its executing code through parallelized instructions set. Effective implementation can thus exploit parallel processing capabilities at all levels of execution by maximizing the speedup [15]. Moreover, even general purpose machines can make use of parallelism to increase their performance, so that a more insightful comprehension of this topic is

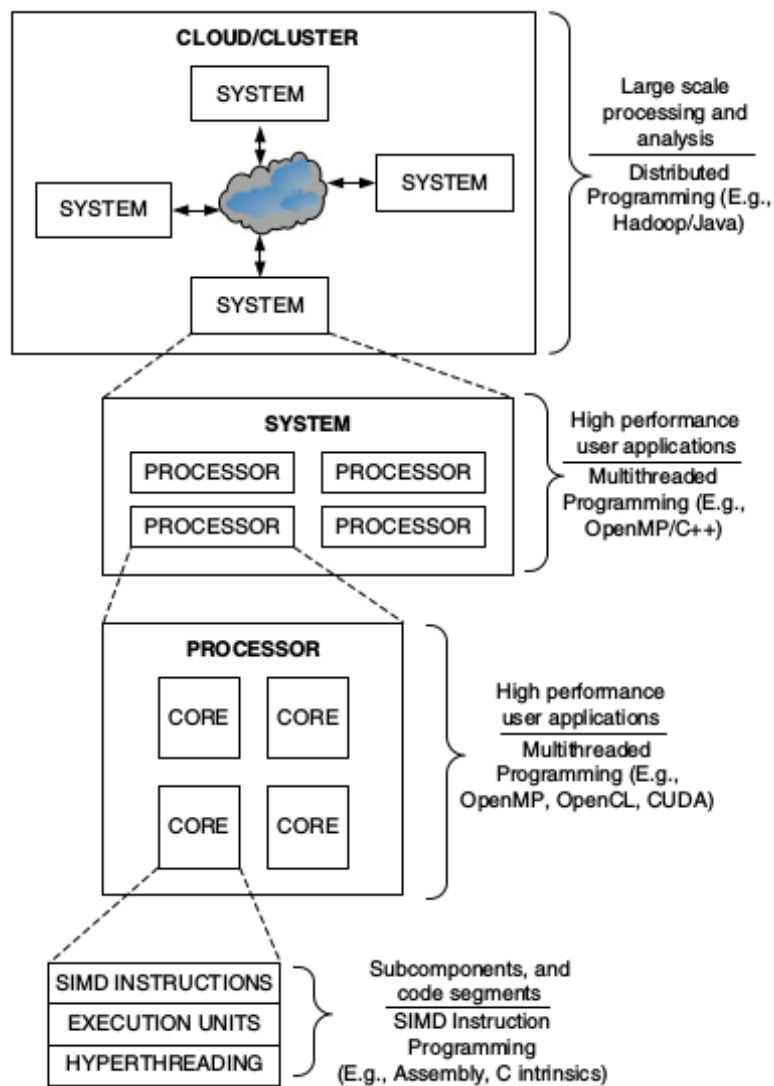


Figure 1.7: *Hierarchy in parallel programming*

now necessary, especially for what is related to **linear algebra** and **scientific computing**.

Chapter 2

Matlab's approach to parallel computing: the PCT

The following chapter aims to describe briefly the features and mechanism through which **MATLAB**, a technical computing language and development environment, exploits the several techniques of *parallel computing* discussed in the previous chapter. Starting from the possibility of having a *parallel computer* for high performance computation, like a *distributed computer* or a *cluster*, using different parallel programming models (i.e. SPMD and or MPI), to the implementation of different parallelism level in the software itself, there are various fashions of obtaining incremented performance through parallel computations.

The information here reported are mainly quoted from [2] and [23].

2.1 Parallel MATLAB

While even in the late 1980s **Cleve Moler**, the original MATLAB author and cofounder of MathWorks, worked on a version of MATLAB specifically designed for parallel computers (Intel HyperCube and Arden Titan in particular ¹), the release of the **MATLAB Parallel Computing Toolbox (PCT)** software and **MATLAB Distributed Computing Server (DCS)** took place only in 2004.

The main reason is that until 2000s, several factors including the business situation, made the effort needed for the design and development of a parallel MATLAB not very attractive ². Anyway, having MATLAB matured into a preeminent technical computing environment, and the access to multipro-

¹Anyone who is interested in this topic may read [16]

²for more information, see [19]

cessor machines become easier, the demand for such a toolbox like the PCT spread towards the user community.

The PCT software and DCS are toolboxes made to extend MATLAB such as a library or an extension of the language do. These two instruments anyway are not the only means available to the user for improving the performance of its code through techniques of parallelization.

In fact, MATLAB supports a mechanism of **implicit** and **explicit** multithreading of computations, and presents a wide range of mechanism to transform the data and make the computation vectorized, so that computation can take advantage of the use of libraries such as BLAS and LAPACK. We will discuss this details later.

However, we can now roughly classify MATLAB parallelism into four categories ([2], pag.285; see [18] too):

- **implicit multithreading**: one instance of MATLAB automatically generate simultaneous multithreaded instruction streams (level 3 and 4 of figure 1.5). This mechanism is employed for numerous built-in functions.
- **explicit multithreading**: through this way, the programmer explicitly creates separate threads from the main MATLAB thread, easing the burden on it. There is no official support to this way of programming, and in fact, it is achieved using MEX functions, innested Java or .Net threads. We will not discuss this method in this thesis ³.
- **explicit multiprocessing**: multiple instances of MATLAB run on several processors, using the PCT constructs and functions to carry out the computation. We will discuss its behaviour and structure in the next sections.
- **distrubuted computing**: multiple instances of MATLAB run independent computation on different computers (such as those of cluster, or a grid, or a cloud), or the same program on multiple data (SPMD). This technique is achieved through the DCS, and it is strictly bounded to the use of the PCT.

³For those who are interested in this topic see [1].

2.2 PCT and MDCS: MATLAB way to distributed computing and multiprocessing

MATLAB Distributed Computing Server and Parallel Computing Toolbox work along as two sides of the same tool. While the DCS works as a library for controlling and interfacing together the computers of a cluster, the PCT is a toolbox that allow the user to construct parallel software.

As we said before, the PCT exploits the advantages offered by a multicore computer allowing the user to implement parallel computing techniques creating multiple instances of MATLAB on several processors.

These instances are called **workers**, or also **headless MATLABs** or **MATLAB computational engines**, and are nothing more than MATLAB processes that run in parallel without GUI or Desktop [2]. The number of workers runned by **the client** (MATLAB principal process, the one the user uses to manage the computation through the GUI) is set by default as the number of processors of the machine. Certain architectures, such as modern Intel CPUs, may create a *logical* core, through the techniques of **hyperthreading**[27], allowing the PCT to assign other processes to both physical and logical cores. Anyway, the user can have more workers than the number of cores of its machine, up to 512 from the R2014a MATLAB release, using the **Cluster Profile Manager** function of the PCT.

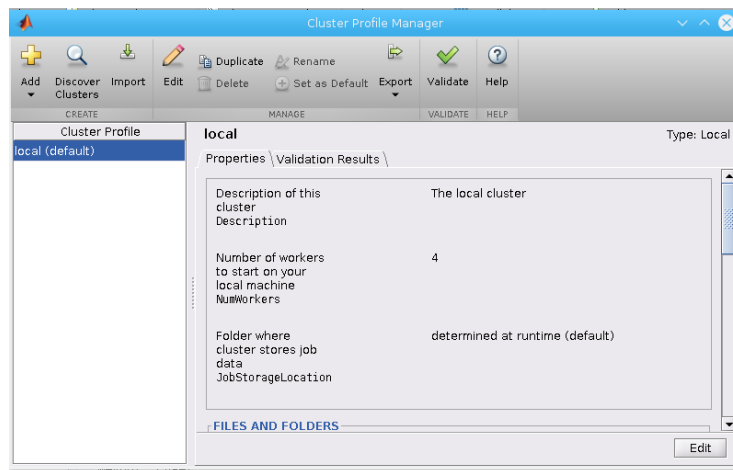


Figure 2.1: Cluster Profile Manager window

The workers can function completely independently of each other, without requiring any communication infrastructure setup between them[23].

Nome	Nome utente	Processore %	Memoria	Memoria condivisa	Titolo della finestra
plasma...	edoardo	2%	99.036 k	85.276 k	Plasma
ksvsuad	edoardo	2%	11.700 k	53.796 k	Monitor di sistema
MATLAB	edoardo	1%	450.284 k	185.972 k	MATLAB R2014b
vlc	edoardo	1%	22.836 k	52.896 k	Bonobo - Black Sands - L...
ksnapshot	edoardo	1%	13.096 k	43.656 k	
textstudio	edoardo		165.956 k	60.776 k	/home/edoardo/Documen...
firefox	edoardo		164.740 k	104.936 k	MATLAB Distributed Com...
evince	edoardo		68.528 k	35.664 k	art%3A10.1007%2Fs107...
evince	edoardo		48.528 k	33.000 k	Accelerating MATLAB Perf...
dolphin	edoardo		16.140 k	57.712 k	Funzionamento PCT — D...
konsole	edoardo		8.824 k	40.476 k	edoardo : bash — Konsole
Xorg	edoardo, root	2%	20.420 k	97.076 k	
kwin_x11	edoardo	2%	33.124 k	59.984 k	
MATLAB	edoardo		211.848 k	137.336 k	
MATLAB	edoardo		210.408 k	138.156 k	
MATLAB	edoardo		209.468 k	136.288 k	
MATLAB	edoardo		193.216 k	137.900 k	
tracker-s...	edoardo		33.024 k	10.720 k	

240 processi Processore: 3% Memoria: 2,3 GiB / 3,8 GiB Swap: 2,1 MiB / 2,9 GiB

Figure 2.2: *Workers processes are delimited by the red rectangle, while the client is delimited by the green rectangle. The computer is an Intel 5200U (2 physical cores double threaded, for a total of 4 cores)*

Anyway, some functions and constructs (like **message passing functions** and **distributed arrays**) may require a sort of communication infrastructure. In this context, we call the workers **labs**.

2.3 Infrastructure: MathWorks Job Manager and MPI

This section aims to describe briefly the implementation details of the PCT and its general operating mechanisms.

The basic concepts of PCT working are the ideas of **jobs** and **tasks**. Tasks are simple function evaluations that can be arranged together sequentially to form a job. Jobs are then submitted to the **Job Manager**, a *scheduler* whose duty is to organize jobs in a queue and dispatch them to the workers which are free and ready to execute them.

This scheduler operates in a Service Oriented Architecture (SOA), fitted in an implementation of JINI/RMI framework, being Matlab based on a JVM and being tasks, jobs and cluster all instances of their specific **parallel.class**. All the data related to a specific task or job (like function code, inputs and outputs, files, etc...) is stored in a relational database with a JDBC driver

```
Command Window
New to MATLAB? See resources for Getting Started.

>> clear
>> c=parcluster('local');
>> j=createJob(c);
>>
>>
>> t = createTask(j, @rand, 1, {{10,10} {10,10} {10,10}});
>>
>> submit(j);
>>
>> wait(j);
>>
>> taskoutpur = fetchOutputs(j);
>>
```

Figure 2.3: *Creation of a job of three equal tasks, submitted to the scheduler*

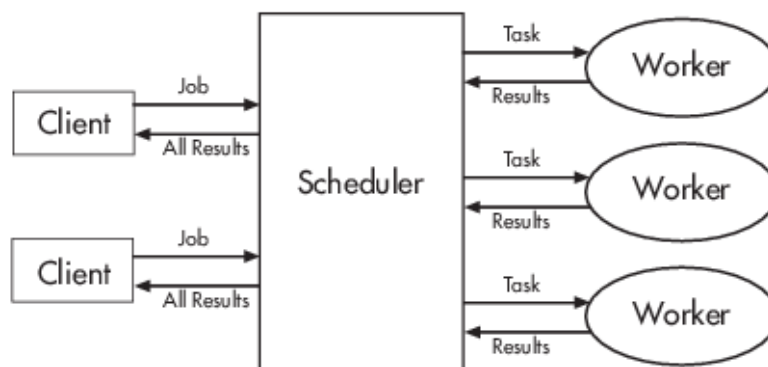


Figure 2.4: *Interaction between the client and the schedulers*

that supports Binary Large Objects (BLOBs). The scheduler is then provided with an API with significant data facilities, to allow both the user and MATLAB to relate with both the database and the workers.

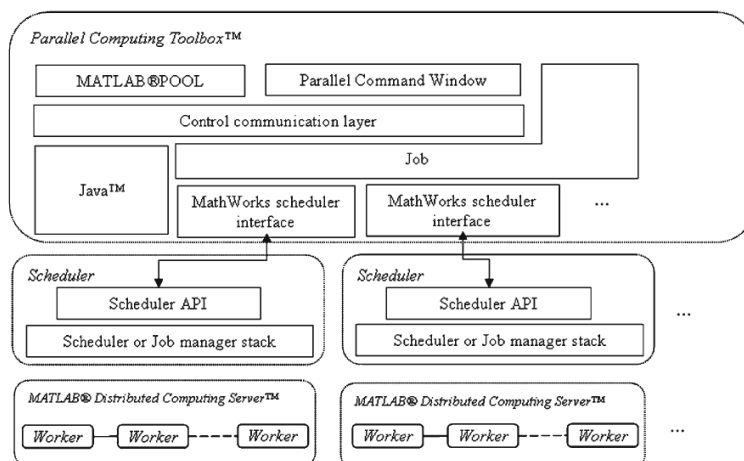


Figure 2.5: Infrastructure for schedulers, jobs and parallel interactive sessions

In fact, all the workers have their separate workspace and can work independently from each other without requiring any communication infrastructure. Having said that, they can still communicate with each other through a **message passing infrastructure**, consisting of two principal layers:

- **MPI Library Layer:** a MPI-2 shared library implementation based on MPICH2 distribution from Argonne National Laboratory, loaded on demand which provides an uniform interface to message passing function in MATLAB and other libraries (like ScaLAPACK);
- **binder layer:** an abstraction layer that provides MATLAB with a standard binary interface to all MPI library functions.

This particular infrastructure is held up and controlled by the scheduler when the user decides to create a **parpool**, a series of MATLAB engine processes in an MPI ring, with a control communication channel back to the MATLAB client. Parpools may comprehend all the workers in the cluster (or all the workers assigned to the processors of our machine) or just few of them, and is started automatically using *parallel constructs* or manually by the user.

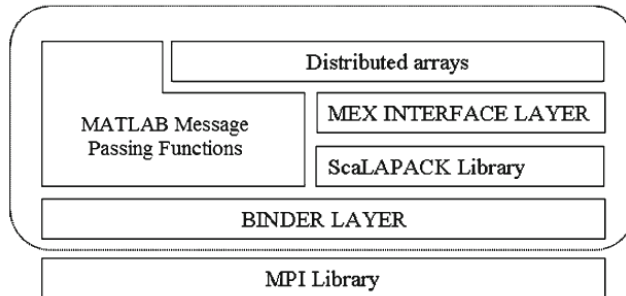


Figure 2.6: Message passing infrastructure

```

Command Window
New to MATLAB? See resources for Getting Started.
>> p=parpool
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

p =

Pool with properties:

    Connected: true
    NumWorkers: 4
    Cluster: local
    AttachedFiles: {}
    IdleTimeout: 30 minute(s) (30 minutes remaining)
    SpmdEnabled: true

fg >> |

```

Figure 2.7: Creation of a parpool: as job and tasks, parpool are instances of objects

```

edoardo@localhost ~]$ ps aux|grep matlab
edoardo 19830 0.0 0.0 14804 1528 pts/2 Ss+ 15:47 0:00 /usr/local/MATLAB/R2014b/bin/glnxa64/matlab_helper /dev/pts/2 yes
edoardo 27598 0.0 0.0 22712 3548 pts/1 S 17:49 0:00 /usr/local/MATLAB/R2014b/bin/glnxa64/mpitexec -configfile /home/edoardo/.matlab/local_cluster_jobs/R2014
by/jobs/mpitexecconfig
edoardo 27705 0.0 0.0 14804 1428 pts/3 Ss+ 17:49 0:00 /usr/local/MATLAB/R2014b/bin/glnxa64/matlab_helper /dev/pts/3 no
edoardo 27707 0.0 0.0 14804 1508 pts/4 Ss+ 17:49 0:00 /usr/local/MATLAB/R2014b/bin/glnxa64/matlab_helper /dev/pts/4 no
edoardo 27709 0.0 0.0 14804 1384 pts/5 Ss+ 17:49 0:00 /usr/local/MATLAB/R2014b/bin/glnxa64/matlab_helper /dev/pts/5 no
edoardo 27714 0.0 0.0 14804 1456 pts/6 Ss+ 17:49 0:00 /usr/local/MATLAB/R2014b/bin/glnxa64/matlab_helper /dev/pts/6 no
edoardo 27836 0.0 0.0 118512 2112 pts/1 S+ 17:50 0:00 grep --color=auto matlab
edoardo@localhost ~]$

```

Figure 2.8: Building and configuration of the MPI ring: the mpiexec config.file (green square) is a configuration file for setting up and running the pool, whose workers are delimited by the blue square

2.4 Some parallel language constructs: **pmode**, **SPMD** and **parfor**

As we stated earlier in the chapter, PCT initial goal was to "extend the MATLAB language with a set of parallel data structures and parallel constructs, which would abstract details as necessary and present to users a language independent of resource allocation and underlying implementation" [23]. This reflected in a series of high-level constructs which allow the user to immediately start using the toolbox without any specific change to its usual working model, yet giving it enough control to what is happening between the workers.

We will discuss shortly two ways of interacting with the PCT: the **pmode**, an interactive command line similar to the normal MATLAB command window, and the **SPMD** and **parfor** constructs.

2.4.1 Parallel Command Window: **pmode**

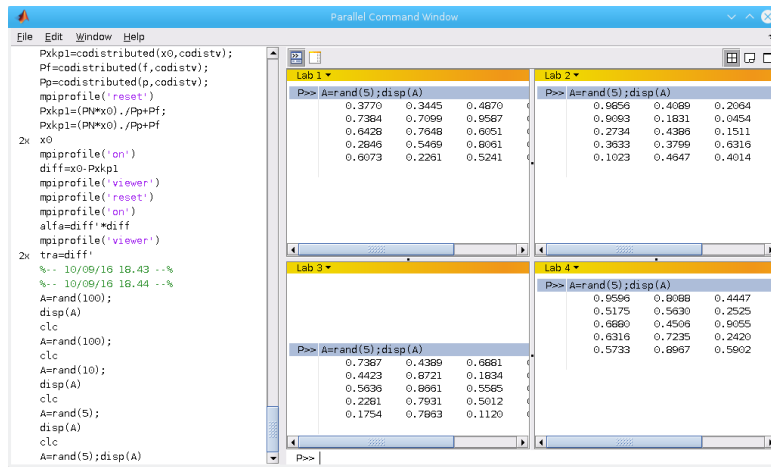
The control communication channel cited earlier when talking about the **parpool**, allow the user to interact with the workers of the pool through a parallel command window which is a counterpart to the normal MATLAB command window. This window provides a command line interface to an SPMD programming model, with the possibility to interrupt and display outputs from all the computational processes.

In fact, the parallel command window has display features that allow the output of the computation to be viewed in several different ways, and its basic functions are the sending and receiving of simple evaluation request messages to and from all labs when the user types a command.

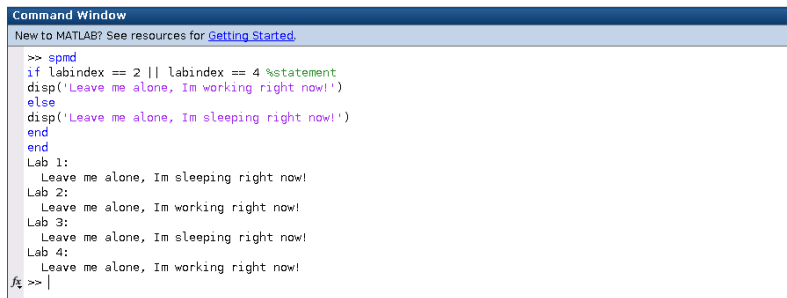
The main problem when using this functionality, is that the parallel command window is not the MATLAB command window: users have to send variables and data with specific functions from the serial workspace to the parallel one (every worker has its separate workspace), and this may be a long and exhausting operation. Anyway, **pmode** is still a great tool for developing and debugging SPMD algorithms, which we will discuss in the next subsection.

2.4.2 SPMD: Single Program Multiple Data

The **spmd** construct implements the Single Program Multiple Data programming model: the statement between the command **spmd** and **end** is executed by the workers of the **parpool** simultaneously, opening a pool if it has not been created yet.

Figure 2.9: *Parallel command window*

Inside the body of the `spmdd` statement, every worker is denoted as a **lab**, and is associated with a specific and unique index. This index allow the user to diversify the execution of the code by the labs using conditional statements (see figure 2.10), but also to use some message passing functions which are high-level abstraction of functions described in the MPI-2 standard.

Figure 2.10: *Use of conditional statements to diversify the execution of the code (parpool of 4 workers)*

These functions comprehend point-to-point and broadcast operations, error and deadlock detection mechanisms, etc..., and are based on a protocol for exchanging arbitrary MATLAB data types (numerical array of any precision, structure arrays, cell arrays, etc...), which consists in sending a first message containing the information about the data, and then a message with the actual payload (which may be serialized in case of non-numeric data types).

Users may use these operations explicitly, or implicitly within the code of other constructs.

For example, trying to reduce the programming complexity by abstracting out the details of message passing. the PCT implements the **PGAS (Partitioned Global Address Space)** model for SPMD through **distributed arrays**. These arrays are partitioned with portions localized to every worker space, and are implemented as MATLAB objects, so that they also store information about the type of distribution, the local indices, the global array size, the number of worker processes, etc...

Their distributions are usually one-dimensional or two-dimensional block cyclic: users may specify their personal data distribution too, creating their own distribution object, through the functions **distributor** and **codistributor**⁴. An important aspect of data distributions in MATLAB is that they are dynamic[23], as the user may change distribution as it likes by redistributing data with a specific function (**redistribute**).

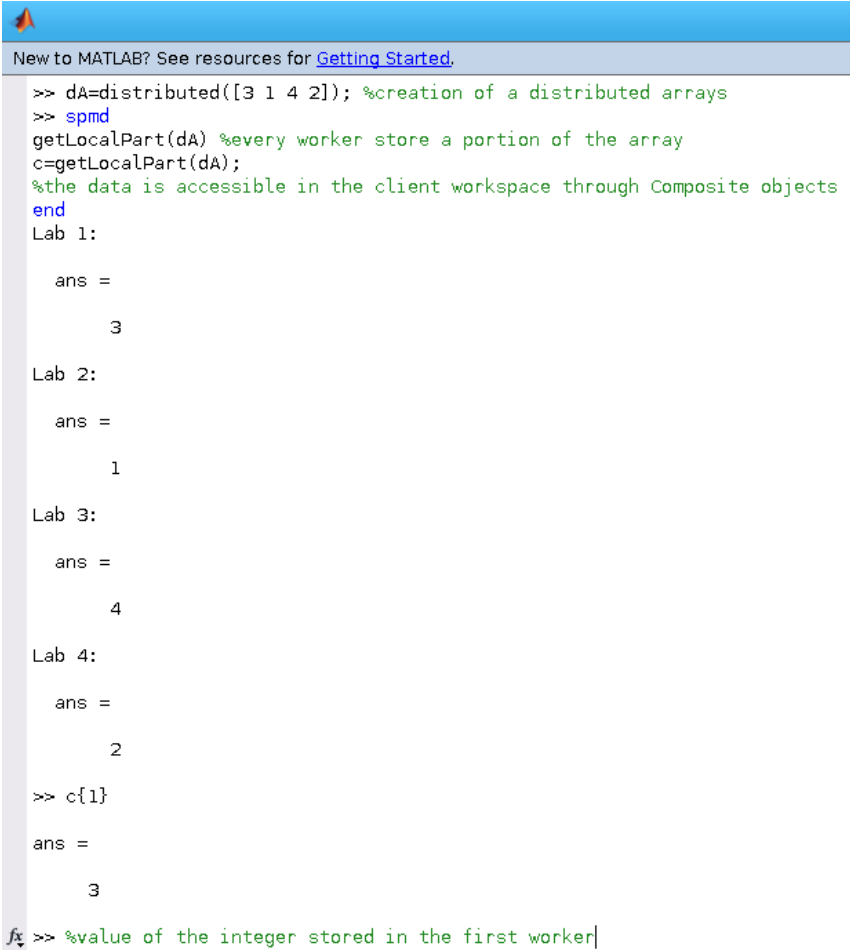
Distributed arrays can be used with almost all MATLAB built-in functions, letting users write parallel programs that do not really differ from serial ones. Some of these functions, like the ones for linear dense algebra, are finely tuned as they are based on the ScaLAPACK interface: ScaLAPACK routines are accessed by distributed array functions in MATLAB without any effort by the user, with the PCT encoding and decoding the data from the ScaLAPACK library⁵. Other algorithms, such as the ones used for sparse matrices, are implemented in the MATLAB language instead.

Anyway, their main characteristic is that their use is easy to the programmer: the access and use of distributed arrays are syntactically equal to regular MATLAB arrays access and use, with all the aspects related to data location and communication between the workers being hidden to the user though they are implemented in the definition of the functions deployed with distributed arrays. In fact, all workers have access to all the portion of the arrays, with all the communication syntax needed to transfer the data from their location to the worker which has not to be specified within the code lines of the user.

The same idea lays behind the **reduction operations**, which allow the user to see the distributed array as a *Composite* object directly in the workspace of the client without any effort by the user.

⁴Please note that *distributed* and *codistributed arrays* are the same arrays but seen from different perspective: a *codistributed array* which exists on the workers is accessible from the client as *distributed*, and viceversa[2]

⁵ScaLAPACK is a library which includes a subset of LAPACK routines specifically designed for parallel computers, written in a SPMD style. For more information, see [9]



```
New to MATLAB? See resources for Getting Started.  
>> dA=distributed([3 1 4 2]); %creation of a distributed arrays  
>> spmd  
getLocalPart(dA) %every worker store a portion of the array  
c=getLocalPart(dA);  
%the data is accessible in the client workspace through Composite objects  
end  
Lab 1:  
ans =  
    3  
Lab 2:  
ans =  
    1  
Lab 3:  
ans =  
    4  
Lab 4:  
ans =  
    2  
>> c{1}  
ans =  
    3  
fx >> %value of the integer stored in the first worker|
```

Figure 2.11: *Some example of distributed arrays*

Thus, the SPMD construct brings the user a powerful tool to elaborate parallel algorithms in a simple but still performing fashion.

2.4.3 Parallel For Loop

Another simple but yet useful instrument to realize parallel algorithms is the **parfor (Parallel For Loop)** construct.

The syntax **parfor i = range; <loopbody> ; end** allows the user to write a loop for a block of code that will be executed in parallel on a pool of workers reserved by the *parpool* command. A *parpool* is initialized automatically if it has not been created yet by submitting the *parfor* command.

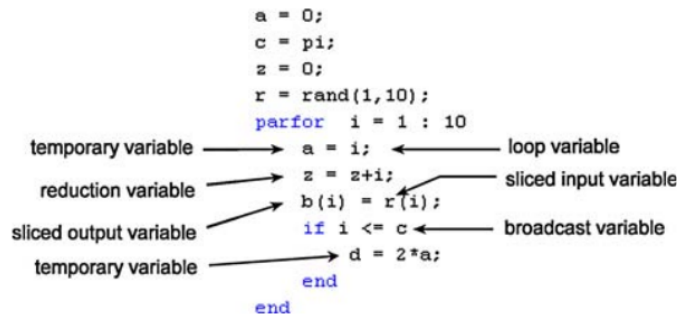
Shortly, when users declare a *for* loop as a *parfor* loop, the underlying execution engine uses the computational resources available in the *parpool* to execute the body of the loop in parallel. In the absence of these resources, on a single processor system, *parfor* behaves like a traditional *for* loop [23]. The execution of the code by the workers is absolutely *asynchronous*: this means that there is no order in the execution of the contents of the loop, so that there are some rules to follow to use the construct correctly.

First of all, the loop must be *deterministic*: its results should not depend on the order in which iterations are executed. This forces the user to use only certain constructs and functions within the *parfor* body, and requires a static analysis of it before its execution.

The main goal of this analysis is to categorize all the variables inside the loop into 5 categories:

- **Loop indexing variables;**
- **Broadcast variables:** variables never target of assignments but used inside the loop; must be sent over from the host;
- **Sliced variables:** indexed variables used in an assignment, either on the right-hand or the left-hand side of it;
- **Reduction variables:** variables that appear at the left-hand side of an unindexed expression and whose values are *reduced* by MATLAB in a non-deterministic order from all the iterations (see section 2.4.2);
- **temporary variables:** variables subjected to simple assignment within the loop.

If anyone of the whole variables of the loop failed to be classified into one of these categories, or the user utilizes any functions that modifies the workers'

Figure 2.12: *Classification of variables in a parfor loop*

workspace in a non-statically way, then MATLAB reports an error.

Transport of code and data for execution is realized through a permanent communication channel setup between the client (which acts as the master in a master-worker pattern) and the workers of the pool (which, indeed, act as the workers), as described in section 2.3. Work and data are encapsulated into a function which can be referenced by a *function handle*. Clumsily, function handles are MATLAB equivalent to pointers to functions in C language; anonymous functions are instead expressions that exist as function handles in the workspace but have no corresponding built-in or user-defined function. Usually, sliced input variables, broadcast variables and index variables are sent as input of this function handle, and sliced output variables are sent back as its outputs. For what concerns the code, if the `parfor` is being executed from the MATLAB command line, the executable package is sent as an anonymous function transported along with the data over the workers; when the `parfor` is being executed within a MATLAB function body, existing as a MATLAB file with appropriate function declaration, this payload is an handle to an automatically generated nested function from the `parfor` body loop, serialized and transmitted over the persistent communication channel described above. The workers then deserialize and execute the code, and return the results as sliced output variables [23].

As SPMD, *parfor* is a very simple construct: yet it cannot be used like a simple for loop, requiring the user some practice to rearrange its algorithms, it can lead to great performance increase with little effort.

2.5 Serial or parallel? Implicit parallelism in MATLAB

Having described in the previous sections the mechanism behind the PCT and some of its construct that allow normal users to approach parallel computing without great labors, it is worth to say that the MathWorks put great effort in implementing mechanisms of implicit parallelization. As we said in the introduction of this chapter, MATLAB presents a wide range of instruments to improve the performance of its computations. Among them, one of the most important is the already mentioned **multithreading**.

This mechanism relies on the fact that "many loops have iterations that are independent of each other, and can therefore be processed in parallel with little effort" [2], so, if the data format is such that it can be transformed as *non-scalar* data, MATLAB tries to parallelize anything that fits this model. The technical details of MATLAB's multithreading are undocumented: it has been introduced in the R2007a version [29], as an optional configuration, becoming definitive in R2008a. We know that it makes use of parallel processing of parts of data on multiple physical or logical CPUs when the computation involves great numbers of elements (20K, 40K, etc...), and it is realized only for a finite number of function⁶.

This useful tool, which operates at level 3 and 4 of parallelism level in programs (see section 1.4), is not the only one helpful to the user to accelerate its code. In fact, MATLAB use today's state-of-the-art math libraries such as PBLAS, LAPACK, ScaLAPACK and MAGMA [17], and for platforms having Intel or AMD CPUs, takes advantages of Intel's Math Kernel Library (MKL), which uses SSE and AVX CPU instructions that are SIMD (Single Instruction Multiple Data) instructions, implementing the level 1 of parallelism level in programs (see figure 1.5), and includes both BLAS and LAPACK⁷.

So, yet the user may not be aware of that, some of his own serial code may be, in fact, *parallelized* in some manner: from multithreaded functions, to SIMD instruction sets and highly-tuned math libraries, there are several levels in which the parallelization of the computing can take place.

This aspect revealed to be very important during the analysis of our algorithms' performance, so we think it has to be described within a specific section. Anyway, we were not able to determine where this mechanism of

⁶An official list of MATLAB's multithreaded functions is available at [29]; a more detailed but unofficial list is available at [7]

⁷For information about the MKL, see [28]; for information about the SSE and AVX set of instructions, see [26]

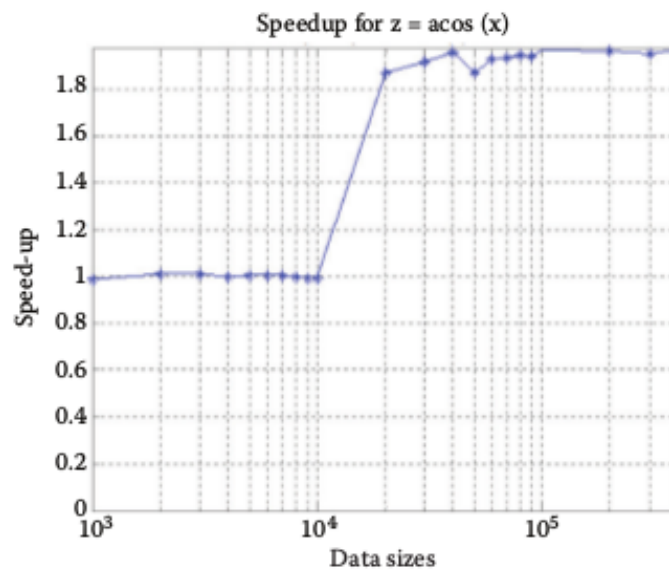


Figure 2.13: *Implicit multithreading speedup for the `acos` function*

implicit parallelization have taken place, since the implementation details are not public.

Chapter 3

Parallelizing the Jacobi method

This chapter and the following one will focus on two linear algebra applications: the **Jacobi iterative method** for solving linear systems and the **Global Lanczos decomposition** used for the computation of the Estrada index of complex networks analysis.

Both these algorithms are theoretically parallelizable, thus, we formulated a serial and a parallel version of them, implemented using the **SPMD** and **parfor** constructs of the PCT. Then, we compared their performance against their serial counterparts, trying to understand and illustrate the reasons behind their behaviour, and highlighting the implications of parallelization within them.

3.1 Iterative methods of first order

The content of the next two sections is quoted from G.Rodriguez's textbook "Algoritmi numerici" [21], and its aim is to recall some basic concepts of linear algebra.

Iterative methods are a class of methods for the solution of linear systems which generates a succession of vectors $x^{(k)}$ from a starting vector $x^{(0)}$ that, under certain assumptions, converges to the problem solution.

We are considering linear systems in the classical form of $Ax = b$, where A is the system matrix, and x and b are the solution and data vector respectively. Iterative methods are useful tools for solving linear systems with large dimension matrices, especially when these are structured or sparsed. In comparison to direct methods, which operate modifying the matrix structure, they do not require any matrix modification and in certain cases their memorization either.

Iterative methods of the first order are methods where the computation of the solution at the step $(k + 1)$ involves just the approximate solution at the step k ; thus, are methods in the form of $x^{(k+1)} = \varphi(x^{(k)})$.

Definition 3.1. We say an iterative method of the first order is *globally convergent* if, for any starting vector $x^{(0)} \in \mathbb{R}^n$, we have that

$$\lim_{k \rightarrow \infty} \|x^{(k)} - x\| = 0$$

where x is the solution of the system and $\|\cdot\|$ identifies any vector norm.

Definition 3.2. An iterative method is *consistent* if

$$x^{(k)} = x \implies x^{(k+1)} = x$$

.

From these definitions, we obtain the following theorem, whose demonstration is immediate.

Theorem 3.1. *Consistency is a necessary, but not sufficient, condition for convergence.*

A **linear, stationary**, iterative method of the **first order** is a method in the form

$$x^{(k+1)} = Bx^{(k)} + f \tag{3.1}$$

where the **linearity** comes from the relation that defines it, the **stationarity** comes from the fact that the **iteration matrix** B and the vector f do not change when varying the iteration index k , and the computation of the vector $x^{(k+1)}$ depends on the previous term $x^{(k)}$ only.

Theorem 3.2. *A linear, stationary iterative method of the first order is consistent if and only if*

$$f = (I - B)A^{-1}b.$$

Demonstration. Direct inspection.

Defining the **error** at k step as the vector

$$e^{(k)} = x^{(k)} - x,$$

where x is the solution of the system, applying the equation 3.1 and the definition 3.2, we obtain the following result

$$e^{(k)} = x^{(k)} - x = (Bx^{(k-1)} + f) - (Bx + f) = Be^{(k-1)} = B^2e^{(k-2)} = \dots = B^k e^{(0)}. \tag{3.2}$$

Theorem 3.3. *A sufficient condition for the convergence of an iterative method defined as 3.1, is that it exists a consistent norm $\|\cdot\|$ so that $\|B\| < 1$.*

Demonstration. From matrix norm properties we have that

$$\|e^{(k)}\| \leq \|B^k\| \cdot \|e^{(0)}\| \leq \|B\|^k \|e^{(0)}\|,$$

thus $\|B\| < 1 \implies \|e^{(k)}\| \rightarrow 0$.

Theorem 3.4. *An iterative method is convergent if and only if the spectral radius $\rho(B)$ of the iteration matrix B is < 1 .*

The demonstration for this theorem can be found at page 119 of [21].

3.2 Additive splitting: the Jacobi method

A common strategy for the construction of linear iterative methods is the one defined as the **additive splitting** strategy.

Basically, it consists of writing the system matrix in the following form

$$A = P - N,$$

where the **preconditioning matrix** P is nonsingular. Replacing A with the previous relation in our linear system, we obtain the following equivalent system

$$Px = Nx + b$$

which leads to the definition of the iterative method

$$Px^{(k+1)} = Nx^{(k)} + b. \tag{3.3}$$

An immediate remark is that a method in such form is consistent; moreover, it can be classified under the previous defined (definition 3.1) category of linear stationary methods of the first order.

In fact, being $\det(P) \neq 0$, we can observe that

$$x^{(k+1)} = P^{-1}Nx^{(k)} + P^{-1}b = Bx^{(k)} + f,$$

with $B = P^{-1}N$ and $f = P^{-1}b$.

From theorem 3.3 and 3.4, a sufficient condition for convergence is that $\|B\| < 1$; in this case $\|P^{-1}N\| < 1$, where $\|\cdot\|$ is any consistent matrix

norm. A necessary and sufficient condition is that¹ $\rho(P^{-1}N) < 1$. Considering the additive splitting

$$A = D - E - F,$$

where

$$D_{ij} = \begin{cases} a_{ij}, & i = j \\ 0 & i \neq j \end{cases}, E_{ij} = \begin{cases} -a_{ij}, & i > j \\ 0 & i \leq j \end{cases}, F_{ij} = \begin{cases} -a_{ij}, & i < j \\ 0 & i \geq j \end{cases},$$

or rather

$$A = \begin{bmatrix} a_{11} & & \\ & \ddots & \\ & & a_{nn} \end{bmatrix} - \begin{bmatrix} 0 & & \\ & \ddots & \\ -a_{ij} & & 0 \end{bmatrix} - \begin{bmatrix} 0 & & -a_{ij} \\ & \ddots & \\ & & 0 \end{bmatrix},$$

the **Jacobi iterative method** presents

$$P = D, \quad N = E + F. \quad (3.4)$$

In this case, being P nonsingular means that $a_{ij} \neq 0, i = 1, \dots, n$. When this condition is not true, if A is nonsingular it must still exist a permutation of its rows that makes its diagonal free of zero-elements. The equation 3.3 then becomes equal to

$$Dx^{(k+1)} = b + Ex^{(k)} + Fx^{(k)},$$

or, expressed with coordinates,

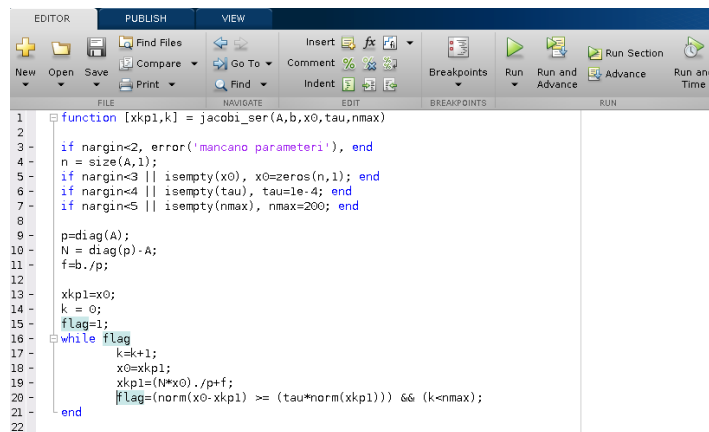
$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} x_j^{(k)} \right], \quad i = 1, \dots, n.$$

From this relation, we can see that the $x^{(k+1)}$ components can be computed from the ones of the $x^{(k)}$ solution in any order independently from each other. In other words, the computation is **parallelizable**.

¹Obviously, for the method to being operative, P must be easier to invert than A .

3.3 Developing the algorithm: `jacobi_ser` and `jacobi_par`

Starting from the theoretical discussion of the previous sections, we have developed two algorithms: a **serial** version of the Jacobi iterative method, and a **parallel** version using the SPMD construct of the PCT.



```

1 function [xkp1,k] = jacobi_ser(A,b,x0,tau,nmax)
2
3 if nargin<2, error('mancano parametri'), end
4 n = size(A,1);
5 if nargin<3 || isempty(x0), x0=zeros(n,1); end
6 if nargin<4 || isempty(tau), tau=1e-4; end
7 if nargin<5 || isempty(nmax), nmax=200; end
8
9 p=diag(A);
10 N = diag(p)-A;
11 f=b./p;
12
13 xkp1=x0;
14 k = 0;
15 flag=1;
16 while flag
17     k=k+1;
18     x0=xkp1;
19     xkp1=(N*x0)./p+f;
20     flag=(norm(x0-xkp1) >= (tau*norm(xkp1))) && (k<nmax);
21 end
22

```

Figure 3.1: *The Jacobi_ser MATLAB function*

Figure 3.1 illustrates our serial MATLAB implementation of the Jacobi iterative method. We chose to implement a sort of *vectorized* version of it, where instead of having a permutation matrix P we have a vector p containing all the diagonal elements of A , and a vector f whose elements are those of the vector data b divided one-by-one by the diagonal elements of A . Thus, the computation of the $x_i^{(k+1)}$ component of the approximated solution at the step $(k + 1)$ is given by

$$x_i^{(k+1)} = \frac{1}{d_i} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} x_j^{(k)} \right] \quad i = 1, \dots, n.$$

Our goal was to make the most out of the routines of PBLAS and LAPACK libraries, trying to give MATLAB a code in the best vectorized manner possible. The matrix N instead, is identical to the one in equation 3.4.

Jacobi_par is the parallel twin of the serial implementation of the Jacobi iterative method of figure 3.1. Like *Jacobi_ser*, it is a MATLAB function which returns the approximate solution and the step at which the compu-

```

9 - p=diag(A);
10 - N = diag(p)-A;
11 - f=b./p;
12 -
13 - c=gcp('ncreate');
14 - if isempty(c)
15 -     c=parpool('local');
16 - end
17 -
18 -
19 - %data distribution and computation are all internal to the spmd body
20 - %spmd
21 - spmd
22 -
23 -     codistm = codistributor1d(1,codistributor1d.unsetPartition,[n,n]);
24 -     %matrix data distribution
25 -     codistv = codistributor1d(1,codistributor1d.unsetPartition,[n,1]);
26 -     %vector data distribution
27 -
28 -     Pp=codistributed(p,codistv);
29 -     Pn=codistributed(N,codistm);
30 -     Pxp1=codistributed(x0,codistv);
31 -     Pf=codistributed(f,codistv);
32 -
33 -     k=0;
34 -     flag=1;
35 -     while flag
36 -         k=k+1;
37 -         x0=gather(Pxp1);
38 -         Pxp1=(Pn*x0)./Pp+Pf;
39 -         flag=(norm(x0-Pxp1) >= (tau*norm(Pxp1))) && (k<nmax);
40 -     end
41 -
42 - end

```

Figure 3.2: *The Jacobi_par MATLAB function*

tation stopped. For both of them, the iterativity of the algorithm is implemented through a **while** construct. We used the **Cauchy criterion of convergence** to stop the iterations, checking the gap between two subsequent approximated solutions with a generic vector norm and a fixed tolerance $\tau > 0$.

Generally, a stop criterion is given in the form of

$$\|x^{(k)} - x^{(k-1)}\| \leq \tau,$$

or in the form of

$$\frac{\|x^{(k)} - x^{(k-1)}\|}{\|x^{(k)}\|} \leq \tau,$$

which considers the relative gap between the two approximated solutions. However, for numerical reasons, we chose to implement this criterion in the form of²

$$\|x^{(k)} - x^{(k-1)}\| \leq \tau \|x^{(k)}\|.$$

To avoid an infinite loop, we also set a maximum number of iterations k , so that our stop criterion is a boolean flag whose value is true if any one of the two expressions is true (the Cauchy criterion or the maximum step number). As we said at the beginning of this section, we used the SPMD construct to develop our parallel version of Jacobi. Figure 3.2 shows that all the com-

²See [21], page 125

putation and data distribution are internal to the SPMD body³. First of all, after having computed our N matrix and our p and f vectors, we put a conditional construct to verify if a parpool was set up and to create one if it has not been done yet. We then realize our own data distribution scheme with the **codistributor1d** function.

This function allows the user to create its own data distribution scheme on the workers. In this case, the user can decide to distribute the data along one dimension, choosing between rows and columns of the matrix with the first argument of the function.

We opted to realize a distribution along **rows**, with equally distributed parts all over the workers, setting the second parameter as **unsetPartition**. With this choice, we left MATLAB the duty to realize the best partition possible using the information about the **global size** of the matrix (the third argument of the function) and the partition dimension.

Figure 3.3 shows a little example of distribution of an identity matrix. As we can see, any operation executed inside the SPMD involving distributed arrays, is performed by the workers on the part they store of it independently. This is the reason why we decided to use this construct to implement the Jacobi method: the computation of the components of the approximated solution at the $(k + 1)$ step could be performed in parallel by the workers by just having the whole solution vector at the prior step k in their workspace (see lines 38–39 of the `Jacobi_par` function in figure 3.2).

To do so, we employed the **gather** function just before the computation of the solution at that step (line 37, figure 3.2).

The **gather** function is a **reduction operation** that allows the user to recollect the whole pieces of a distributed array in a single workspace. Inside an SPMD statement, without any other argument, this function collects all the segments of data in all the workers' workspaces; thus, it perfectly fits our need for this particular algorithm, recollecting all the segments of the prior step solution vector.

3.4 Numerical tests

For our first comparative test, we decided to gauge the computational speed of our `Jacobi_par` script in solving a linear system whose matrix A is a sparse

³Somebody may argue that we should have distributed the data *outside* the SPMD construct, to speed up the computation and lighten the workers' execution load. However, we did not observe any performance increase in doing such a distribution. We suppose that this feature is related to the fact that distributed arrays are an implementation of the PGAS model (see section 2.4.2)


```
>> a=eye(4);
>> spmd
distr=codistributor1d(1,codistributor1d.unsetPartition,[4 4]);
ad=codistributed(a,distr);
ad=labindex*ad;
ad
end
Lab 1:

    This worker stores ad(1,:).

        LocalPart: [1 0 0 0]
        Codistributor: [1x1 codistributor1d]

Lab 2:

    This worker stores ad(2,:).

        LocalPart: [0 2 0 0]
        Codistributor: [1x1 codistributor1d]

Lab 3:

    This worker stores ad(3,:).

        LocalPart: [0 0 3 0]
        Codistributor: [1x1 codistributor1d]

Lab 4:

    This worker stores ad(4,:).

        LocalPart: [0 0 0 4]
        Codistributor: [1x1 codistributor1d]

>> |
```

Figure 3.3: *Distributing and working with distributed arrays*

square matrix of 10000 elements, generated through the **rand** function of MATLAB, using the **tic toc** function.

We developed a function called **diag_dom_spar** whose aim was to made our matrix A diagonally dominant, thus, being sure that our solution would converge⁴. The data vector b was developed by multiplying A times the exact solution vector.

```

Command Window
New to MATLAB? See resources for Getting Started.
>> parpool(4);%parallel pool of 4 workers setup
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
>> main_test
data setup
Jacobi_ser
time=0.001331, iterations=2, error=0
Jacobi_par
time=1.69336, iterations=2, error=0
speedup=0.000702976
>> %test execution and outputs display
fg >> |

```

Figure 3.4: *Display outputs of our comparative test*

All tests have been executed on an **Intel(R) Xeon(R) CPU E5-2620** (2 CPUs, 12 physical cores with Intel hypertexting, for a total of 24 physical and logical cores), varying the number of workers in the parpool from a minimum of 1 to a maximum of 24.

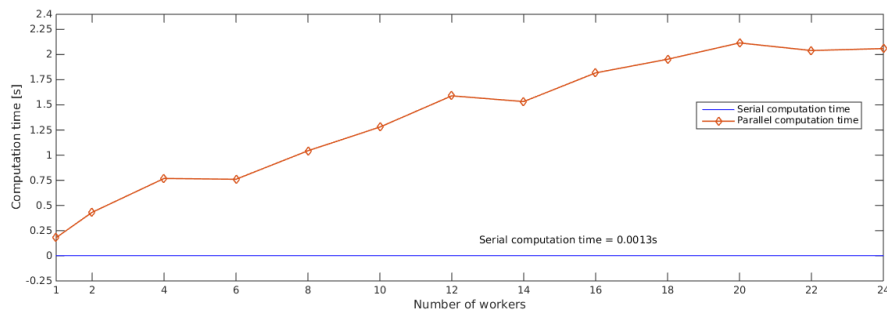


Figure 3.5: *Computation time for Jacobi_ser and Jacobi_par*

As we can see from figure 3.5, our serial implementation revealed to be *always faster* than our parallel one. In fact, *Jacobi_par* produced slower computations and decreasing performances with increasing number of workers. Looking for an explanation for this behaviour, we decided to debug our algorithm using the **MATLAB Parallel Profiler** tool.

MATLAB Parallel Profiler is nothing different from the regular MATLAB

⁴See [22] page 109 for a demonstration of this statement.

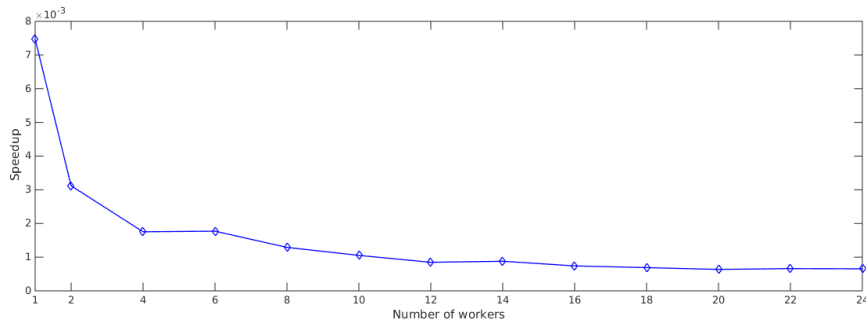


Figure 3.6: Speedup increase (serial-parallel execution time ratio)

Profiler: it just works on parallel blocks of code enabling us to see how much time each worker spent evaluating each function, how much time was spent on communicating or waiting for communications with the other workers, etc...⁵

```

Command Window
New to MATLAB? See resources for Getting Started.

>> spmd
mpiprofile('on') %activating the parallel profiler
[x,k] = jacobi_par(A,b,x0,tau,nmax); %executing the algorithmt
mpiprofile('off') %deactivating the profiler
info=mpiprofile('info'); %gathering the information from the workers
end
>> mpiprofile('viewer',[info{:}])
>> %opening the profiler to inspect the gathered information
fx >> |

```

Figure 3.7: Using the parallel profiler

An important feature of this instrument is the possibility to explore all the children functions called inside the execution of a body function.

Thus, inside the *jacobi_par* function, we identified the two major bottlenecks being the execution of the SPMD body and the gathering operation of the previous step approximated solution.

The SPMD body involves a series of operations with codistributed arrays, such as element-wise operation, norm computation, etc... that are, for now, implemented as methods of MATLAB objects[12]. The overhead of method

⁵See [2], pag. 309 for an exhaustive discussion.

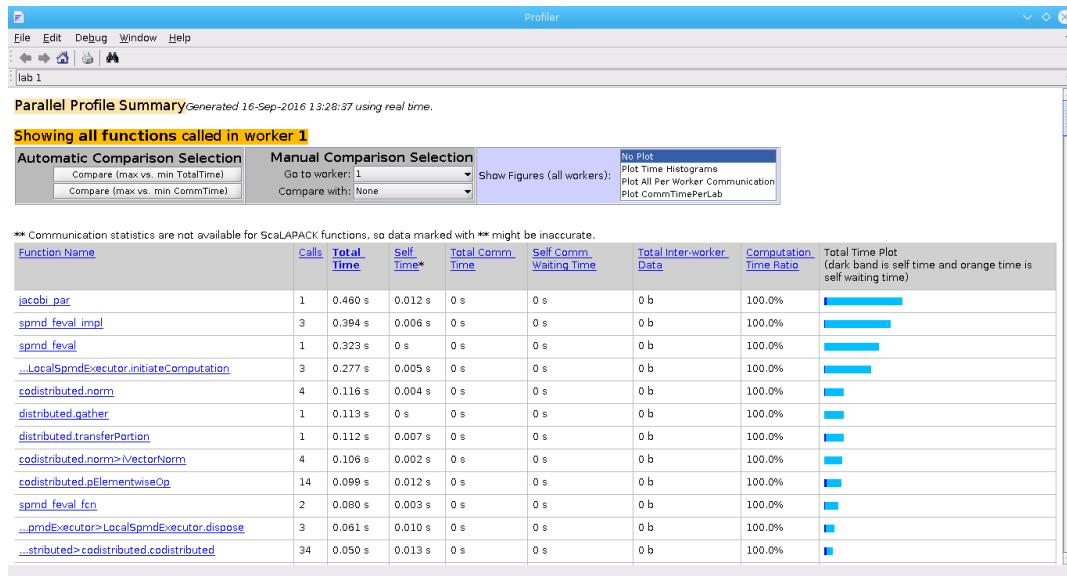


Figure 3.8: MATLAB Parallel Profiler

dispatching for these operations may be quite large, especially if compared to the relatively little amount of numerical computation required for small data sets. Even in absence of communication overheads, as it happens in this case where the computation of the different components of the solution is executed independently by each worker, the parallel execution may reveal to be rather a lot slower than the serial. Thus, increasing the number of workers worsens this situation: it leads to more subdivision of the computation, with related calls to slow data operations, and more communication overheads in gathering the data across all the computational engines for the next iteration.

Lines where the most time was spent.

Line Number	Code	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
13	spmd	1	0.327 s	0 b	0 b	0 s	0 s	71.1%	
35	x = gather(Pxkp1);	1	0.113 s	0 b	0 b	0 s	0 s	24.6%	
36	k = k{1};	1	0.008 s	0 b	0 b	0 s	0 s	1.7%	
10	N = diag(p) - A;	1	0.001 s	0 b	0 b	0 s	0 s	0.2%	
3	if nargin<2, error('mancano...');	1	0.001 s	0 b	0 b	0 s	0 s	0.2%	
All other lines			0.010 s	0 b	0 b	0 s	0 s	2.2%	
Totals			0.460 s	0 b	0 b	0 s	0 s	100%	

Figure 3.9: Jacobi_par time spending functions

Therefore, in this context the parallelization realized through the SPMD con-

Children (called functions)

Function Name	Function Type	Calls	Total Time	Data Sent	Data Rec	Comm Waiting Time	Active Comm Time	% Time	Time Plot
codistributed.norm	function	4	0.116 s	0 b	0 b	0 s	0 s	41.9%	
codistributed.mtimes	function	2	0.033 s	0 b	0 b	0 s	0 s	11.9%	
...istributed>codistributed.codistributed	subfunction	4	0.024 s	0 b	0 b	0 s	0 s	8.7%	
sprnd feval fcn>get f/body	nested function	2	0.019 s	0 b	0 b	0 s	0 s	6.9%	
codistributed.minus	function	2	0.018 s	0 b	0 b	0 s	0 s	6.5%	
codistributed.rdivide	function	2	0.016 s	0 b	0 b	0 s	0 s	5.8%	
...tor1d>codistributor1d.codistributor1d	subfunction	2	0.014 s	0 b	0 b	0 s	0 s	5.1%	
codistributed.plus	function	2	0.012 s	0 b	0 b	0 s	0 s	4.3%	
codistributed.gather	function	2	0.011 s	0 b	0 b	0 s	0 s	4.0%	
sprnd feval fcn>get funpack inputs	nested function	2	0.008 s	0 b	0 b	0 s	0 s	2.9%	
...dExecutor>AbstractSprndExecutor.unpack	subfunction	7	0.001 s	0 b	0 b	0 s	0 s	0.4%	
Self time (built-ins, overhead, etc.)			0.005 s	0 b	0 b	0 s	0 s	1.8%	
Totals			0.277 s	0 b	0 b	0 s	0 s	100%	

Figure 3.10: *Codistributed arrays operations: as we can see, they are all implemented as methods of the `codistributed.class`*

struct and the codistributed arrays did not perform in an efficient manner: in fact, to have some performance benefit with codistributed arrays, they should work with "data sizes that do not fit onto a single machine" [12].

Anyhow, we have to remember that our serial implementation may, in fact, *not* be serial: going beyond the sure fact that the BLAS and LAPACK routines are more efficient than the MATLAB `codistributed.class` operations, we cannot overlook the possibility that our computation may have been multi-threaded by MATLAB's implicit multithreading mechanisms, or our program executed by parallelized instruction sets (such as the Intel's MKL), being our machine equipped with an Intel processor. Obviously, without any other type of information besides the one reported in section 2.5 and the computation time measured, we cannot identify if one or more or none of these features played a role inside our test.

Chapter 4

Computing the trace of $f(\mathbf{A})$

This last chapter focuses on the parallelization of the computation of upper and lower bounds for the Estrada index of complex networks analysis using the Global Lanczos decomposition. We will spend some few words about this topic in the next section to acclimatize the reader with the basic concepts of it.

4.1 Global Lanczos decomposition

The *global Lanczos decomposition* is a recent computational technique developed starting from the standard **Lanczos block decomposition**. A comprehensive discussion about this argument can be found in [6] and in [11]. The Lanczos iteration is a **Krylov subspace projection method**. Given a matrix $A \in \mathbb{R}^{n \times n}$ and a vector $b \in \mathbb{R}^n$, for $r = 1, 2, \dots$, a Krylov subspace with index r generated starting from A and b , is a mathematical set defined as

$$\mathcal{K}_r := \text{span}(\mathbf{b}, A\mathbf{b}, \dots, A^{r-1}\mathbf{b}) \subseteq \mathbb{R}^n. \quad (4.1)$$

There are several algorithms that exploits Krylov's subspaces, especially for the resolution of linear systems.

The **Arnoldi iteration algorithm**, for example, builds up an orthonormal basis $\{\mathbf{q}_1, \dots, \mathbf{q}_l\}$ for the \mathcal{K}_l subspace, replacing the numerically unstable $\{\mathbf{b}, A\mathbf{b}, A^{l-1}\mathbf{b}\}$ basis. For doing so, it makes use of the *Gram-Schmidt* orthogonalization method (see [21], page 144), being l the number of iteration of the algorithm.

The final result of this method, is a $H_l \in \mathbb{R}^{l \times l}$ matrix defined as

$$H_l = Q_l^T A Q_l \quad (4.2)$$

where the Q_l matrix is equal to $Q_l = [\mathbf{q}_1 \quad \mathbf{q}_2 \quad \dots \quad \mathbf{q}_l] \in \mathbb{R}^{n \times l}$.

Annotation 4.0.1. The matrix

$$H_L = Q_l^T A Q_l$$

is the representation in the orthonormal base $\{\mathbf{q}_1, \dots, \mathbf{q}_l\}$ of the orthogonal projection of A on \mathcal{K}_l . In fact, being $y \in \mathbb{R}^l$, $Q_l \mathbf{y}$ indicates a generic vector of \mathcal{K}_l , as it is a linear combination of its given base.

The **Lanczos method** applies the Arnoldi algorithm to a symmetric matrix A . In this way, the H_l matrix of equation 4.2 becomes tridiagonal, and it is given in the form of

$$H_l = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \ddots & & \\ & \ddots & \ddots & \beta_{l-1} & \\ & & \beta_{l-1} & \alpha_l & \end{bmatrix}.$$

This particular matrix structure leads to a reduction in the computational complexity of the whole algorithm ¹.

Now, the **block Lanczos method** is similar to the standard Lanczos iteration. Starting from a symmetric matrix A of dimension n , and, instead of a vector b , a matrix W of size $n \times k$, with $k \ll n$, the result of this algorithm is a partial decomposition of the matrix A in the form of

$$A[V_1, V_2, \dots, V_l] = [V_1, V_2, \dots, V_l, V_{l+1}]T_{l+1,l}, \quad (4.3)$$

where we obtain a matrix \mathcal{V} whose $V_j \in \mathbb{R}^{n \times k}$ blocks are an orthonormal base for the Krylov subspace generated by A and W defined as

$$\mathcal{K}_l := \text{span}(W, AW, A^2W, \dots, A^{l-1}W) \subseteq \mathbb{R}^{n \times k}.$$

The only differences between this method and the standard one are that:

- the internal product of \mathcal{K}_l is defined as $\langle W_1, W_2 \rangle := \text{trace}(W_1^T W_2)$;
- the norm induced by the scalar product is the Frobenius norm

$$\|W_1\|_F := \langle W_1, W_1 \rangle^{1/2}.$$

The **Global Lanczos method** is very similar to the regular block Lanczos method. The only difference lies in the fact that the orthonormality of the

¹We suggest the reader who is interested in this topic to see [21] from page 141 to page 149, for demonstrations, implementations and implications of these algorithms.

\mathcal{V} base is required only between the blocks of which it is made of, while, in the standard block Lanczos iteration, every column of every block has to be orthonormal within each other.

In this case, the matrix T_l becomes a tridiagonal symmetric matrix in the form of

$$T_l = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_l \\ & & & \beta_l & \alpha_l \end{bmatrix};$$

the matrix T_{l+1} instead, is the matrix T_l plus a row of null-elements besides the last one, which is equal to β_{l+1} . Thus, it is equal to

$$T_{l+1,l} = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_l \\ & & & \beta_l & \alpha_l \\ & & & & \beta_{l+1} \end{bmatrix}. \quad (4.4)$$

It is immediately remarkable that T_l , even if of l dimension (with l being the number of iterations of the algorithm), is rather smaller in dimension with respect to the starting A matrix, yet being its eigenvalues a good approximation of A 's extremal eigenvalues. Moreover, it can be used to express T_{l+1} as

$$T_{l+1} = T_l \otimes I_k,$$

being k the initial block-vector dimension.

This relation is quite important for the application of this technique to the resolution of linear systems. For this purpose, starting from [6], in [10] we developed an algorithm (in collaboration with Anna Concas), which takes a generic block W as input together with the matrix A , and takes into account the possibility that certain coefficients may present a value which is minor than a fixed tolerance threshold, leading to a **breakdown** situation. A breakdown occur is very useful in the application of the Global Lanczos iterations to the resolution of linear systems: roughly speaking, it denotes that the approximated solution found at the j step is the exact solution of the linear system².

²For a more detailed explanation see [21], page 142.

Algorithm 1 Global Lanczos decomposition method

-
- 1: **Input:** symmetric matrix $A \in \mathbb{R}^{n \times n}$, block-vector $W \in \mathbb{R}^{n \times k}$
 - 2: algorithm iteration number l , τ_β breakdown threshold
 - 3: $\beta_1 = \|W\|_F$, $V_1 = W/\beta_1$
 - 4: **for** $j = 1, \dots, l$
 - 5: $\tilde{V} = AV_j - \beta_j V_{j-1}$, $\alpha_j = \langle V_j, \tilde{V} \rangle$
 - 6: $\tilde{V} = \tilde{V} - \alpha_j V_j$
 - 7: $\beta_{j+1} = \|\tilde{V}\|_F$
 - 8: **if** $\beta_{j+1} < \tau_\beta$ **then** exit for breakdown **end**
 - 9: $V_{j+1} = \tilde{V}/\beta_{j+1}$
 - 10: **end for**
 - 11: **Output:** Global Lanczos decomposition 4.3
-

4.2 Gauss quadrature rules and Global Lanczos

In [13], Golub and Meurant showed links between the **Gauss-quadrature rules** and the block Lanczos decomposition. In [6], the authors extended these bonds between quadrature rules and the Global Lanczos decomposition. Defining $\mathcal{I}f := \text{trace}(W^T f(A)W)$, we can say that it is equal to

$$\|W\|_F^2 \int f(\lambda) d\mu(\lambda),$$

with $\mu(\lambda) := \sum_{i=1}^k \mu_i(\lambda)$; we can then define the following Gauss-quadrature rules

$$\mathcal{G}_l = \|W\|_F^2 \mathbf{e}_i^T f(T_l) \mathbf{e}_i, \quad \mathcal{R}_{l+1, \zeta} = \|W\|_F^2 \mathbf{e}_i^T f(T_{l+1, \zeta}) \mathbf{e}_i,$$

being T_l the result matrix of the Global Lanczos decomposition of matrix A with initial block-vector W .

These rules are exact for polynomial of degree $2l - 1$ and $2l$, so that, if the derivatives of f behave nicely and ζ is suitably chosen, we have

$$\mathcal{G}_{l-1}f < \mathcal{G}_l f < \mathcal{I}f < \mathcal{R}_{l+1, \zeta} < \mathcal{R}_{l, \zeta} f ;$$

thus, we can determine the upper and lower bounds of the function $\mathcal{I}f$.

Let $G = [\mathcal{V}, \varepsilon]$ be a graph defined by a set of nodes \mathcal{V} and a set of edges ε . The adjacency matrix associated with G is the matrix $A = [a_{ij}] \in \mathbb{R}^{m \times m}$

defined by $a_{ij} = 1$ if there is an edge from node i to node j , and $a_{ij} = 0$ otherwise [6]. If G is undirected, then A is symmetric.

Let us define the **Estrada index** of graph G as

$$\text{trace}(\exp(A)) = \sum_{i=1}^m |\exp(A)|_{ii} = \sum_{i=1}^m \exp(\lambda_i),$$

where $\lambda_i, i = 1, \dots, m$ denotes the eigenvalues of the adjacency matrix A associated with the graph. This index gives a global characterization of the graph, and is used to express meaningful quantities about it [6].

Being G undirected, and thus A symmetric, we can exploit the bounds for $\text{trace}(W^T f(A)W)$ to approximate the Estrada index by

$$\text{trace}(\exp(A)) = \sum_{j=1}^{\tilde{n}} \text{trace}(E_j^T \exp(A) E_j), \quad (4.5)$$

where

$$E_j = [\mathbf{e}_{(j-1)k+1}, \dots, \mathbf{e}_{\min\{jk, n\}}], j = 1, \dots, \tilde{n}; \tilde{n} := \lceil \frac{n+k-1}{k} \rceil.$$

This result is rather remarkable, as it shows that there is a level of *feasible parallelization*.

Besides the computational speed increase given by the use of the Global Lanczos decomposition in the evaluation of the trace at each step of the summation, *each term of it* can, in fact, **be computed independently**.

4.3 Developing the algorithm: `par_trexpгаuss`

In [6], M.Bellalij and colleagues developed an algorithm for the serial computation of the Estrada index using the Global Lanczos decomposition, whose basic functioning is illustrated in Algorithm 2.

Of this algorithm, the authors created a MATLAB implementation, whose performances have been tested in [6] on the computing of the index of 6 undirected networks adjacency matrices obtained from real world applications, against the standard Lanczos method and the `expm` function of MATLAB. This work was the starting point for the development of our **parallel** version of this algorithm, whose central aspect is the use of the **parfor** construct of the Parallel Computing Toolbox.

As we can see from figure 4.1, in Reichel and Rodriguez's implementation of Algorithm 2, the computation of the Estrada index (equation 4.5) is obtained

```

46 - | if k == 1
47 - |     for i = 1:m
48 - |         bounds = [];
49 - |         w(i) = 1;
50 - |         [sg infog] = gaussexp(A,W,theta,opts,show);
51 - |         trc = trc + sg;
52 - |         info = info || infog(1);
53 - |        .mvp =.mvp + infog(2);
54 - |         w(i) = 0;
55 - |     end
56 - |     iters =.mvp;
57 - | else
58 - |     lstblk = floor((m+k-1)/k);
59 - |     bounds = cell(lstblk,1);
60 - |     for i = 1:lstblk
61 - |         if i < lstblk
62 - |             w((i-1)*k+1:i*k,:) = eye(k);
63 - |         else
64 - |             w = w(:,1:m-(i-1)*k);
65 - |             w((i-1)*k+1:m,:) = eye(m-(i-1)*k);
66 - |         end
67 - |         [tw infow bounds{i}] = gausstrex(A,w,theta,opts,show);
68 - |         trc = trc + tw;
69 - |         info = info || infow(1);
70 - |         iters = iters + infow(2);
71 - |        .mvp = MVP + infow(3);
72 - |         if i < lstblk
73 - |             w((i-1)*k+1:i*k,:) = zeros(k);
74 - |         end
75 - |     end
76 - | end

```

Figure 4.1: MATLAB implementation of Algorithm 2: the **for** cycle inside the red rectangle is the implementation of relation 4.5

Algorithm 2 Approximation of $\text{trace}(W^T f(A)W)$ by Gauss-type quadrature based on the global Lanczos algorithm

```

1: Input: symmetric matrix  $A \in \mathbb{R}^{n \times n}$ , function  $f$ 
2:         block-vector  $W \in \mathbb{R}^{n \times k}$ , constants:  $\zeta, \tau, N_{max}, \varepsilon$ 
3:  $V_0 = 0, \beta_1 = \|W\|_F, V_1 = W/\beta_1$ 
4:  $\ell = 0, flag = \mathbf{true}$ 
5: while flag and ( $\ell < N_{max}$ )
6:    $\ell = \ell + 1$ 
7:    $\tilde{V} = AV_\ell, \alpha_\ell = \text{trace}(V_\ell^T \tilde{V})$ 
8:    $\tilde{V} = \tilde{V} - \beta_j V_{\ell-1} - \alpha_j V_\ell$ 
9:    $\beta_{j+1} = \|\tilde{V}\|_F$ 
10:  if  $\beta_{j+1} < \tau\beta$ 
11:     $T = \mathbf{tridiag}([\alpha_1, \dots, \alpha_\ell], [\beta_2, \dots, \beta_\ell])$ 
12:     $\mathcal{G}_\ell = [f(T)]_{11}$ 
13:     $\mathcal{R}_{\ell+1} = \mathcal{G}_\ell$ 
14:    break // exit the loop
15:  else
16:     $V_{\ell+1} = \tilde{V}/\beta_{\ell+1}$ 
17:  end if
18:   $\tilde{\alpha}_{\ell+1} = \zeta - \beta_{\ell+1}P_{\ell-1}(\zeta)/P_\ell(\zeta)$  //  $P_{\ell-1}, P_\ell$  orthogonal polynomials
19:   $T = \mathbf{tridiag}([\alpha_1, \dots, \alpha_\ell], [\beta_2, \dots, \beta_\ell])$ 
20:   $\mathcal{G}_\ell = [f(T)]_{11}$ 
21:   $T_\zeta = \mathbf{tridiag}([\alpha_1, \dots, \alpha_\ell, \tilde{\alpha}_{\ell+1}], [\beta_2, \dots, \beta_{\ell+1}])$ 
22:   $\mathcal{R}_{\ell+1} = [f(T_\zeta)]$ 
23:  flag =  $|\mathcal{R}_{\ell+1} - \mathcal{G}_\ell| > 2\tau|\mathcal{G}_\ell|$ 
24: end while
25:  $\mathcal{F} = \frac{1}{2}\beta_1^2(\mathcal{G}_\ell + \mathcal{R}_{\ell+1})$ 
26: Output: Approximation  $\mathcal{F}$  of  $\text{trace}(W^T f(A)W)$ ,
27:         lower and upper bounds  $\beta_1^2\mathcal{G}_\ell$  and  $\beta_1^2\mathcal{R}_{\ell+1}$ 
28:         for suitable functions  $f$ , number of iterations  $\ell$ 

```

through a **for** cycle where each term of the summation is evaluated *serially*,

and then summed to the other terms in the `trc` variable (line 70 of figure 4.1). The single terms computation, which is an evaluation of the trace of an exponential function through the Global Lanczos decomposition, is realized through the `gausstrexp` function, which uses an algorithm similar to Algorithm 1 for obtaining the decomposition.

As we have already highlighted the feasible parallelizability of the computation of the Estrada index in equation 4.5, the actual MATLAB implementation of it presents itself as a perfect candidate for the use of the `parfor` construct.

```

46 - | if k == 1
47 - |     for i = 1:m
48 - |         bounds = [];
49 - |         W(i) = 1;
50 - |         [sg infog] = gaussexp(A,W,theta,opts,show);
51 - |         trc = trc + sg;
52 - |         info = info || infog(1);
53 - |        .mvp =.mvp + infog(2);
54 - |         W(i) = 0;
55 - |     end
56 - |     iters =.mvp;
57 - | else
58 - |     lstblk = floor((m+k-1)/k);
59 - |     bounds = cell(lstblk,1);
60 - |     parfor i = 1:lstblk
61 - |         W=zeros(m,k);
62 - |         if i<lstblk
63 - |             W((i-1)*k+1:i*k,:) = eye(k);
64 - |         else
65 - |             W = W(:,1:m-(i-1)*k);
66 - |             W((i-1)*k+1:m,:) = eye(m-(i-1)*k);
67 - |         end
68 - |         [tw infow bounds{i}] = gausstrexp(A,W,theta,opts,show);
69 - |         trc = trc + tw; %reduction variable
70 - |         %info = info || infow(1);
71 - |         iters = iters + infow(2); %reduction variable
72 - |        .mvp =.mvp + infow(3); %reduction variable
73 - |         if i<lstblk
74 - |             W((i-1)*k+1:i*k,:) = zeros(k);
75 - |         end
76 - |     end
77 - | end
78 -

```

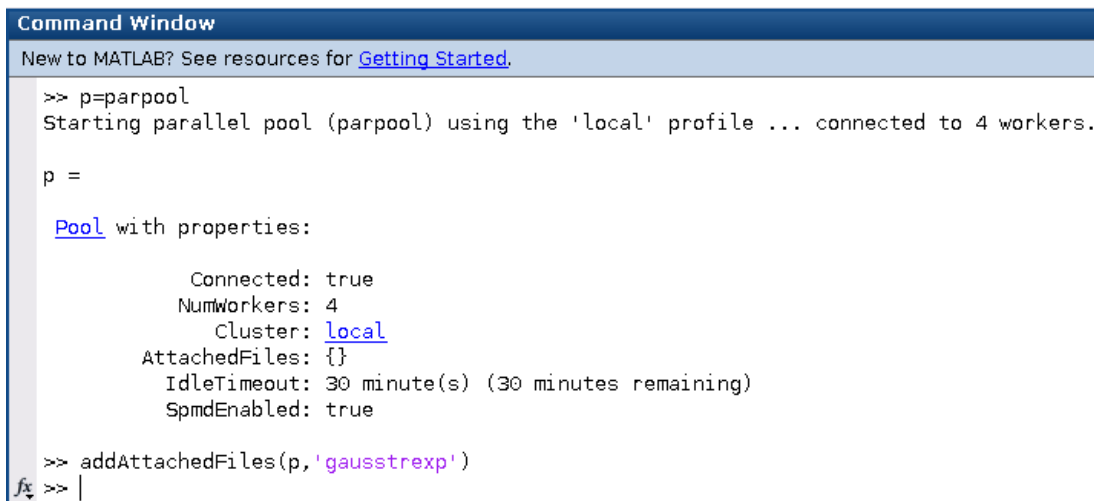
Figure 4.2: *MATLAB parallel implementation of algorithm 2*

In fact, the computation of the summation in its serial implementation needs

just few syntax adjustments to be performed in parallel through the `parfor` construct, since basically it just needs the single terms of the summation to be evaluated by each worker.

First of all, since we cannot use distributed arrays, we have to allocate each E_j block for the Global Lanczos decomposition; thus we have to create them in each worker's workspaces at every iteration (line 61 to 67 of figure 4.2). Since this operation is equal to the creation of a full-zero elements matrix, excepts for a square block of dimension k identical to an identity matrix of the same dimensions, it does not really heavy the computational complexity of the whole algorithm.

The second operation needed, is the delivery of the `gausstrexp` function to every workers: since this is not a MATLAB built-in function, we have to transmitt it over their workspaces before the computation. To do so, we use the `addAttachedFiles` function, which allows the user to send to the workers any other file necessary for his own software to be executed.



```

Command Window
New to MATLAB? See resources for Getting Started.

>> p=parpool
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

p =

Pool with properties:
    Connected: true
    NumWorkers: 4
    Cluster: local
    AttachedFiles: {}
    IdleTimeout: 30 minute(s) (30 minutes remaining)
    SpmdEnabled: true

>> addAttachedFiles(p,'gausstrexp')
fx >> |

```

Figure 4.3: Use of the `addAttachedFiles` function

There is no need to any other code modification.

In fact, the computation of the summation can be performed independently since the `trc` variable (line 69, figure 4.2) is interpreted by MATLAB as a *reduction* variable (see section 2.4.2). This means that the trace computation of the single terms of the summation can be performed in parallel by the worker, while the recollection and execution of the definitive sum of each term is executed by the client without any effort by the user to explicitly declare this operation in his code. The same thing happens for the number of iterations and the other outputs returned by the `gausstrexp` function.

4.4 Numerical tests

For our second comparative test, we have decided to conduct the same test executed in [6], with the same undirected networks, challenging the performances of the serial algorithm against those of the parallel one.

As we have said before, all networks are obtained from real world applications, and we have computed their Estrada index with the optimal block dimension k_{min} found in the [6] test, a large enough N_{max} parameter to not affect the computations, and a stop tolerance $\tau = 10^{-3}$. This means that the iterations are terminated when the relative distance between the computed upper and lower bounds for the trace are sufficiently close [6].

The adjacency matrices are all stored using MATLAB's sparse storage format; for the four smallest networks, which have all less than 5000 nodes, we have also computed their index using a dense structure.

Again, our computer presented an **Intel(R) Xeon(R) CPU E5-2620** (2 CPUs, 12 physical cores with Intel hypertreading, for a total of 24 physical and logical cores) processor, and we performed our tests increasing the number of worker from 2 to 24.

Their results have been presented by Professor Rodriguez at the international **20th ILAS conference 2016**, during the Minisymposium on Matrix Methods in Network Analysis [5].

For all the computation involving sparse matrices, our parallel algorithm revealed to be *always faster* than its serial twin.

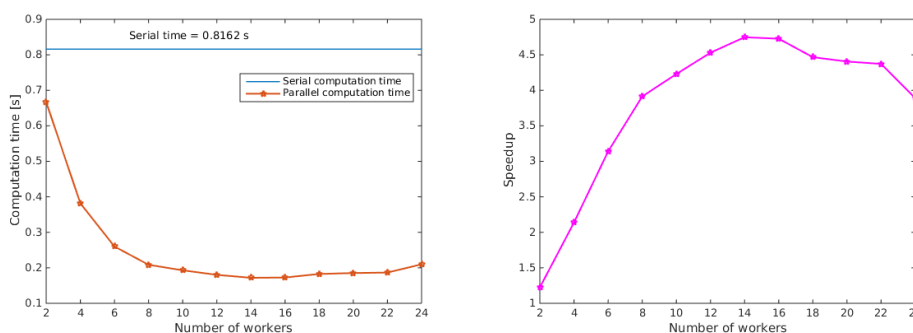


Figure 4.4: *Computation time and speedup for the Yeast network adjacency matrix (2114 nodes, sparse); k_{min} block-size is equal to 60*

For dense-structured matrices, this statement is not always true: in fact, in some cases there is a minimum number of workers that has to be involved

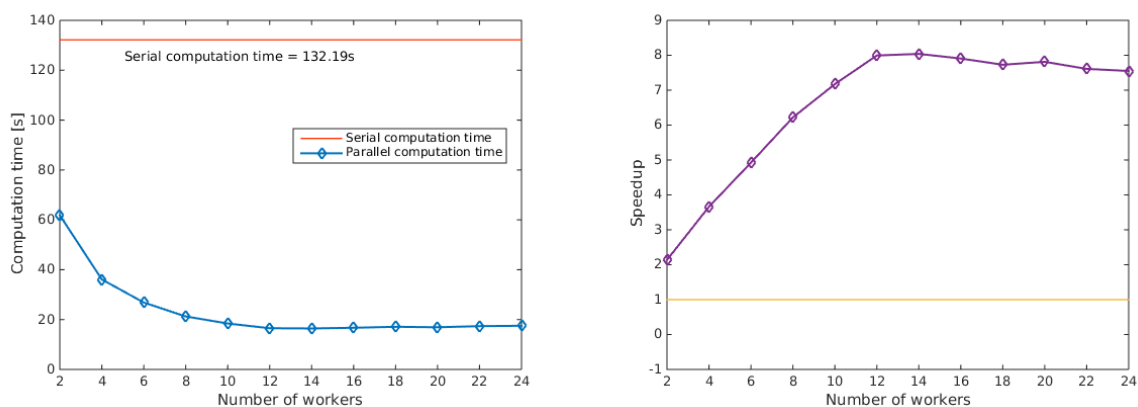


Figure 4.5: *Computation time and speedup for the Internet network adjacency matrix (22963 nodes, sparse); k_{min} block-size is equal to 8*

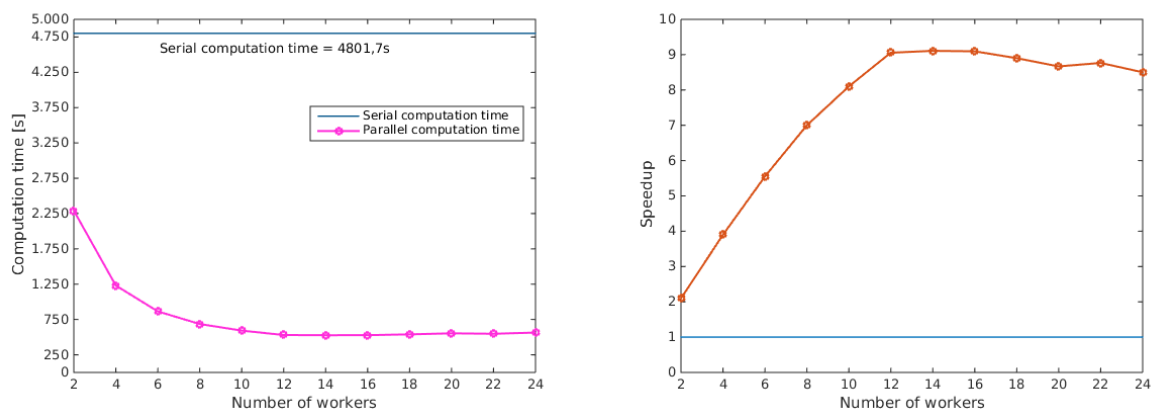


Figure 4.6: *Computation time and speedup for the Facebook network adjacency matrix (63731 nodes, sparse); k_{min} block-size is equal to 60*

into the computation to have some performance increase. In figure 4.7 for example, where we have reported the test results for the Estrada index computation of the matrix Power, the actual speedup of the execution time kicks in only when we use more than 4 workers inside the parpool.

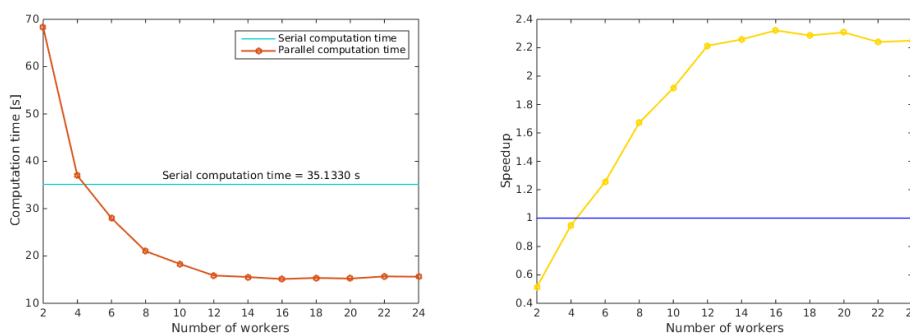


Figure 4.7: *Computation time and speedup for the Power network adjacency matrix (4941 nodes, dense); k_{min} block-size is equal to 40*

However, the average results are rather important, though the computations peaked very high values of speedup: more than 9, for example, for the Facebook adjacency matrix (see figure 4.6).

As inside the *parfor* construct we cannot use any type of Profiler like we did in chapter 3 with the *SPMD* syntax, we have less information about the use of computing resources by our algorithm.

What we have noticed, is that during the execution of the serial algorithm, MATLAB seems not to use all the available processors to carry out the computation; when executing the parallel algorithm instead, it exploits all the resources to finish its job.

From figure 4.8, the impression is that only two processors are working: the computational load switches between the cores leaving operating just two of them per time. Figure 4.9 shows instead that during the parallel computation, all the processors are intensively exploited: in fact, as figure 4.10³ illustrates, all the workers executed an even number of iterations, everyone of them being working for more than the 80 percent of the time. However, we did not investigate any further this aspect of our test, since, again, we do not have very much information about MATLAB's internal mechanisms.

³Figure 4.10 has been created through the use of the *parTicToc* timing utility by Sarah Wait Zanereck, which can be found at [30].

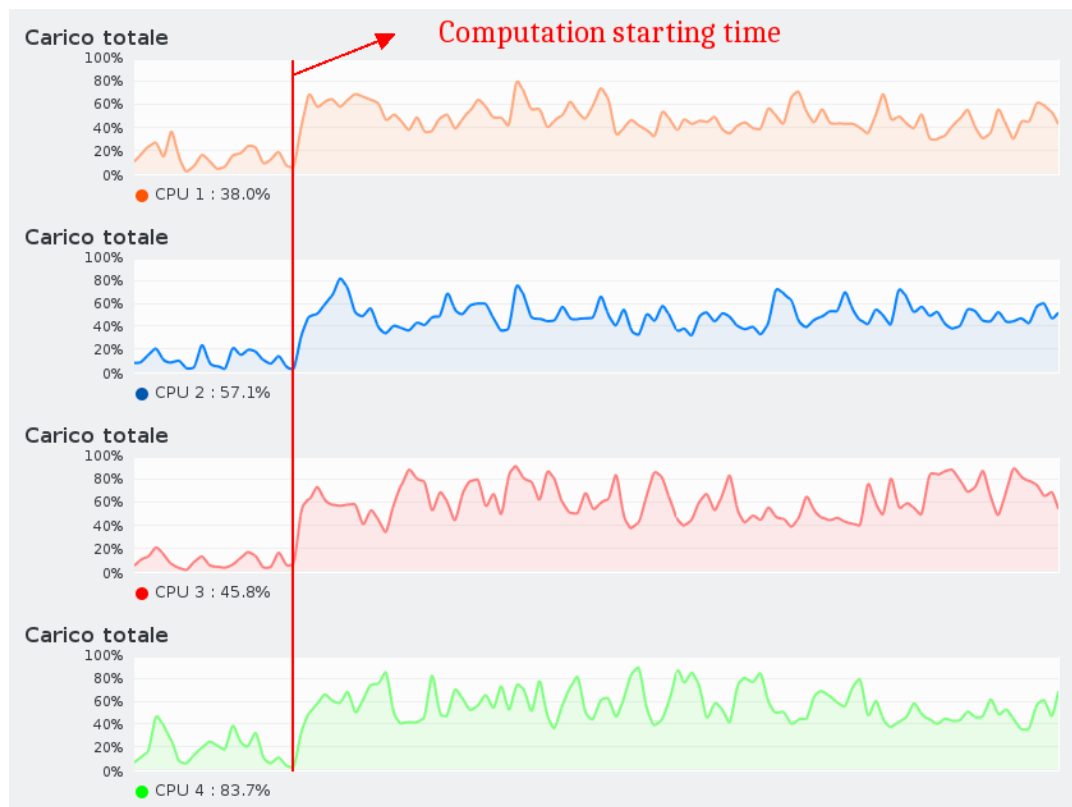


Figure 4.8: Processors' computational load during the *serial* computation of the Estrada index of the Internet matrix (22963 nodes, sparse). The computer is an Intel 5200U (2 physical cores with hyperthreading, for a total of 4 logical and physical processors)

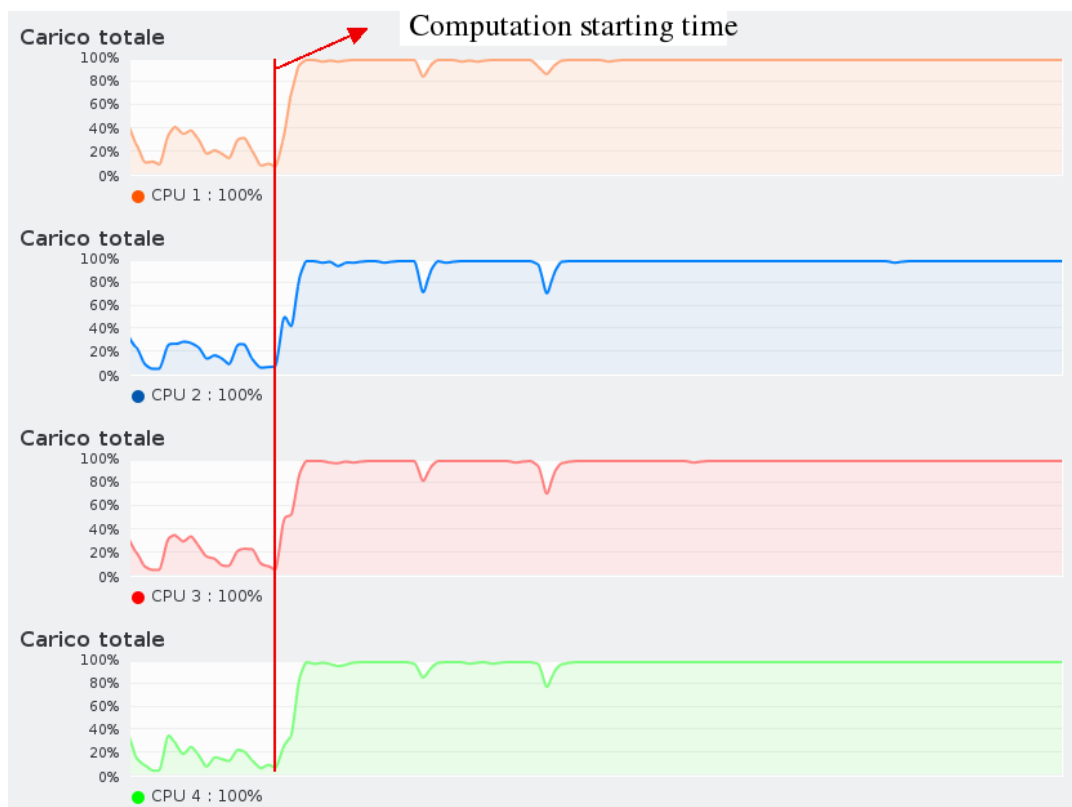


Figure 4.9: Processors' computational load during the **parallel** computation of the Estrada index of the Internet matrix (22963 nodes, sparse). The computer is an Intel 5200U (2 physical cores with hyperthreading, for a total of 4 logical and physical processors)

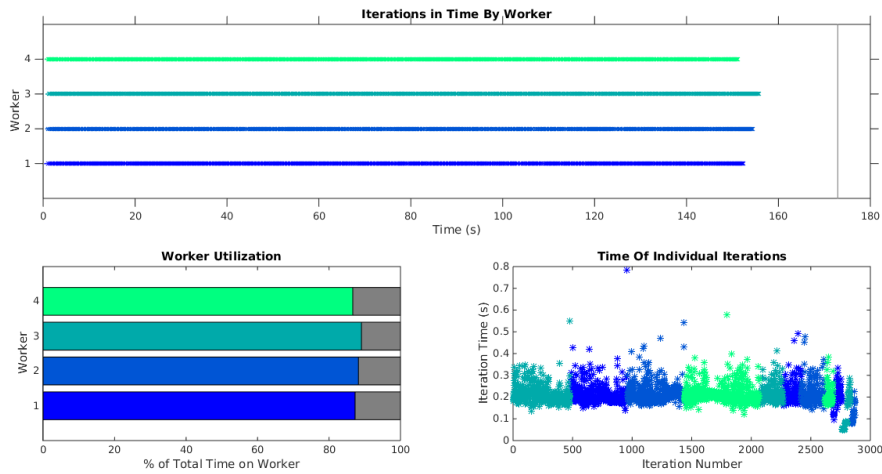


Figure 4.10: *Some information about the worker’s executional load during the parallel computation of the Estrada Index for the Internet network (22963 nodes, sparse). The machine is an Intel 5200U (2 cores with hyperthreading, for a total of 4 processors)*

Another remarkable observation is that while for the first tests the increase in speedup is almost linear with increasing number of workers, in general all the computations after reaching a maximum did not show any other improvement in their execution time.

We think that this feature may be related to the singular structures of the matrices, that may be particularly suited for certain data distributions related to a determined number of workers: thus, in this context, like for *jacobi_par* increasing the number of workers in the pool may not always lead to better performances.

It is peculiar though, that all tests show a leveling in the performance of the algorithm for parpools of more of **12** workers: it would seem that MATLAB could not really take advantage of the use of logical cores as it does for the physical ones.

Chapter 5

Conclusion

Our first goal when we started working on this bachelor thesis was to understand and explore the advantages, drawbacks and in general the overall implications in the parallel implementations of linear algebra algorithms.

We soon realized that our intent actually involved the knowledge and understanding of several scientific fields, especially computer science and engineering. Even when using an high level toolbox as MATLAB's PCT, the awareness of its implementation details played a major role in evaluating the behaviour and performances of our software.

From the use of lowly-tuned math operations with codistributed arrays, to an intensive use of processors with the parfor construct, to the mechanism of implicit multithreading, in many cases our predictions based on just the theoretical formulation of our algorithms had a surprising hidden side, leading us to two main remarks.

On one hand, the use of multiple computing resources is becoming the next step in the evolution of both software programming and hardware architecture design, requiring engineers and software developers to master the concepts about parallel computing to really exploit the possibilities in terms of performances that nowadays computers have to offer at different software levels. This concerns not only the development of new softwares that are able to take advantage of the parallel hardware characteristics, but even the reformulation of the existent algorithms in such a way that keeps them competitive in a parallel environment.

On the other hand, it is quite difficult to ask average computer users, even with good programming skills and crafts, but who may not be expert computer scientists, to struggle with all the aspects related to the realization and, particularly, the optimization of parallel softwares. It is quite likely that software and hardware companies will put great effort in realizing high-level programs that solve this duties without any stress for the user: the

Parallel Computing Toolbox and the Intel Math Kernel Library are perfect examples of that.

Anyhow, for what concerns our applications and tests, we also realized that parallel programming requires an intimate knowledge of both the problem and the instruments through which the software is implemented. Many factors from the instruction set employed, to the paradigm implemented by the programming language, to the architecture of the machine, can play a determinant role in the whole performance of a program as parallelism can take place at many levels.

Besides the negative (such as those of the parallelization of the Jacobi method) and positive (such as those of the computation of the Estrada index) results of our research, in a field like linear algebra a deeper understanding of the internal functioning mechanisms of MATLAB would have brought more validity to our work.

However, it leaves open new perspectives for further researches in this topic, such as the use of explicit multithreading for parallel computations [1], use of different parallel programming models with various memory structures and access, use of other data distributions to look how they affect the execution performances using both the **SPMD** and **parfor** construct.

Bibliography

- [1] Y.M. Altman. Explicit multithreading in matlab part 1, <http://undocumentedmatlab.com/blog/explicit-multi-threading-in-matlab-part1>.
- [2] Y.M. Altman. *Accelerating MATLAB Performance: 1001 tips to speed up MATLAB programs*. CRC Press, 2014.
- [3] Blaise Barney. Classification of parallel computers, https://computing.llnl.gov/tutorials/parallel_comp/parallelClassifications.pdf.
- [4] Blaise Barney. Introduction to parallel computing, https://computing.llnl.gov/tutorials/parallel_comp/.
- [5] M. Bellalij, E. Cannas, A. Concas, C. Fenu, D. Martin, R. Reichel, G. Rodriguez, H. Sadok, and T. Tang. Efficient computation of complex networks metrics by block Gauss quadrature rules. 20th ILAS Conference 2016, Minisymposium on *Matrix Methods in Network Analysis*, KU Leuven, Belgium, July 11–15, 2016, 2016.
- [6] M. Bellalij, L. Reichel, G. Rodriguez, and H. Sadok. Bounding matrix functionals via partial global block Lanczos decomposition. *Applied Numerical Mathematics*, 94:127 – 139, 2015.
- [7] Mike Croucher. Which matlab functions are multicore aware?, <http://www.walkingrandomly.com/?p=1894>.
- [8] Vassilios V. Dimakopoulos. *Parallel Programming Models*. Springer New York, New York, NY, 2014.
- [9] J.J. Dongarra and D.W. Walker. *The design of linear algebra libraries for high performance computers*. Aug 1993.

- [10] Anna Concas Edoardo Cannas. Risoluzione di sistemi lineari tramite il metodo global lanczos, <http://bugs.unica.it/~gppe/did/ca/tesine/15cc.pdf>.
- [11] L. Elbouyahyaoui, A. Messaoudi, and H. Sadok. Algebraic properties of the block GMRES and block Arnoldi methods. *Electronic Transation on Numerical Analysis*, 33:207–220, 2009.
- [12] Edric M Ellis. Matlab news group, http://it.mathworks.com/matlabcentral/newsreader/view_thread/265302.
- [13] G.H. Golub and G. Meurant. *Matrices, Moments and Quadrature with Applications*. Princeton Series in Applied Mathematics. Princeton University Press, 2009.
- [14] UPCRC Illinois. Parallel computing research at illinois the upcrc agenda, http://rsim.cs.illinois.edu/Pubs/UPCRC_Whitepaper.pdf.
- [15] Hari Kalva, Aleksandar Colic, Adriana Garcia, and Borko Furht. Parallel programming for multimedia applications. *Multimedia Tools and Applications*, 51(2):801–818, 2011.
- [16] Cleve Moler. The intel hypercube part 1, <http://blogs.mathworks.com/cleve/2013/10/28/the-intel-hypercube-part-1/>.
- [17] Cleve Moler. Matlab incorporates lapack, <http://it.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html>.
- [18] Cleve Moler. Parallel matlab: Multiple processors and multiple cores, <http://it.mathworks.com/company/newsletters/articles/parallel-matlab-multiple-processors-and-multiple-cores.html>.
- [19] Cleve Moler. Why there isn't a parallel matlab, http://www.mathworks.com/tagteam/72903_92027v00Cleve_Why_No_Parallel_MATLAB_Spr_1995.pdf.
- [20] D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.
- [21] G. Rodriguez. *Algoritmi Numerici*. Pitagora Editrice Bologna, Bologna, 2008.

- [22] G. Rodriguez and S. Seatzu. *Introduzione alla matematica applicata e computazionale*. Pitagora, 2010.
- [23] Gaurav Sharma and Jos Martin. Matlab®: A language for parallel computing. *International Journal of Parallel Programming*, 37(1):3–36, 2009.
- [24] Donald J. Becker Thomas Sterling. Beowulf: A parallel workstation for scientific computation , <http://www.phy.duke.edu/~rgb/bragma/resources/beowulf/papers/icpp95/icpp95.html>.
- [25] A.S. Tanenbaum. *Architettura dei calcolatori. Un approccio strutturale*. Pearson, 2006.
- [26] Intel Developer Team. Intel® architecture instruction set extensions programming reference, <https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf>.
- [27] Intel Support Team. Intel® hyper threading technology, <http://www.intel.it/content/www/it/it/architecture-and-technology/hyper-threading/hyper-threading-technology.html>.
- [28] Intel Support Team. Parallelism in the intel® math kernel library, https://software.intel.com/sites/default/files/m/d/4/1/d/8/4-2-ProgTools_-_Parallelism_in_the_Intel_C2_AE_Math_Kernel_Library.pdf.
- [29] MathWorks Support Team. Which matlab functions benefit from multithreaded computation?, <http://it.mathworks.com/matlabcentral/answers/95958-which-matlab-functions-benefit-from-multithreaded-computation>.
- [30] Sarah Wait Zaraneck. partictoc, <https://it.mathworks.com/matlabcentral/fileexchange/27472-partictoc>.