Università
Ca'Foscari
Venezia

Master's Degree

in Economics and Finance

Master's Degree in Economics and Finance
Erasmus Mundus Joint Master's Degree in
Models and Methods of Quantitative Economics

Final Thesis

Improving computer's performance with parallel computing

an application to the Metropolis-Hastings algorithm using MATLAB and R

Supervisor
Ch.ma Prof.ssa Antonella Basso

Assistant supervisor
Ch. Prof. Paolo Pellizzari

Graduand
Fabio Grattoni
857086

Academic Year
2019 / 2020

Contents

Introduction **5**

1 Parallel Computing **7**

1.1 Elements of parallel computing 7

1.1.1 Useful concepts in parallel computing 9

1.1.2 Conditions for parallel computing 10

1.1.3 Performance metrics for parallel programming 11

Speedup 11

Efficiency 12

Amdahl's law 12

Scalability and Gustafson's law 13

1.2 Parallel computing in MATLAB® 16

1.2.1 Parallel pools and MATLAB workers 17

Find the number of cores of your machine 18

Modify the parallel pool profile 18

1.2.2 The `parfor` function 19

Exception to the independence of `parfor` loops 20

1.2.3 The `spmd` function: 20

1.2.4 The `parfeval` function 22

Execute a function asynchronously and fetch output 23
Plot the result of a function during a computation: 24
When to use the `parfeval` over the `parfor`? 26
1.3 Parallel computing in R 26
1.3.1 The `parallel` library 26
Find the number of cores of your machine 27
Initialization of the parallel pool 28
Parallel versions of `apply` functions 28
1.3.2 `foreach` and the `doParallel` libraries 28
Registering the `doParallel` parallel backend 29
Difference between `foreach` and `for` loops 30
The use of the `.combine` option 30
Loading libraries on the parallel workers with the `.library` option 31
Getting information about the parallel backend 31
Stopping the cluster 31
1.4 Access to the VERA centre's virtual machines 32
1.4.1 Access from a Windows machine 32
Access to the Windows servers 32
Access to the Linux servers 34
1.5 Benchmark problems and scale-up study 36
1.5.1 Problems' specification and code 36
Monte Carlo method: simulation of π 36
Inversion of randomly generated matrices 40
Maximum eigenvalue of random matrices 42
1.5.2 Scale-up study and summary of the results 44

2 Simulation 48

2.1 Simulation models 48

2.2 Mathematical methods to generate stochastic sequences 50

2.2.1 Tests for statistical randomness 51

2.2.2 Random number generation: the Linear Congruential Generator 52

2.2.3 Random variable generation: the Inverse-Transform Method . . . 54

2.3 Variance reduction techniques 57

3 Introduction to Metropolis-Hastings algorithm 60

3.1 Motivations of the method 60

3.2 The Metropolis-Hastings algorithm 62

3.3 Diagnostics and calibration of practical implementation 63

4 Application of the M.-H. algorithm in parallel 69

4.1 Problem applied in the parallel analysis 69

4.1.1 A note on the parallelisation of the algorithm 70

4.2 Calibration of the parallel algorithm 70

4.2.1 Parameters used in the analysis 71

4.2.2 Improving the results with different starting points: the pathological case 72

4.3 Code 73

4.3.1 MATLAB code for the Metropolis-Hastings algorithm 74

4.3.2 R code for the Metropolis-Hastings algorithm 76

4.4 Results 77

Conclusion 80

Bibliography 82

Introduction

History of computers has been defined by an exponential growth in computing power potential. So much so that for many years researchers, companies, programmers and normal users who needed to perform computational demanding tasks could just wait few years and the technological improvements would grant an improvement in computer's potential that was, in fact, exponential. To our days, this increment has seen a slowdown while the problems' dimension have kept rising at a, possibly, even faster rate due to modern computing necessities, newer applications and the availability of growing quantities of data opening for a vastness of possibilities. Thus, nowadays it is needed to find alternative, more efficient, ways of getting an increase in computing power within the boundaries of modern computers. Here comes into play the topic that this thesis explores: *parallel computing*.

Parallel computing is a programming paradigm that offers the possibility to exploit unused computing power. Often, in modern computers, modern architecture often provides a large number of different processors which are not always fully exploited. All of the processors available can potentially be used at the same time, on the same task, in order to reduce computing time and obtain results more effectively, and this is the new paradigm that parallel computing offers. In the first Chapter this alternative paradigm is analysed, firstly from a theoretical standpoint and, secondly, with a more practical approach.

Before working on the actual parallelisation of a problem, it is needed to take some preliminary steps, since some conditions and requirements must be met. The problem should be computationally intensive to make parallelisation relevant and be comprised of tasks that are independent from one another. These conditions are explored and discussed further in Section 1.1.2.

An important aspect upon which this thesis dwells on is the trade-off that parallel computing involves. Using more than one core at the same time can, in fact, determine a performance improvement, however, uncovers also some difficulties in terms of how the problem geta split among the available cores, how these are instructed and how they need to be synchronised. On this aspect, some metrics used to determine the usefulness and efficacy of parallel computing are discussed in Section 1.1.3, helping in the analysis of whether parallelisation is worth the cost that it involves.

In Sections 1.2 and 1.3 the practical implementation of parallel computing in MATLAB and R respectively is presented. Furthermore, the thesis presents how to access the VERA machines in Section 1.4, with which the experiments have been performed. This thesis is, in fact, born thanks to the collaboration with the VERA centre (Venice centre in Economic and Risk Analytics for public policies) as a development of the work started as the object of an extracurricular stage conducted in collaboration with professors A. Basso and R. Casarin, members of the VERA Centre. In particular, access to some powerful computers of the centre was granted for the analysis lead in this thesis.

The last section of the first chapter, Section 1.5, is devolved to the presentation of some problems, used as benchmark for an applied analysis of parallel computing. The problems are described and the code is displayed. Then, a scale-up study is performed, applying the same problems with a varying number of cores in parallel in order to observe quantitatively how the parallel application can be beneficial, delivering the same results in a fraction of the time.

Chapter 2 focuses on computer simulation, a method to numerically solve problems otherwise too complex or even impossible. Firstly, the general definition and different elements of models, together with the the evaluation of its validity and the general limitations are explored. Secondly, in Section 2.2, the discussion moves on to some of the mathematical methods to generate pseudorandom numbers and stochastic sequences in a deterministic way with the computer. Finally, some techniques of variance reduction are presented. These consist of techniques used to increase the precision of the estimators that the model generates, making the simulation process more effective.

Chapter 3 exposes the Metropolis-Hastings algorithm, a Markov chain Monte Carlo method used to sample a sequence of random numbers from a target distribution that is difficult or impossible to compute. Section 3.1 presents the motivation of the method while Section 3.2 describes the algorithm itself. Section 3.3, then, analyses all calibration process that needs to be done when practically applying the algorithm.

In light of these considerations, the last chapter consists of a practical, parallel, application of the Metropolis-Hastings algorithm. A new problem (made of specific target and proposal distributions) is taken into consideration for the experiment. The problem is described in Section 4.1.1 and then its calibration is discussed in Section 4.2.2. Then the specific code used is presented and, finally, in Section 4.4, the results of the scale-up experiment similar to the one performed in Section 1.5 are showcased and commented.

Chapter 1

Parallel Computing

1.1 Elements of parallel computing

Von Neumann's architecture is a computer architecture based on the work of the great Hungarian-American mathematician and physicist John Von Neumann in 1945. During the years the term has evolved to mean any computer that cannot execute at the same time an *instruction fetch* and a *data operation*, defined as movement of data and instructions essential for the computation process: instructions that need to be sent to the processing units and results that need to be taken from the them. The key component in the architecture design, responsible for this limitation, is the BUS: the communication system employed to connect all of the computer's components to the motherboard. It is a pivotal resource and the fact that it is shared between program memory and data memory determines the limitation that has been defined with the name of *Von Neumann's bottleneck* (Backus 1978). There is a limited throughput, data transfer rate, between the CPU and the memory. The single BUS present in the design can only access to one of the two classes of memory at a time, determining a loss of potential computations that the CPU could do. The CPU is in fact continually forced to wait for the needed data to move to or from memory. This architecture, and the consequent limitation described, determined what has been called the serial computing paradigm, meaning that the computer is able to perform only one action at a time. A solution to this limitation can be found in the employment of parallel computing models. Such models develop on the paradigm for which many calculations can be carried out simultaneously by employing more processing units (CPUs). Large problems can often be divided into smaller series of computations. These series can be further divided, distributed and, therefore, executed on different computational resources. These different computational resources will therefore be working at the same time on different parts of the same problem. In this way more resources present in most computers, or clusters of computers, can be exploited simultaneously, with results obtained in a more efficient way. Parallel computing, in fact, can take advantage and keep working on multiple cores at the same time.

For many decades Moore's Law bestowed a wealth of transistors that hardware designers transformed into usable performance (Adve et al. 2008). Such law is the observation that the number of transistors in a dense integrated circuit used to double about every two years. It has been a trend that held true up until around 2010 and it has been useful for long-term planning and to set targets for the industry of semiconductors. But things need to evolve as this trend is not valid anymore (Catalant Staff 2015). The sequential programming interface, for the most part, however, did not change as, in order to have an increase in computing power, it was sufficient to wait few years for hardware improvements. But, in recent years, as the power limit as been approached, sequential programming could not determine a sufficient improvement in performance anymore. "The computer industry has accepted that future performance increases must largely come from increasing the number of processors (or cores) on a die, rather than making a single core go faster"(Adve et al. 2008). Parallel computing can help tackle more and more challenging problems, offering a way to take advantage of multi-core processors that nowadays are very common also on personal computers. This procedure is even more rewarding when working on big computer clusters where there is more potential to be exploited, since parallel computing became the dominant paradigm in computer architecture (Asanović et al. 2006), solving at the same time also the concern related to increase in power consumption (and consequently heat generation). Parallelism is, in fact, an energy-efficient way to easily achieve increase in performance (Anantha, Samuel, and Robert 1992).

Traditionally computer software has been written for serial computations. Any task is represented by an algorithm: a series of instructions that the computer executes one after the other. After a task is completed the computer can move to the next one and so on. All of the instructions are executed on the same central processing unit, which can only do one computation at a time. The frequency scaling approach has been the main form of performance improvement since the beginning of computers. It focuses on the performance improvement of the common central processing unit as to enhance the performance of the system containing the processor in question. Parallelism, on the other hand, uses multiple processing units simultaneously to solve a problem. It achieves this by separating a problem into many subtasks one independent from the other. The results are combined afterwards, after completion. Of course not every task can be parallelised. The simultaneous execution implies the application on an algorithm that is not affected by the order of completion of each task. These aspects are analysed in further detail in Section 1.1.2.

1.1.1 Useful concepts in parallel computing

In order to provide an introduction to parallel computing clearly, some useful concepts that will be used throughout the paper are needed. These are both technical concepts, fundamental in order to understand parallel computing, and definitions that will define a common nomenclature useful to avoid confusion:

- **CPU**: acronym for Central Processing Unit. Sometimes also the term processor is used, even though it is a more general term. The CPU handles basic instructions and allocates the more complicated tasks to other specific chips to get them to do what they do best.

- **Core**: it is a term that will be used many times due to its key importance for the concept of parallel computing. Modern days CPUs have multiple cores. These cores are the actual central processing units. A CPU with two cores (dual core) can actually do two different computations at the same time. This is the corner-stone of parallel computing, the main feature leading to the creation of such paradigm.

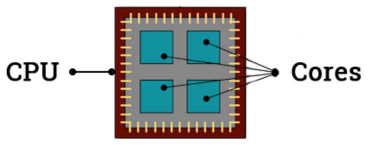
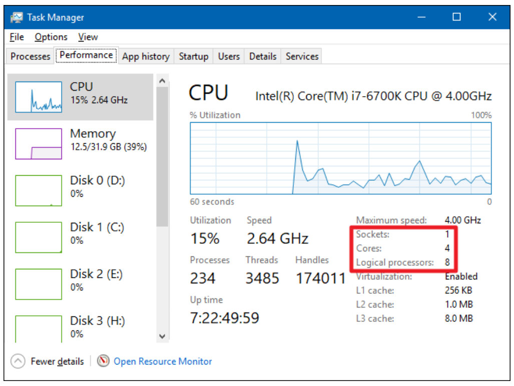


Figure 1.1: Quad-core CPU representation

- **Multi-threading**: also called hyper threading, it is a technology which creates virtual processor cores. They aren't as powerful as physical cores but they may help improve performance. With multi-threading a single core can make two computations at a time.



CPU Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz

Utilization 15%
Speed 2.64 GHz
Processes 234
Threads 3485
Handles 174011
Up time 7:22:49:59

Maximum speed:	4.00 GHz
Sockets:	1
Cores:	4
Logical processors:	8
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	8.0 MB

Figure 1.2: Windows' Task Manager shows the multi-threading property very well as the number of logical processors compared to the number of cores.

- **Thread**: smallest sequence of programmed instructions that can be managed independently. "Thread" is generally accepted as generic term for subtasks.

- **Cluster**: group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. The single machines composing the cluster are connected by a network. Clusters by design are ideal to be used for parallel computing due to their high number of cores compared to personal computers. There is more potential to be exploited by parallelising.

1.1.2 Conditions for parallel computing

The idea of parallelism, in its essence, is fairly simple but its implications are powerful. Yet, the many conditions that a program must fulfil in order to be parallelisable make it a nearly impossible activity to automatize.

The simplest, trivial, situation in which parallelisation can be applied is the one in which the program is composed of many tasks, everyone perfectly independent from one another. In this case we call such programs *embarrassingly parallel.* These are the cases in which little to no manipulation is needed in order to make them parallel so these are the cases in which there may be more to gain. There is little human cost involved. Furthermore, complete independence among tasks means also less communication or synchronisation among different cores, which makes the parallelisation more efficient, as we will see exploring the less trivial cases of parallelisation.

Problems that present some more difficulties, which need some communication among different tasks, and therefore cores, even though not many times per second, are problems which are said to present coarse-grained parallelism. If they need to communicate many times per second they present fine-grained parallelism. If the program needs a level of synchronisation, it will need some form of barriers, that usually take the form of a lock or a semaphore. These are techniques, programming language constructs, that help control the access to some specific data or resource.

In addition to these considerations, one element to take into account is whether parallelisation is profitable in terms of speed gains. Not always, in fact, it is beneficial to parallelise. It may be the case that the program is not suitable for parallelisation, or it may be the case that after parallelisation the program may require some extra steps that may be difficult to program, or they may lead to loss in efficiency due to the needed interaction among the different cores. A new level of analysis is needed and in order to perform it, it is necessary to understand all possible factors that may play a role in the process of parallelisation. Considerations, about how the number of cores will affect improvements or the evaluation of the trade-off between the performance improvement and the human cost involved in transforming the program into a parallel one, must be done. In order to perform this analysis it is necessary to have some metrics, tools that help measure performance.

1.1.3 Performance metrics for parallel programming

The main criterion for the performance evaluation of a parallel program is its running time $T_p(n)$. It is the time elapsed between the beginning of the computation and the end of it on every processor involved (Rauber and Rünger 2013). It is usually defined, with a specification p of the number of processors, as a function of the size n of the problem. The problem size is given by the size of the data input, for instance the number of different simulations in a Monte Carlo problem. This is a very generic measure that incorporates different chunks of time devoted to various things: the runtime for the *execution of local computations* of each participating processor, the runtime for the *exchange of data* between processors, the runtime devoted to eventual *synchronisation* of data access, and *waiting times* due to possible unequal load of distribution of processes or mutual exclusive access to some data. The time spent for exchanging data, synchronization and waiting can be considered as overhead since it does not contribute directly to the computations to be performed. These are complementary activities, needed in order to perform the computations.

Speedup

For the evaluation of parallel programs it is useful to compare the parallel version to a serial (or purely sequential) one. We define, therefore, the speedup $S_p(n)$ of a parallel program with parallel execution $T_p(n)$ as:

$$\text{Speedup: } S_p(n) = \frac{T^*(n)}{T_p(n)} \tag{1.1}$$

where p is the number of processors involved in the parallelisation, and $T^*(n)$ is the serial execution time to solve the same problem with the best sequential algorithm.

Theoretically, $S_p(n) \leq p$ always holds. If we observe a situation in which $S_p(n) > p$, it means that the sequential algorithm used was not optimal, and could easily be improved deriving it from the parallel one. The new sequential algorithm could be constructed by a round-robin simulation of what would be executed in the participating p processors. So the first p steps of the new sequential algorithm would be the first step of each of the p processors, the second p steps the second step of all of the p processors and so on. Thus, the new sequential algorithm would perform p times more steps than each one of the parallel one. Therefore, we have that the new sequential algorithm would have execution time $p \cdot T_p(n) \overset{(1.1)}{\Leftrightarrow} p \cdot \frac{T^*(n)}{S_p(n)} < T^*(n)$ given that $S_p(n) > p$. This result is a contradiction of the assumption that the best sequential algorithm has been used to compare the execution times. Nevertheless, since the best sequential algorithm may be difficult to determine, or may be simply unknown or impossible to prove that is the best possible one, the speedup is computed normally using a sequential version of the parallel implementation (Rauber and Rünger 2013).

The typical situation is that a parallel implementation does not reach linear speedup ($S_p(n) = p$), which can be seen as the optimal result. The decrease in speedup may be caused by the limit in scalability of the parallel program or by the overhead cost for the management of parallelism. As we have previously seen, this overhead can be caused by the necessity to exchange data between different processors, by synchronization or by waiting times caused by an unequal load balancing between the processors.

Efficiency

An alternative measure of performance of a parallel program is its efficiency. This measure captures the fraction of time for which a processor is usefully employed by computations that would have to be performed also by a sequential program (Rauber and Rünger 2013). It is computed as:

$$\text{Efficiency} = E_p(n) = \frac{T^*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)}$$

where $T^*(n)$ is the sequential execution time of the best sequential program, $T_p(n)$ is the parallel execution time on the parallel algorithm using p processors and $C_p(n)$ is the cost, a measure of the total amount of work performed by all processors, and is defined as $C_p(n) = p \cdot T_p(n)$. The ideal speedup $S_p(n) = p$ corresponds to an efficiency of $E_p(n) = 1$.

Amdahl's law

Ideally the speedup from parallelisation would be linear. Two cores would do a computation twice as fast compared to a single core, three cores three times as fast and so on. Of course the number of cores employed constitutes an upper bound to the speedup. In reality we do not observe a linear increase. The parallel execution time cannot be arbitrarily reduced by employing parallel resources. The reason lies in the fact that there are some elements of the program that cannot be parallelised. Most algorithms have a near-linear speedup for a small number of processing elements. The marginal return of adding one more processing unit is, nevertheless, decreasing. This improvement flattens out into a constant value for large number of processing elements. The potential speedup of an algorithm on a parallel computing platform is quantitatively given by Amdahl's law (Amdahl 1967). In a program that is composed by a (constant) fraction f, $f \in [0, 1]$, that must be executed sequentially, then the sequential execution time will be $f \cdot T^*(n)$. The fraction $(1 - f) \cdot T^*(n)$ is the share of the program that is parallelisable which, once parallelised with p processors, has an execution time of $\frac{1-f}{p} \cdot T^*(n)$. Therefore, the total execution time of the parallel program will be $T_p(n) = f \cdot T^*(n) + \frac{1-f}{p} \cdot T^*(n)$. From this, and the definition of speedup from equation (1.1), it easy to see that the attainable speedup $S_p(n)$ is therefore:

$$\begin{aligned} S_p(n) &= \frac{T^*(n)}{T_p(n)} \\ &= \frac{T^*(n)}{f \cdot T^*(n) + \frac{1-f}{p} \cdot T^*(n)} \\ &= \frac{1}{f + \frac{1-f}{p}} \\ &\leq \frac{1}{f} = \lim_{p \to \infty} S_p(n) \end{aligned}$$

This estimation assumes that, as when we defined the speedup, the best possible sequential algorithm is used and that the parallel fraction is perfectly parallelisable. The consequence of this law lies exactly in the upper bound that theorizes and how it is directly connected with the sequential share of the program. To understand the consequences an example is useful. Suppose 20% of a program must be run sequentially, we have $f = 0.2$. The attainable speedup, the upper bound, is given by $1/f = 5$. No matter how many processors are involved in the parallel execution of the program, the speedup will never be greater than 5.

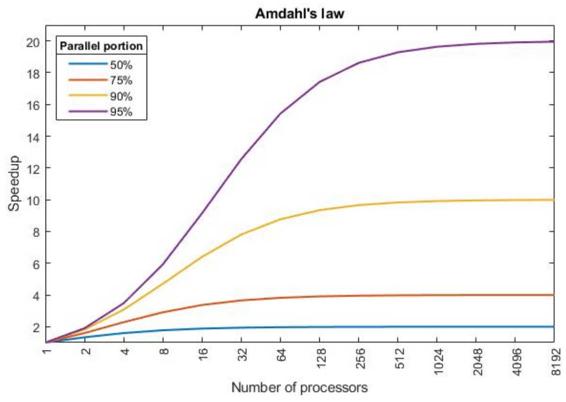


Figure 1.3: Graphical representation of Amdahl's law. It is possible to see how the speedup of a program is limited by how much the program can be parallelised.

Scalability and Gustafson's law

Scalability is a measure describing whether a performance improvement can be reached that is proportional to the number of processors employed (Rauber and Rünger 2013). As we have seen, scalability depends on the structure of the algorithm and its parallel execution. Amdahl's law shows that for a fixed problem size n a saturation level of the speedup can be observed as the number of processors p is increased. The upper bound observed before. This may be seen as a rather pessimistic view of potential speedup. On the other hand, we generally observe that as more computation resources become available they tend to get used on larger problems; with larger datasets, greater number of simulations and so on. With the problems becoming larger, the parallel side of the program usually increases much faster than the serial part, determining a decrease of the serial fraction of the program overall, resulting in a linear increase in the speedup (McCool, Reinders, and Robison 2012). Therefore, if this is the case of the problem considered, Gustafson's law (Gustafson 1988) gives a more realistic, and less pessimistic, assessment of the parallel performance, with the dimension of the program varying linearly with the number of processors. In fact, users have usually access on specific parameters that can be adjusted to allow the program to run in some desired amount of time. Therefore, it may be assumed that run-time on the parallel application, and not problem size, is constant. If τ_f is the constant time of the sequential program and $\tau_v(n,p)$ is the execution time of the parallelisable part of the program for problem size n and p processors, then the speedup is expressed as:

$$S_p(n) = \frac{T^*(n)}{T_p(n)} = \frac{\tau_f + \tau_v(n,1)}{\tau_f + \tau_v(n,p)} \tag{1.2}$$

Given perfect parallelisation, we may rewrite:

$$\tau_v(n,1) = T^*(n) - \tau_f \tag{1.3}$$

$$\tau_v(n,p) = \frac{T^*(n) - \tau_f}{p} \tag{1.4}$$

It follows, by adding (1.3) and (1.4) in (1.2):

$$S_p(n) = \frac{\tau_f + T^*(n) - \tau_f}{\tau_f + \frac{T^*(n)-\tau_f}{p}} = \frac{\frac{\tau_f}{T^*(n)-\tau_f} + 1}{\frac{\tau_f}{T^*(n)-\tau_f} + \frac{1}{p}}, \tag{1.5}$$

and therefore, if $T^*(n)$ increases strongly monotonically in n, we have

$$\lim_{n\to\infty} S_p(n) = p.$$

A graphical illustration of this property is given in Figure 1.4. Bars represent the computational time. First a task performed serially, and then, the same one performed with two processors. In the second set of bars, the number p of processors is doubled and so is the parallel part $\tau(2n, 2p)$ of the problem. In this way the final parallel computation time is the same of the first case. But having doubled the size of the problem, to achieve this, the speedup is increased. The degree of the increase is proportional to the share of the initial problem that can be parallelised. We can show this behaviour by developing equation (1.2) in a different way. First we observe that $\tau(n,1) = \tau(n,p) \cdot p$. Then, without the loss of generality, we set $\tau_f + \tau_v(n,p) = T_p(n) = f \cdot T_p(n) + (1-f) \cdot T_p(n) =$

$1 = f + (1 - f)$. Therefore we have:

$$S_p(n) = \frac{\tau_f + \tau_v(n,p) \cdot p}{\tau_f + \tau_v(n,p)} = f + (1 - f) \cdot p.$$

We see from this formula that the increase of the speedup as a function of p, with the size n of the problem that increases proportionally, is represented as a line with slope $(1 - f)$ as the ones in figure 1.5, i.e. the share of parallel computation of the algorithm.

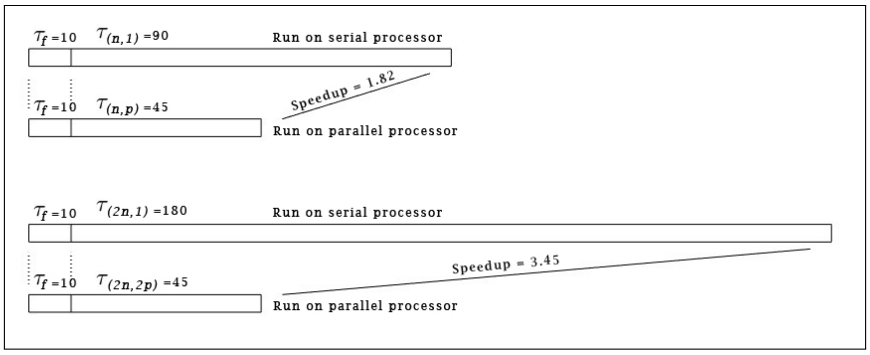


Figure 1.4: Example of parallelisation of problems with size increasing proportionally to the number of processors employed.

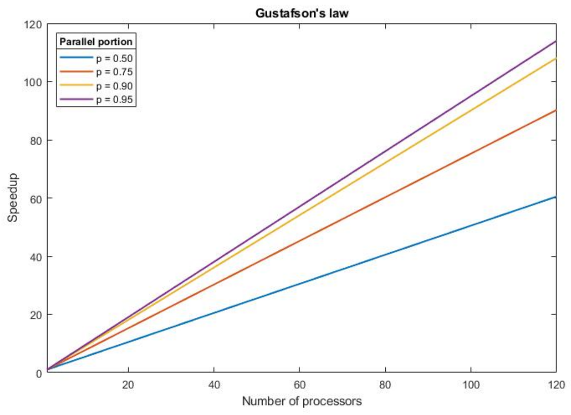


Figure 1.5: Graphical representation of the Gustafson's law. The speedup of a program is linear to the number of cores involved and increases with the parallelisable share.

A further element to be considered is the so called overhead cost. When a parallel computation is set, there is a computational cost that affects speed improvements. Said cost is needed in order to transfer the code to be executed on a different processor or transfer data between different cores, and so on. This is why parallelisation should be evaluated case by case and this is where some possible inefficiencies may lie, making the predictions given by the laws we have just saw very rare to observe, because of decreases in efficiency. In the case of computation-heavy parallel programs, most CPU-time will be spent doing the computations, so there may be a high improvement through parallelisation and the predictions may be quite accurate. For data-heavy programs in which computations performed are trivial, the overhead time to transfer the data to the many processors and then retrieve the results back may overweight the gain in computation time. Each situation is different, and each problem must be evaluated in order to determine whether parallelisation is the suitable path to take and which parallelisation approach should be preferred.

1.2 Parallel computing in MATLAB®

In the last few years MATLAB has implemented in its proprietary software a series of tools and functionalities that take advantage of parallel computing principles (MathWorks 2020). Some form of parallelisation is already implemented under the hood, with some functions that run in parallel, without any different procedure required by the user. This is done for instance in the matrix multiplication. The sintax is not changed but the performance is improved considerably. We can show this by running MATLAB from the terminal in the default way (`> matlab`) or with the `-singleCompThread` option (`> matlab -singleCompThread`). Then we can run the same piece of code in the two different instances of MATLAB and measure the time that the program takes to make the computation:

```
rng(42); %set the seed to get consistent results
a = rand(5000);
b = rand(5000);
tic; a*b; toc
```

Computation time with the MATLAB with no different option was 3.96 seconds, while with the -singleCompThread option computation time was 6.28 seconds. The same conclusion can be drawn by observing the CPU usage during computation. The normal MATLAB client gets to use double the amount of CPU compared to the single threaded one.

In addition to these changes, new possibilities for parallel computing have been implemented in the Parallel Computing Toolbox™, which is a toolbox with functions aimed at computation on multicore computers, GPU's and computer clusters. This toolbox with its functions and usage will be explored in the following sections, but first it will be necessary to install it. To do so it is sufficient to follow the path `Home\Add-Ons\Get Add-Ons` inside of MATLAB. Then it will be sufficient to search for the toolbox and click install.

1.2.1 Parallel pools and MATLAB workers

Two fundamental elements of MATLAB's Parallel Computing Toolbox^{TM} are the concepts of **parallel pool** and **workers**.

The **workers** are MATLAB processes that run in background without a graphical desktop. Functions in the Parallel Computing Toolbox can be used to divide the different tasks and assign them to different workers that can then perform the computations in parallel. Workers can be run locally or it is possible to scale up to run the workers on a cluster of machines. The MATLAB session you interact with is known as the MATLAB *client*.

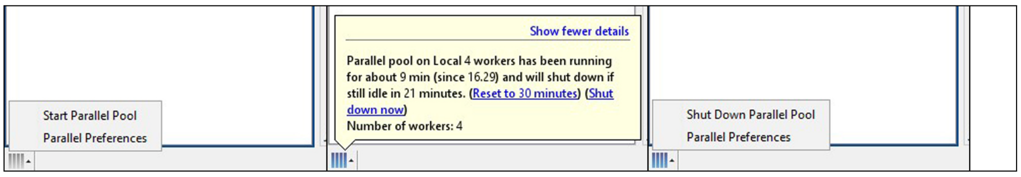
The **parallel pool** is the name given to the pools of workers. By default, parallel language functions will initialize a parallel pool when necessary. The default parallel pool assigns one worker per core available. This is because, although there could be more virtual/logical cores available, some resources are shared among these. One of the shared resources is the floating point unit (FPU) which is often used by MATLAB because it is a double-precision floating point. This is very important in the case of computationally intensive code. If this is not the case, for instance if it is input/output (IO) intensive, using more workers per core would be more efficient.

A parallel pool can be initialized with the command `parpool()` which will start it with the default settings. It is not always necessary to run this command since the parallel pool will be automatically started by the functions in the Toolbox that make use of parallel computing and, therefore, require a pool. The number of workers of the parallel pool will be the one defined in the pool's default settings. This may be customized with the command `parpool(poolsize)`. Also different parallel pool's profiles can be used by means of the command `parpool(profilename)`. How to set and modify parallel pool's profiles is detailed in Section 1.2.1.

A general structure to start a parallel pool is:

```matlab
numWorkers = 2; % number of desired workers
poolObj = parpool(numWorkers, 'profilename');

    % parallel computation goes here

delete(poolObj); % closes the parallel pool
```

In the bottom left corner of the MATLAB window there is the pool status indicator which indicates whether a pool is running or not, how long it has been idle and when it will automatically shut down, the number of workers and it lets you quickly start and stop a parallel pool in addition to have a quick access to the parallel preferences.



Parallel pool on Local 4 workers has been running for about 9 min (since 16.29) and will shut down if still idle in 21 minutes. (Reset to 30 minutes) (Shut down now)
Number of workers: 4

Figure 1.6: Different states of the parallel pool indicator

To get the current parallel pool object (in the case the object was not created when starting the pool) the command `gcp()` will address it. To avoid starting a pool by using it if it was not already active, is it possible to use `gcp('nocreate')`. A common way to close a parallel pool is with the command `delete(gcp('nocreate'))`.

Find the number of cores of your machine

```
feature('numcores')
```

```
MATLAB detected: 2 physical cores.
MATLAB detected: 4 logical cores.
MATLAB was assigned: 4 logical cores by the OS.
MATLAB is using: 2 logical cores.
MATLAB is not using all logical cores because hyper-threading is enabled.
ans =
     2
```

This function gives as a result the information above depicted for a dual-core computer. Furthermore, it gives a value as a result: the number of physical cores present in the machine, in this case 2.

In some cases it could be useful to get the number of **logical cores** programmatically as a value that can subsequently used in the script, and to do so it is sufficient to run the command `str2num(getenv('NUMBER_OF_PROCESSORS'))`. This command will return the number of logical cores, or processes, in our case 4.

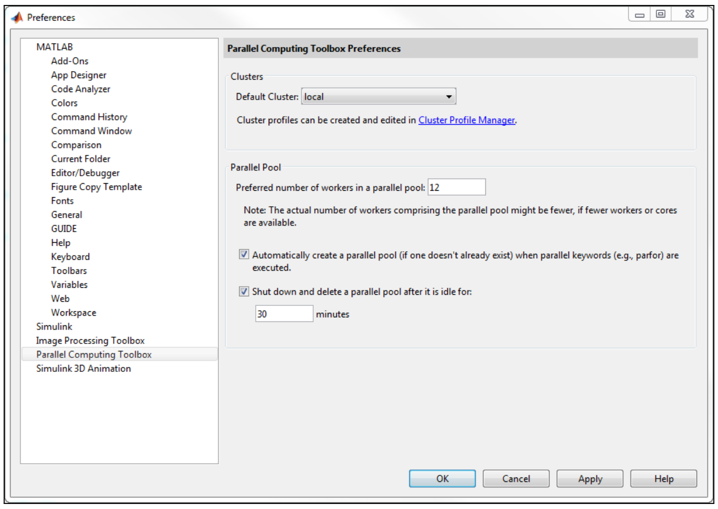
```
str2num(getenv('NUMBER_OF_PROCESSORS'))
```

```
ans =
     4
```

Modify the parallel pool profile

In order to modify the parallel pool settings it is necessary to reach the Parallel Computing Toolbox in the preferences. In order to do so type `preferences` in the MATLAB command line and then select `Parallel Computing Toolbox` in the Navigation tree on the left. In this preferences window it is possible to modify the following settings:

- **Default cluster:** choose the cluster to use. The default one is 'local'.
- **Preferred number of workers:** specify the number of workers in your parallel pool. The actual pool size may then be limited by licensing, cluster size, and cluster profile settings.
- **Automatically start a parallel pool:** selecting this option many different MATLAB functions will start automatically a parallel pool with the default values if necessary (if a pool does not yet exist). If a parallel pool is started automatically it is still possible to access it with the `gcp()` function.
- **Shut down and delete a parallel pool:** it is possible to set the idle time of the parallel pool before its automatic shutdown. The countdown starts when the pool stops to be used. If it gets used again the countdown resets.



Parallel Computing Toolbox Preferences

Clusters

Default Cluster: local

Cluster profiles can be created and edited in Cluster Profile Manager.

Parallel Pool

Preferred number of workers in a parallel pool: 12

Note: The actual number of workers comprising the parallel pool might be fewer, if fewer workers or cores are available.

☑ Automatically create a parallel pool (if one doesn't already exist) when parallel keywords (e.g., parfor) are executed.

☑ Shut down and delete a parallel pool after it is idle for:

30 minutes

Figure 1.7: Preferences window of Parallel Computing Toolbox

1.2.2 The `parfor` function

```
parfor loopvar = initval:endval; statements; end
parfor (loopvar = initval:endval, M); statements; end
```

The `parfor` is a widespread function due to its similarities with the usual `for` and, therefore, its ease of use. The mere modification from a `for` to a `parfor` lets the code run in parallel starting, therefore, a parallel pool (if it does not yet exist).

A necessary condition for the `parfor`-loop to work is for the statements that run inside the loop to be independent from one another. More precisely, all the code that follows the `parfor` statement should not depend on the loop iteration sequence since they are not executed in a guaranteed order.

The function will also verify that the statements inside the loop are independent, showing an error whenever this is not satisfied (for instance if a variable computed in a previous iteration is called and used for a new computation).

The `M` variable in the specification of the characteristics of the loop, lets you modify the maximum number of workers or threads used.

Exception to the independence of `parfor` loops

An exception to independence is given by the *reduction variables*. A reduction variable is a variable that accumulates a value that depends on all iterations altogether, but it is independent of the iteration order. A typical usage of reduction variables is the following in which the variable `X` accumulates the values of all of the `d(i)` variables:

```
X = ...;           % Do some initialization of X
parfor i = 1:n
    X = X + d(i);
end
```

This loop is equivalent to the following expression, where you calculate each `d(i)` by a different iteration, and for which the summation order does not matter:

```
X = X + d(1) + ...  + d(n)
```

1.2.3 The `spmd` function:

```
spmd
    statements
end
```

The *spmd*, acronym for *single program, multiple data*, is a technique employed to achieve parallelism. Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster.

The “single program” aspect of *spmd* means that the identical code runs on multiple workers. You run one program in the MATLAB client, and those parts of it labeled by *spmd* blocks run on the workers, simultaneously. Each worker can operate on a different dataset or different portions of distributed data, and can communicate with other participating workers while performing the parallel computations. When the *spmd* block is complete, your program continues running in the client. To execute it in parallel, you must first create a pool of MATLAB workers using `parpool` or have your parallel preferences to allow the automatic start of a pool.

Inside the body of the *spmd* statement, each MATLAB worker has a unique value of `labindex`, while `numlabs` denotes the total number of workers executing the block of code in parallel. Within the body of the `spmd` statement, communication functions for communicating jobs (such as `labSend` and `labReceive`) can transfer data between the workers.

Values returning from the body of a `spmd` statement are converted into a **composite** object on the MATLAB client. This is an object containing references to the values stored on the remote workers. Those values can be retrieved using cell-array indexing. The actual data remains available on the workers for subsequent `spmd` execution, so long as the composite object exists on the client and the parallel pool remains open. A basic example of how to use the `spmd` function and how to retrieve the values from the composite variable is presented below:

```matlab
poolObj = parpool(2);

spmd
   disp("Worker n." + labindex + "/" + numlabs + " is computing " +
       labindex + "^2.")
   result = labindex^2;
end

disp("The results retrieved from the composite variable are: ")
disp(result{1});
disp(result{2});

delete(poolObj);
```

The result of this code will be:

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
Lab 1:
  Worker n.1/2 is computing 1^2.
Lab 2:
  Worker n.2/2 is computing 2^2.

The results retrieved from the composite variable are:
  1
  4
Parallel pool using the 'local' profile is shutting down.
```

Another example of use of the `spmd` is using the `labindex` variable to open different datasets and apply the same function to them:

```
spmd (3)
   labdata = load(['datafile_' num2str(labindex) '.ascii'])
   result = MyFunction(labdata)
end
```

1.2.4 The `parfeval` function

The `parfeval` function is useful to evaluate functions in the background, in a parallel way, without waiting for them to be complete. Intermediate evaluation of the computation's result can be useful to create a plot showing the progress, or to stop an optimization procedure that reached a good enough state. This can be done on one or all of the workers, with `parfeval` or `parfevalOnAll`. Note the difference with the `parfor`, where you have to wait for the loop to be complete.

```
F = parfeval(p,fcn,numout,in1,in2,...)
F = parfeval(fcn,numout,in1,in2,...)
```

- `p`: parallel pool object;
- `fcn`: function to execute on a worker, specified as a function handle, for example `fcn = @myFunction`;
- `numout`: number of outputs expected from `fcn`;
- `in1, in2, ...`: input variables of the function `fcn`;

Furthermore, the output:

- `F`: future object, returned as a `parallel.FevalFuture`, that represents the execution of `fcn` on a parallel worker and holds its results. Use `fetchOutputs` or `fetchNext` to collect the results.

The function `wait(F)` can be used to make the execution of the program stop, waiting for the `parfeval` cycle to end. Furthermore, you can cancel the execution with `cancel`.

Execute a function asynchronously and fetch output

The `parfeval` function permits the execution of a function in the background, while leaving the MATLAB client able to execute other pieces of code.

For instance it is possible to make a single request to the parallel pool and then retrieve the outputs by means of `fetchOutputs`:

```
F = parfeval(@myFun,1,in1,in2,...);
  %% more code can be executed here while parfeval runs on the
      background
value = fetchOutputs(F);
```

Another option is making a vector of future requests on the parallel pool at the same time and/or fetching the output one at a time, as soon as they become available:

```
% we preallocate a variable with results for efficiency
f(1:4) = parallel.FevalFuture;

for idx = 1:4
    f(idx) = parfeval(@myFun,1,in1,in1,...);
end

results = NaN(1,4);
for idx = 1:4
    [completedIdx,value] = fetchNext(f);
    results(completedIdx) = value; % the value is stored in the
        variable results
    fprintf('Got result with index: %d.\n', completedIdx);
end
```

```
Got result with index: 2.
Got result with index: 1.
Got result with index: 4.
Got result with index: 3.
```

The results are stored in the variable `results`. As the example shows, the order of completion is not guaranteed. The `fetchNext` function, as we said, will retrieve the values as soon as they are ready.

The sintax of the `fetchNext` is the following:

```
[idx,B1,B2,...,Bn] = fetchNext(F)
[idx,B1,B2,...,Bn] = fetchNext(F,TIMEOUT)
```

where `idx` is the index of the `parfeval`'s completed future object and `B1,B2,...` are the future results. Furthermore, `F` is the future object and `TIMEOUT` is the time in seconds that the function can wait for the results to become available. If they don't before the timeout, then `fetchNext` will return a vector with empty arguments.

Plot the result of a function during a computation:

The `parfeval` permits to gather the values of a function before the whole completion of the computations, in order to use them for further applications. One of these applications can be to plot and keep updated a graph with all of the results already available. An example in which a series of random walk trajectories (or paths) are computed is presented below:

$$X_t = X_{t-1} + \varepsilon_t \qquad \begin{aligned} &\varepsilon_t \sim \mathcal{N}(0, \sigma^2) \\ &t = 1, \dots, T \\ &\text{with: } X_0 = \mu \end{aligned}$$

```matlab
% Example: using the parfeval function to run a parallel code in the
    background and fetch the results gradually from it
pp = parpool;

numPaths = 100;
numSteps = 252;
mn = 0;
sd = 1;

F(1:numPaths) = parallel.FevalFuture;
for i = 1:numPaths
    F(i) = parfeval(@randomWalk, 1, mn, sd, numSteps);
end

% create the shell of the plot
figure
axes()
xlim([0,numSteps])
ylim([-numSteps*sd*0.2,numSteps*sd*0.2])
ylabel("Value of random walk")
xlabel("Number of steps")
hold on

```

```
while true
    % we fetch the result
    [idx, path] = fetchNext(F);
    % we update the plot with the new results
    plot(path);
    title("Number of different paths generated: " + num2str(idx));
    if idx == numPaths; break; end
end
hold off

delete(pp);

% we define the function used in the parfeval
function path = randomWalk(mean,sd, steps)
    path = NaN(1,steps);
    path(1) = mean;
    for i = 2:steps
        path(i) = path(i-1) + normrnd(mean,sd);
    end
end
```

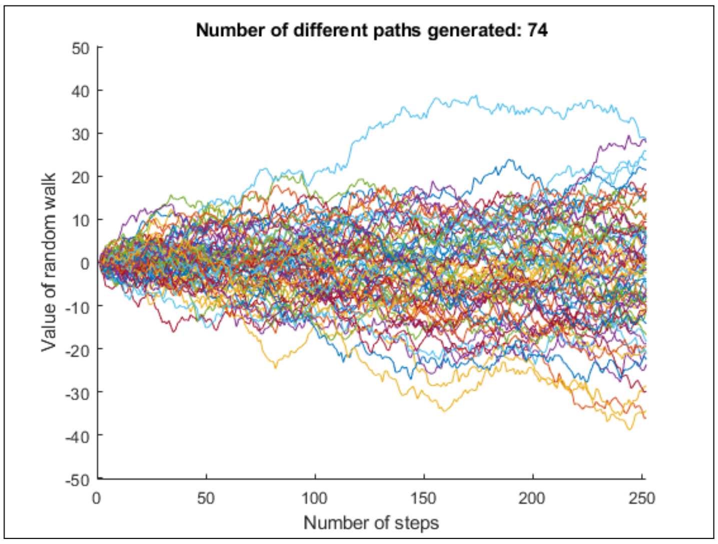


Figure 1.8: Plot that reached the 74th iteration. It is updated by means of the `parfeval` function.

When to use the `parfeval` over the `parfor`?

The coding structure of the `parfeval`, compared to the `parfor`, is more involved. More functions and steps are necessary in order to use it. It could be useful, for instance, in an optimization algorithm computed in parallel, where the `parfeval` function, fetching the result during the computation itself, could stop the algorithm as soon as the desired level of accuracy is reached. Or `parfeval` could be used when you want to plot the results as soon as they are available. In any case the main use of the function is to execute code in the background, letting MATLAB execute code and computations with the client and the remaining workers left.

1.3 Parallel computing in R

R, compared to MATLAB, is not a proprietary language. It is open-source, free to use and it is possible to contribute to it (R Core Team 2019). This is the main reason why the approach to parallel computing in R may be done in many different ways, making use of many different libraries that its users over the years have developed. In this guide one of these approaches will be analysed, without diving too deep into low-level settings and considerations that are beyond the purpose of this guide. This approach to parallel computing in R will make use of the following libraries, for which the usage and contents will be later explained in detail:

- `parallel`: built-in library used for the creation of the parallel pool. It manages the creation of the parallel structure and the communication among processors;
- `foreach`: library similar to the common `for` loop, used to split a program and assign each part to different parallel workers;
- `doParallel`: library used to create the parallel backend needed for the use of the `foreach` function.

1.3.1 The `parallel` library

The `parallel` package is developed by the R Core Team and it comes with the R installation. It represents the combination of `multicore` and `snow` packages which are packages that have been used before for parallel computation, for which it represents a replacement. These libraries contained functions to connect the different cores together and functions to take advantage of the multiple cores (R Core Team 2019). From these packages it inherited the ”master-workers” framework design. In it a master R process, running either interactively or as a batch process, creates a cluster of R workers that perform computations on behalf of the master process (Rossini, Tierney, and Li 2003).

This package is principally concerned with *coarse-grained parallelisation*, meaning that it handles running large chunks of computations in parallel. The crucial point is that these chunks of computation are unrelated and do not need to communicate in any way.

The basic computational model is:

(a) Start up M ‘worker’ processes, and do any initialization needed on the workers;

(b) Send any data required for each task to the workers;

(c) Split the task into M roughly equally-sized chunks, and send the chunks (including the R code needed) to the workers;

(d) Wait for all the workers to complete their tasks, and ask them for their results;

(e) Repeat steps (b–d) for any further tasks;

(f) Shut down the worker processes.

Among the initialisations which may be needed in the M workers there is the loading of libraries used in the program and initialize the random number stream.

Find the number of cores of your machine

```r
detectCores()
```

```
[1] 4
```

In operating systems that allow hyper-threading, like Windows, the attribute `logical` may help identify the number of actual cores.

```r
detectCores(logical = FALSE)
```

```
[1] 2
```

In setting up parallel computations it can be helpful to have an idea of the number of CPUs or cores available. It can only be considered, however, an indicative information. The program can, in fact, only determine the total number of CPUs or cores/processors physically present in the machine. This may not be the number of cores available to the current user, which may have a restriction on accessing all of them. Therefore, this information should only be taken into account as a guideline.

Initialization of the parallel pool

```r
cl <- makeCluster(<size of pool>)
   # parallel algorithm
stopCluster(cl)
```

The `makeCluster()` function creates a set of R instances running in parallel and communicating over sockets. In Unix-style operating systems the R instances are made using the fork mechanism which means that the instances created are copies of the master one. On Windows this is not possible so the R instances are started from zero. This means that an eventual additional setup (loading of libraries or setting of random number generators) is required.

After the parallel computations have been performed, it is good practice to stop the parallel pool with the `stopCluster(<cluster object>)` command.

Parallel versions of `apply` functions

One of the main direct applications of the `parallel` library is surely the use of the parallelised versions of `lapply`, `sapply`, `apply` and related functions. The parallelised analogous functions of the `lapply` function, just taken as example without the loss of generality, are

```r
parLapply(cl, x, FUN, ...)
mclapply(X, FUN, ..., mc.cores)
```

The `mclapply` function, that is not available on Windows, sets up a pool of `mc.cores` workers just for the computation that it performs, whereas, the `parLapply` function makes use of an existing parallel pool of workers, specified by the `cl` object. Therefore, with this second function the workflow is the following:

```r
cl <- makeCluster(<size of pool>)
# one or more parLapply calls
stopCluster(cl)
```

To gather information about these functions it is possible to consult R help page by typing `?apply` in the R console.

1.3.2 `foreach` and the `doParallel` libraries

The use of the foreach package Microsoft and Weston 2019b, and consequently the `foreach` function, comes from the relevant similarity in structure that shares with the very easy, widespread and familiar `for` loop. This makes it easy to convert a for into a `foreach` which has the capabilities to perform computations in parallel. In fact, unlike many parallel programming packages for R, `foreach` doesn't require the body of the `for` loop to be converted into a function. The conversion to a parallel program can be done in an easier way.

```r
results <- foreach(i=1:n) %do% {
    # computations
}
```

Registering the `doParallel` parallel backend

It is important to note that, in order to make the foreach library work in parallel, a second library is needed, the `doParallel` library. This is a package that provides a parallel backend for the foreach/`%dopar%` function using the parallel package (Microsoft and Weston 2019a). It means that it provides a mechanism to execute `foreach` loops in parallel. The user must register a parallel backend before the foreach execution, otherwise its execution will be done sequentially, even when the `%dopar%` operator is used. To open the backend it is necessary to use the `registerDoParallel` function. The function can specify the number of processors to be used or it can directly take as input a parallel object as defined by the `makeCluster` function.

This is, therefore, the structure of the `foreach` function with the proper setup of the parallel backend by means of the `doParallel` library:

```r
library(doParallel) #this will load the foreach and parallel libraries
cl <- makeCluster(2)
registerDoParallel(cl)
foreach(i=1:3) %dopar% {
    sqrt(i)
}
stopCluster(cl)
```

```
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051
```

Difference between `foreach` and `for` loops

The example in the previous subsection shows that the main difference between `for` and `foreach`, is the fact that the latter returns a list of values. It is, in fact, a function and being a function it cannot change the value of global variables but it can only return an object or a data value as result, which is a list by default. The `for`, on the contrary, does not return a value and uses side effects to convey its result. It is very common to modify the value of a variable, which was previously defined, inside of a `for` loop; the `foreach` can instead be used to directly define the value of a variable. This characteristic can easily be shown with an example:

```
result <- foreach(i=1:3, .combine='c') %dopar% sqrt(i)
result
```

```
[1]        1 1.414214 1.732051
```

The use of the `.combine` option

The example in the previous subsection uses an additional argument for the creation of the object to be returned by the `foreach` function: the `.combine` option. By default the object returned is a list, a very versatile object since it can contain any R object in itself. But there may be the need to produce different objects and this can be done by combining the various outputs into different objects. It is possible to specify the `'c'` function to get a vector as a result, because the standard `c` function is used to concatenate the results. It is possible to get a matrix as result by means of the `'rbind'` and `'cbind'` functions. It is also possible to specify other functions like, but not limited to, `'+'` or `'*'` to get a single result, respectively, the total sum or the total product of the different elements. Some examples are presented below:

```
foreach(i=1:3, .combine='cbind') %do% rnorm(4)
```

```
       result.1    result.2   result.3
[1,] -0.4245840  0.02308216  0.6581994
[2,]  0.6860865 -0.37099407 -0.6858409
[3,] -1.7815098 -2.40871376  0.4807190
[4,]  0.7813660  0.97425489 -0.4594264
```

```
foreach(i=1:3, .combine='*') %do% sqrt(i)
```

```
[1] 2.44949
```

It is also possible to use a user-defined function to combine the results. In the next example the function `cfun` takes two inputs and returns only the higher one.

```r
cfun <- function(a,b) if(a>b) a else b
foreach(i=1:3, .combine='cfun') %do% sqrt(i)
```

```
[1] 1.732051
```

The `.inorder` option, which by default is set to TRUE, is used to specify whether the order in which the arguments are combined is important. This is a relevant option when using the `foreach` function in parallel. If the order is not important, for example when using the `'+'` combining function, then the `.inorder` option may be set to FALSE, getting some performance improvements. When the call ends, the result of `foreach` will be shown in the same order as for a normal loop. However, there is no guarantee for the order in which they are combined, even if the final result is then displayed in the expected order, the same as if a sequential `for` loop was used.

Loading libraries on the parallel workers with the `.library` option

When working on the Windows operating system, the parallel workers will be created opening new R instances. This means that eventual libraries needed during the computations will have to be loaded on each worker. For this purpose the `.library` option can be added to the `foreach` function, specifying the one or more libraries to be loaded on all of the different workers.

```r
foreach(i=1:3, .library=c('library1','library2')) %dopar% {
    # computation performed on each worker
}
```

Getting information about the parallel backend

It is possible to get information about the parallel backend that was set in order to make use of the `foreach` parallel capabilities. The command `getDoParWorkers()` allows to know how many workers the `foreach` is going to use.

Two more commands may give more information, mainly useful for documentation purposes: `getDoParName()` returns the name of the currently registered backend and `getDoParVersion()` returns its version number.

Stopping the cluster

It is a good practice to close the parallel cluster after the use. We have seen in Section 1.3.1 how to close the parallel pool created by the `parallel` library with the `stopCluster(<cluster object>)` function. If, however, the cluster object was automatically created by the `registerDoParallel` function then the `doParallel` will close it automatically with the `.onUnload` function. This procedure can also be done manually by means of the `stopImplicitCluster()` function.

1.4 Access to the VERA centre's virtual machines

The VERA Centre of the Department of Economics at Ca' Foscari University of Venice has available five virtual machines:

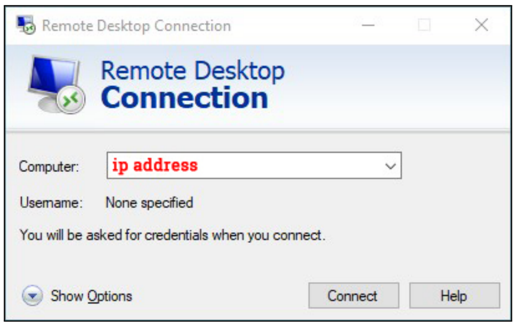
- Vera 1: Linux machine with 64 cores
- Vera 2: Linux machine with 24 cores
- Vera 3: Linux machine with 40 cores
- Vera 4: Windows machine with 40 cores
- Vera 5: Windows machine with 24 cores

1.4.1 Access from a Windows machine

Access to the Windows servers

The process of getting access to the remote desktop running windows is fairly easy, and doing so from a Windows computer does not require software installation.

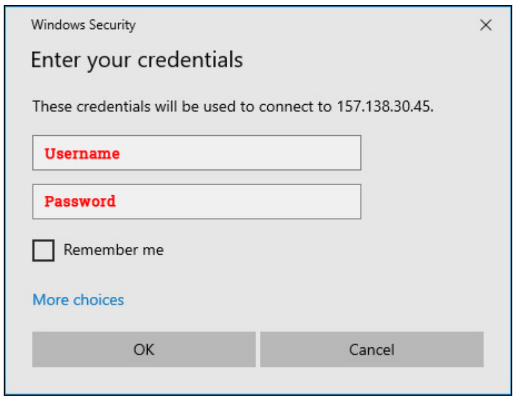
The first step is to search in the search bar for **Remote Desktop Connection**. It is a program already included in the Windows utilities. Figure 1.9 provides a snapshot of the main window of Remote Desktop Connection.



Computer: ip address
Username: None specified
You will be asked for credentials when you connect.

Figure 1.9: Remote Desktop Connection window.

After having inserted the **ip address** of the VERA machine to connect with, and having pressed **connect**, the username and password will be asked as shown in Figure 1.10.



Windows Security

Enter your credentials

These credentials will be used to connect to 157.138.30.45.

Username

Password

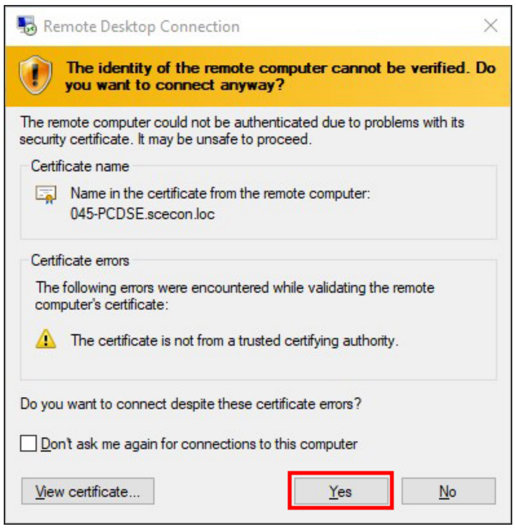
Remember me

More choices

OK Cancel

Figure 1.10: Username and password are necessary to connect to the virtual machine.

After having inserted username and password and pressed **ok**, the warning in Figure 1.11 may appear. It is sufficient to press **Yes** and the connection with the virtual machine will start.



Remote Desktop Connection

The identity of the remote computer cannot be verified. Do you want to connect anyway?

The remote computer could not be authenticated due to problems with its security certificate. It may be unsafe to proceed.

Certificate name

Name in the certificate from the remote computer:
045-PCDSE.scecon.loc

Certificate errors

The following errors were encountered while validating the remote computer's certificate:

The certificate is not from a trusted certifying authority.

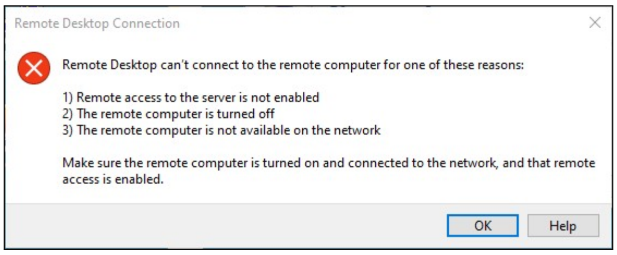
Do you want to connect despite these certificate errors?

Don't ask me again for connections to this computer

View certificate... Yes No

Figure 1.11: This warning can be ignored since the remote computer is internal to the university and can therefore be trusted.

It could happen to face an error, the one in Figure 1.12. As can be seen through the description of the error, this can be due to many factors. The machine may be available or turned off, but the most common problem is that it is impossible to connect to the remote computer because the connection is not initiated on the university's network, i.e. *eduroam*. If the connection is done in the university premises there should be no problem as long as the *eduroam* connection is used, but is is also possible to connect to the machines from anywhere. In order to do so it is necessary to make use of the University's VPN (Virtual Private Network). This program is able to simulate your computer to be connected through the *eduroam* network making all of the reserved resources accessible remotely. Anyone having a *unive* account can use the VPN.


Remote Desktop Connection

Remote Desktop can't connect to the remote computer for one of these reasons:

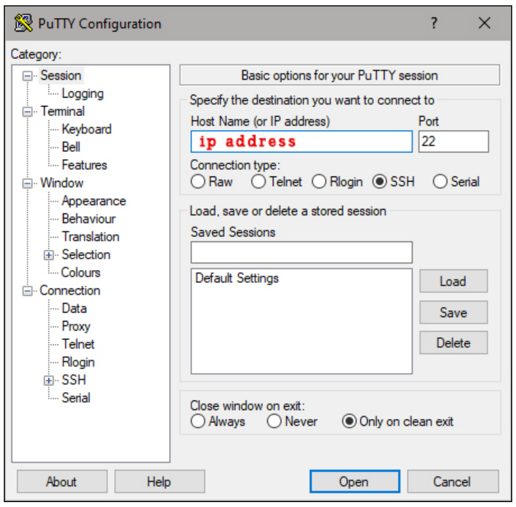
1) Remote access to the server is not enabled
2) The remote computer is turned off
3) The remote computer is not available on the network

Make sure the remote computer is turned on and connected to the network, and that remote access is enabled.

Figure 1.12: Error message. This happens because the computer is not connected at the eduroam network.

Access to the Linux servers

In order to connect to the Linux servers it is necessary to download some programs that make it a fairly easy process. The first program is called **PuTTY**, which is a free client program that manages the connection to any server with the SSH protocol, used to run a remote session on a computer over a network. In simple terms, running the program on a Windows machine creates a connection to the Linux server. It then opens a terminal that communicates to the server. Anything that is typed in that terminal is sent to the server and anything that is sent back from the server is displayed in the terminal itself. So it is possible to run on the server as if you were sitting in the console.


PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): ip address

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

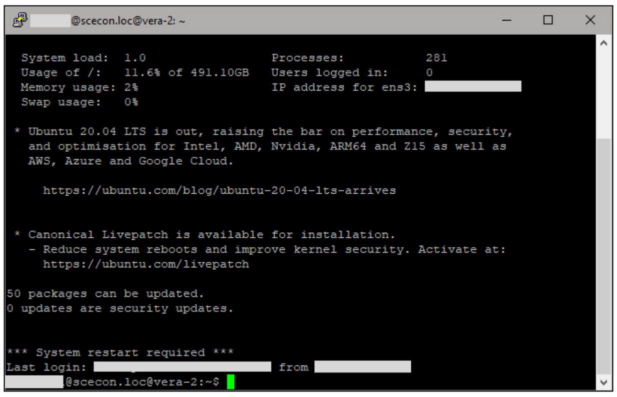
Saved Sessions

Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

Figure 1.13: PuTTY interface

In practice, when opened, the PuTTY interface is the one in figure 1.13. Here it is only necessary to insert the ip address, check that the port is specified to be 22 and press **open**. This will open the connection to the server and open a terminal window. In this terminal window the program will ask username and password. If correctly inserted then the connection will be set and it will be possible to interact with the server through the terminal, shown in figure 1.14.

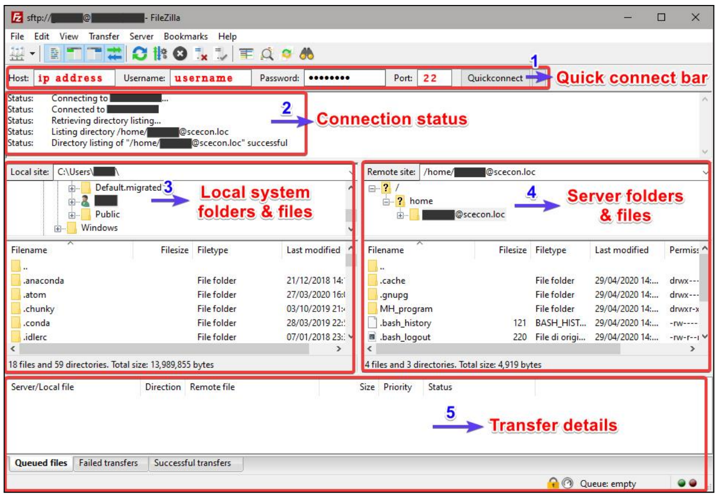


[redacted]@scecon.loc@vera-2: ~

```
  System load:  1.0                 Processes:           281
  Usage of /:   11.6% of 491.10GB   Users logged in:     0
  Memory usage: 2%                  IP address for ens3: [redacted]
  Swap usage:   0%

 * Ubuntu 20.04 LTS is out, raising the bar on performance, security,
   and optimisation for Intel, AMD, Nvidia, ARM64 and Z15 as well as
   AWS, Azure and Google Cloud.

     https://ubuntu.com/blog/ubuntu-20-04-lts-arrives


 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

50 packages can be updated.
0 updates are security updates.


*** System restart required ***
Last login: [redacted] from [redacted]
[redacted]@scecon.loc@vera-2:~$
```

Figure 1.14: PuTTY terminal connection to the Linux server

In order to make the file management and transfer from the computer to the server easier and more intuitive, a new program can be used: **FileZilla Client**. It is a free FTP (File Transfer Protocol) software that uses an intuitive graphical user interface, shown in figure 1.15, and makes possible to transfer files between a client and a server.



Host: ip address Username: username Password: ●●●●●●●●● Port: 22 Quickconnect

```
Status:    Connecting to [redacted]...
Status:    Connected to [redacted]
Status:    Retrieving directory listing...
Status:    Listing directory /home/[redacted]@scecon.loc
Status:    Directory listing of "/home/[redacted]@scecon.loc" successful
```

Local site: C:\Users\[redacted]\

Default.migrated
[redacted]
Public
Windows

Filename	Filesize	Filetype	Last modified
..			
.anaconda		File folder	21/12/2018 14:
.atom		File folder	27/03/2020 16:
.chunky		File folder	03/10/2019 21:
.conda		File folder	28/03/2019 22:
.idlerc		File folder	07/01/2018 23:

18 files and 59 directories. Total size: 13,989,855 bytes

Remote site: /home/[redacted]@scecon.loc

/
home
[redacted]@scecon.loc

Filename	Filesize	Filetype	Last modified	Permis:
..				
.cache		File folder	29/04/2020 14:...	drwx---
.gnupg		File folder	29/04/2020 14:...	drwx---
MH_program		File folder	29/04/2020 14:...	drwxr-x
.bash_history	121	BASH_HIST...	29/04/2020 14:...	-rw----
.bash_logout	220	File di origi...	29/04/2020 14:...	-rw-r--r

4 files and 3 directories. Total size: 4,919 bytes

Server/Local file	Direction	Remote file	Size	Priority	Status

Queue: empty

Figure 1.15: FileZilla user interface

The top section of the window is used to insert the credentials for the connection to the server. So, as usual, it is necessary to insert ip address, username and password. Note that also in this case it is necessary to specify the port to be 22. If not specified the program, when trying to connect will show an error and the connection will not be possible. After having clicked on **Quickconnect** the *connection status* section will show the state of the connection. Below the status section the window is divided in two. The left side shows the files on the local computer, while on the right, if the server is connected, the server files will be displayed. Between these two section it is possible to easily move files, for which transfer details will be show below.

1.5 Benchmark problems and scale-up study

1.5.1 Problems' specification and code

In this chapter some benchmark problems from numerical analysis will be used to test the efficiency of the parallel implementation, specifically analysing the potential improvements reached using the university's machines. These are examples that are prone to parallelisation, due to the embarrassingly parallel algorithm in the first case, or the computation demanding tasks in the second and third ones. These are not, in fact, data demanding problems. The parallel implementation does not, therefore, require the transfers of big amounts of data between different workers which would affect efficiency and therefore potential performance improvements and it is often the cause of lost potential improvements.

Before presenting the results achieved on the VERA machines, the examples will be discussed, and the code used in the experiment presented in the following subsections.

Monte Carlo method: simulation of π

This problem is, possibly, the most used example to present the capabilities of the Monte Carlo simulation. It is a typical example of integral approximation, as we will proceed to compute the area below a function, applied to the computation of one of the numbers that have fascinated mathematicians the most across history: π.

The idea behind this problem is simple and very easy to visualize, making it the perfect example to show off the principles behind the Monte Carlo technique. π will, in fact, be calculated by computing the area of the unit circle. π will then be extrapolated from it. Since we are computing the area of a circle of radius 1, the formula simplifies and the area of the circle will be our actual approximation of π.

$$\begin{aligned} Area &= \pi r^2 && \text{with } r = 1 \\ &= \pi \end{aligned}$$

To compute the area of the circle we will proceed by generating a large number of random points $a_i = (x_i, y_i)$ with $x_i \sim \mathcal{U}(0,1)$ and $y_i \sim \mathcal{U}(0,1)$ where $\mathcal{U}(a,b)$ denotes a uniform distribution on the interval (a,b). By generating a large number of points we will be able to cover more or less uniformly the whole area taken into consideration. As shown in figure 1.16 we can do this on a single quadrant, in this case the first one, and then assuming the same result applies to the other three. This is particularly useful because in this way the generation of the random numbers used for the points coordinates is simpler. Generating, in fact, numbers on a different range, say $[-1,1]$, requires an extra step to convert each random variable from having a $[0,1]$ range, as it is given by most random number generation functions. An extra step that would slow down the computation when repeated millions of times.

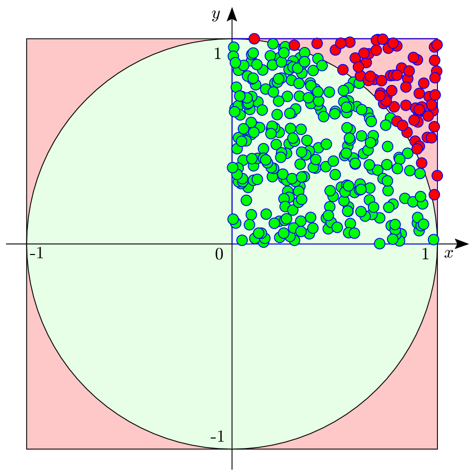


Figure 1.16: Graphical representation of the Monte Carlo method to compute π with 350 points (green and red circles).

Then we can use these random points to approximate the area of the quarter of the circle, compared to the area of the square surrounding it, by simply counting the number of points inside of the circle and divide it by the total number of points. This is the reason why this represents an example of integral numerical solution, because the result approximated is equivalent to the integral with which we can compute the area below the function. To determine if a point a_i is inside of the circle the following function is used:

$$\begin{cases} x_i^2 + y_i^2 \leq 1 & a_i \text{ lies inside of the circle} \\ \text{otherwise} & a_i \text{ lies outside of the circle} \end{cases}$$

Thus, the formula to compute the area of the circle, and, therefore π is the following:

$$Area = 4 \cdot \frac{N_{inside}}{N_{total}}$$

where N_{inside} is the variable containing the number of points generated that actually lie inside of the circle, and N_{total} is the total number of iterations that the simulation performs.

Below the code used in the experiment, both in MATLAB and R, is presented. The functions that are presented in this chapter can be used in other scripts by importing them. In MATLAB simply by having the files in the same folder it is possible to call them as `functionName(inputs)`. In R, instead, functions lying in external files can be imported with the command `functionName <- dget("filename.R")`. Both functions return a list containing both the approximation of π and the time spent to compute it, information actually used later for the scale-up analysis with different amount of cores employed.

The total workload has not been divided across single iterations, i.e. sending individuals iterations to single workers and then repeating this process until the desired amount of iterations has been reached. Instead, the amount of iterations that each worker needs to do is divided and determined beforehand. Thus the workers are instructed once and they have already defined the amount of work they need to do. The process of transferring the instructions to single workers is reduced to a minimum, being only performed once. Then, upon completion partial results are collected and combined. This procedure is ideal in this case because this parallel application has a very high number of iterations, each taking up just a small fraction of a second. Therefore, each tiny computation that is added to the single iteration adds up, being repeated millions of times, to a considerable amount of time. At the same time, being the single computation very simple and fast we do not worry about one worker taking more time than the others, therefore not taking full advantage of the parallelisation, because the difference should be minimal and the reduction of time outweighs this slim possible inefficiency.

MATLAB code for Monte Carlo simulation

```matlab
function [time, res] = BenchMC(n, nCores)
%BencMH function that, by means of MonteCarlo simulation, simulates pi
% n     : number of simulation steps
% nCores: number of cores employed

delete(gcp('nocreate'))
parpool(nCores);

tic

workload = round(n/nCores); %rounded iterations for each worker
mypi = zeros(1, nCores);
parfor i=1:nCores
    partial = 0;
    for j = 1:workload
        x = rand; y = rand;
        if (x^2+y^2) <= 1
            partial = partial + 1;
        end
    end
    mypi(i) = partial;
end

pi = (sum(mypi)/n) * 4;

time = toc;
delete(gcp('nocreate'))
res = pi;
end
```

R code for Monte Carlo simulation

```r
function(n, nCores) {
#BencMH function that, by means of MonteCarlo simulation, simulates pi
# n     : number of simulation steps
# nCores: number of cores employed

  cl <- makeCluster(nCores)
  registerDoParallel(cl)

  workload <- round(n/nCores, 0) # rounded iterations for each worker

  time <- system.time(
    mypi <- foreach(i = 1:nCores, .combine='+') %dopar% {
      piTemp <- 0
      for (j in 1:workload) {
        x <- runif(1)
        y <- runif(1)
        if (x^2+y^2 <= 1) {
          piTemp <- piTemp + 1
        }
      } #for
      return(piTemp)
    }# foreach
  )[3] #Sys.time

  mypi = (mypi/n) * 4

  stopCluster(cl)

  return(list(mypi, time))
}
```

Inversion of randomly generated matrices

The second problem consists in the computation of the inverse of a matrix that has been randomly generated. Specifically, the matrix is formed by the sum of the squared elements of two matrices, composed themselves of normally distributed numbers. Formally:

Generate two independent matrix-variate normal random variables

$$X_{jik} \overset{i.i.d.}{\sim} \mathcal{N}_{m\times m}(O_{m\times m}, I_{m^2}), \quad j = 1, 2, \quad i = 1, \dots, n, \quad k = 1, \dots, K$$

and let $\tilde{y}_{pq,ik} = (x^2_{pq,1ik} + x^2_{pq,2ik})$ with $x_{pq,jik}$ the (p, q)-th element of X_{jik}. Finally compute the inverse:

$$Y_{ik} = \tilde{Y}^{-1}_{ik}, \quad , i = 1, \dots, n, \quad k = 1, \dots, K$$

MATLAB code for random matrix inversion

```
function [time] = BenchInvMatrix(i, k, m, nCores)
%BenchInvMatrix function that invertes randomly generated matrices
%     n = parameter identifying dimension of the matrices
% nCores = num. of cores used for the computation

delete(gcp('nocreate'))
parpool(nCores);
tic

parfor v=1:i
    for u=1:k
        x=inv(randn(m,m).^2+randn(m,m).^2);
    end
end

time = toc;
delete(gcp('nocreate'))
end
```

R code for for random matrix inversion

```
function(i, k, m, nCores) {
  #BencInvMatrix function that invertes randomly generated m by m
      matrices
  # i: number of parallel iterations
  # k: number of iteration for each _i_
  # m: dimension of the matrix MxM
  # nCores: number of cores employed

  # function to compute the power of a matrix
  # taken from
      https://stat.ethz.ch/pipermail/r-help/2007-May/131330.html
  # by: Alberto Vieira Ferreira Monteiro
  matpowfast <- dget("matpowfast.R")

  cl <- makeCluster(nCores)
  registerDoParallel(cl)

  time <- system.time(
    temp <- foreach(v = 1:i, .combine='c') %dopar% {
      for (u in 1:k) {
        A <- matpowfast(matrix(rnorm(m*m), nrow = m, ncol = m), 2)
        B <- matpowfast(matrix(rnorm(m*m), nrow = m, ncol = m), 2)
        temp <- solve(A+B)
      } #for
      return(NULL)
    }# foreach
  )[3] #Sys.time

  stopCluster(cl)

  return(time)
}
```

This function makes use of the `matpowfast()` function[1] which is a fast algorithm to compute the power of a matrix as needed by the specified problem.

```
function(mat, n)
{
  if (n == 1) return(mat)
  result <- diag(1, ncol(mat))
  while (n > 0) {
    if (n %% 2 != 0) {
      result <- result %*% mat
      n <- n - 1
    }
    mat <- mat %*% mat
    n <- n / 2
  }
  return(result)
}
```

Maximum eigenvalue of random matrices

The third problem consists in the computation of the eigenvalues of big randomly generated matrices. Then all of the values are compared and the maximum is found. The eigenvalue calculation is a computationally demanding operation for which, therefore, the employment of parallel computing can be beneficial in reducing computation time.

MATLAB code for the maximum eigenvalue

```
function [time] = BenchEig(k, dim, nCores)
```

[1]Function taken from the notes of Alberto Vieira Ferreira Monteiro, available at: https://stat.ethz.ch/pipermail/r-help/2007-May/131330.html [accessed on 18/05/2020]

```
%BenchEig function that find the maximum eigenvalue of randomly
%         generated matrices
%   k = number of matrices to compute
% dim = dimension of matrices: dim x dim

delete(gcp('nocreate'))
parpool(nCores);

tic
b = zeros(1,k);
parfor i = 1:k
    b(i) = max(eig(rand(dim)));
end
time = toc;
delete(gcp('nocreate'))
end
```

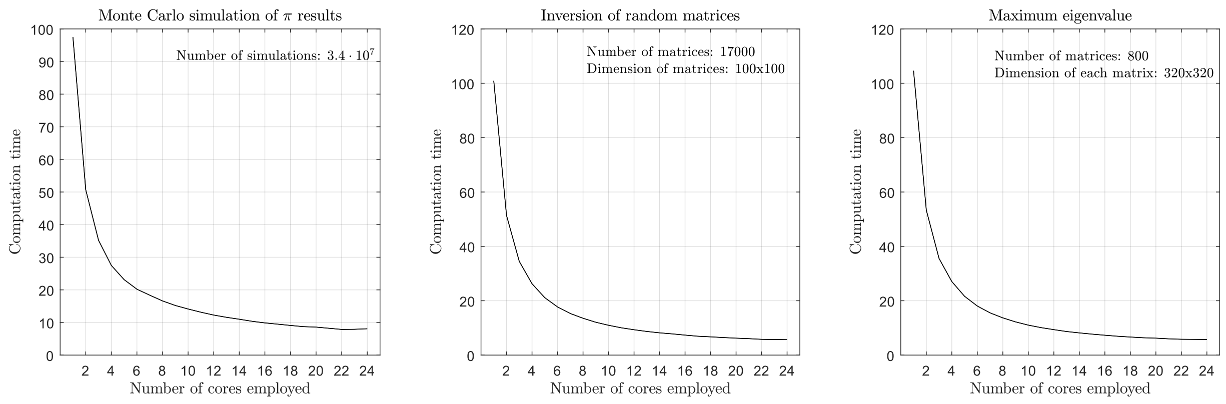
R code for the maximum eigenvalue

```
function(k, dim, nCores) {
  #BenchEig function that computes the eigenvalues or k randomly
      generated
  #         dim x dim matrices and finds the maximum value.
  # k: number of randomly generated matrices
  # dim: dimension of the randomly generated matrix: dim x dim
  # nCores: number of cores employed

  cl <- makeCluster(nCores)
  registerDoParallel(cl)

  time <-system.time(
    temp <- foreach(v = 1:k, .combine='c') %dopar% {
      m <- matrix(round(runif(n = dim*dim, min = 0,max = 1), 2), nrow =
          dim, ncol = dim)
      a <- eigen(m, only.values = T)
      b <- max(Re(a$values))
      return(NULL)
    }# foreach
  )[3] #Sys.time

  stopCluster(cl)

  return(time)
}
```

1.5.2 Scale-up study and summary of the results

A scale-up study of varying number of cores employed on the VERA's machines has been performed using the problems just described as benchmark. Each problem has been repeated with varying number of cores, from 1 to the maximum of 24. Then, the same experiment has been performed 25 times and the average outcome is taken as result. This has been done in order to mitigate the variation in computing time due to the stochastic nature of the Metropolis-Hastings algorithm.

Results of the benchmark problems with R on the Linux machine



Results of the benchmark problems with MATLAB on the Linux machine

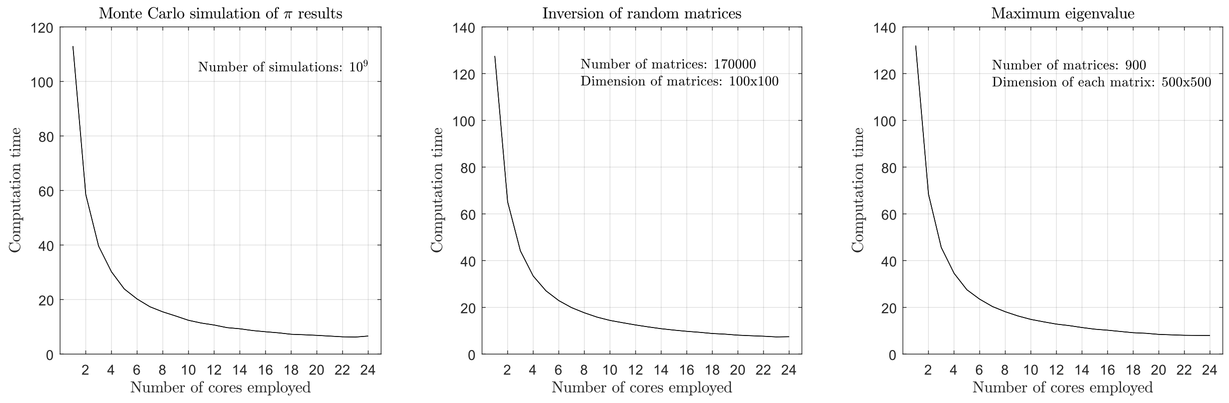


Figure 1.17: Plot of the computation time of the three benchmark problems, in relation to the number of cores employed for the computation.

The examples have been run both in R and MATLAB. They have been, however, performed with different parameter settings such that the sequential version (with just one core) took around 110 seconds to run. Different settings have been considered, to highlight the strengths and weaknesses of each language. In fact to reach an equivalent computation time for the simple Monte Carlo exercise, in R $3.4 \cdot 10^7$ simulations are performed while on MATLAB this number is increased to 10^9. For the exercise of inversion of matrices an order of magnitude is increased in the same direction, from $17,000$ in R to $170,000$ matrices in MATLAB. On the other hand, the Metropolis-Hastings has proven to be more efficient in R where each path was carried on for $4,000$ steps, while only $1,000$ on MATLAB. This process of "balancing" the programs has been done because the comparison of the two programs themselves was not the goal of the study. Not to mention that the written programs, despite being written to be as similar as possible, they may still present inefficiencies and asymmetries that influence the execution times. The goal of the scale-up-study was the one of comparing the parallel algorithm run-times when actually applied in practice with increasing number of cores employed, comparing therefore the behaviour of the two programming languages with different problems. Thus, to make it more easy to compare, the analysis needed to be performed from a similar starting point, namely the sequential version running in around 110 seconds.

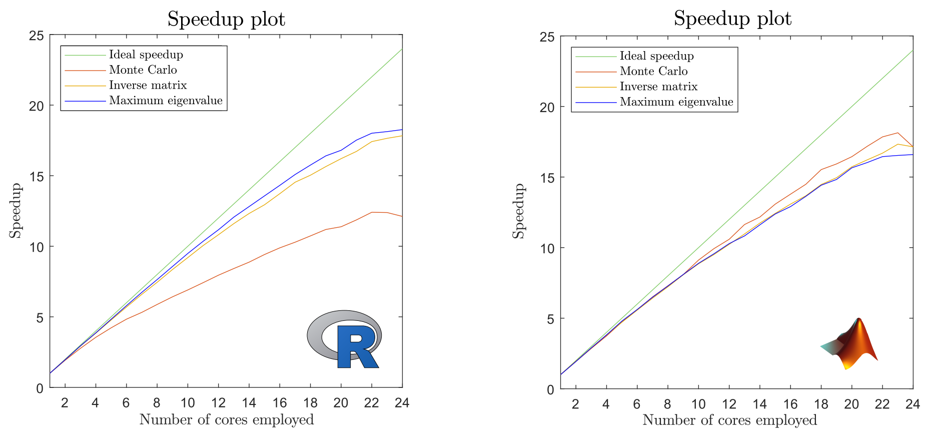


Figure 1.18: Comparison of the speedup performance of the three benchmark examples, with the reference of the ideal speedup, both for R and MATLAB.

The results, in seconds, are presented, for more details, in Table 1.1 but visualized in Figures 1.17 and 1.18. From these results the first conclusion that can be done is that the parallel computing may definitely prove itself useful determining a considerable reduction in computation time. We see from Figure 1.17 the actual results. We see a considerable reduction but it is still difficult to compare each program and extract the meaning from these numbers. This is easier to do analysing the speedup, which is a measure that, taking as reference the sequential algorithm of the respective exercise, is possible to be compared.

In Figure 1.18 we see results from both languages in two plots, respectively for R and MATLAB. In the case of R we can appreciate the constant increase in speedup, while loosing progressively efficiency as the speedup was departing from the ideal measure of speedup, i.e. the ideal scenario in which each additional core employed decreases perfectly proportionally the computation time. We can see that the benchmark problems that resulted gaining the most are the ones that are more computationally intensive: the inverse and the computation of the maximum eigenvalue of random matrices. The reason may lie in the fact that there is less communication between workers, while the workers themselves have more computations to perform. Analysing the results of the MATLAB speedup plot we observe, on the other hand, a great speedup achieved, staying fairly close to the ideal speedup. The different programs themselves yield a similar result.

To conclude this analysis we observe that is a suitable path to achieve a considerable reduction in computation time, even if done minimizing the effort, maintaining the algorithms as close as possible to the sequential ones but taking some changes, some precautions, to make it parallel.

Cores	MC time	Speedup	MI time	Speedup	EIG time	Speedup
R						
1	97.49	1.00	100.91	1.00	104.59	1.00
2	50.70	1.92	51.41	1.96	53.35	1.96
3	35.21	2.77	34.49	2.93	35.55	2.94
4	27.50	3.55	26.26	3.84	27.06	3.87
5	23.13	4.22	21.13	4.78	21.64	4.83
6	20.20	4.83	17.73	5.69	18.06	5.79
7	18.37	5.32	15.30	6.59	15.53	6.73
8	16.61	5.88	13.55	7.44	13.69	7.64
9	15.21	6.42	12.08	8.35	12.21	8.57
10	14.13	6.90	10.97	9.20	11.02	9.49
11	13.15	7.42	10.04	10.05	10.11	10.35
12	12.27	7.95	9.33	10.82	9.37	11.17
13	11.58	8.42	8.70	11.60	8.67	12.06
14	11.00	8.87	8.19	12.32	8.16	12.81
15	10.37	9.41	7.80	12.94	7.71	13.57
16	9.87	9.89	7.35	13.73	7.31	14.31
17	9.48	10.29	6.94	14.54	6.94	15.08
18	9.09	10.73	6.71	15.03	6.64	15.75
19	8.72	11.19	6.45	15.64	6.38	16.40
20	8.58	11.38	6.23	16.21	6.23	16.80
21	8.23	11.87	6.05	16.72	5.98	17.52
22	7.87	12.41	5.80	17.41	5.82	18.01
23	7.91	12.39	5.74	17.65	5.79	18.12
24	8.07	12.11	5.67	17.83	5.75	18.26
MATLAB						
1	112.94	1.00	127.56	1.00	132.01	1.00
2	58.55	1.93	65.21	1.96	68.48	1.93
3	39.61	2.85	44.16	2.89	45.63	2.89
4	30.21	3.74	33.42	3.82	34.58	3.82
5	23.89	4.73	27.07	4.71	27.55	4.79
6	20.21	5.60	22.90	5.57	23.51	5.62
7	17.36	6.51	19.89	6.42	20.36	6.48
8	15.50	7.29	17.68	7.22	18.14	7.28
9	13.99	8.09	15.81	8.07	16.32	8.09
10	12.40	9.12	14.42	8.85	14.85	8.89
11	11.39	9.92	13.42	9.50	13.81	9.56
12	10.67	10.59	12.47	10.23	12.84	10.28
13	9.71	11.64	11.63	10.97	12.20	10.83
14	9.29	12.17	10.87	11.74	11.37	11.61
15	8.64	13.08	10.27	12.42	10.67	12.38
16	8.20	13.79	9.75	13.08	10.23	12.91
17	7.81	14.49	9.33	13.67	9.69	13.63
18	7.28	15.53	8.82	14.47	9.16	14.42
19	7.10	15.93	8.53	14.95	8.91	14.83
20	6.88	16.44	8.11	15.73	8.44	15.65
21	6.61	17.18	7.87	16.22	8.25	16.03
22	6.35	17.84	7.65	16.69	8.05	16.45
23	6.26	18.14	7.36	17.33	8.01	16.53
24	6.63	17.14	7.45	17.14	7.98	16.59

Table 1.1: Detailed results of the benchmarks on the Linux machine.

Chapter 2

Simulation

Computer simulation has been employed in a wide variety of disciplines. Through computer simulation, one can study the behaviour of real-life systems that are too difficult to examine analytically. Furthermore, recent advances in methodologies, software availability, hardware improvement and stochastic optimization have combined to make simulation one of the most widely accepted and used tools in system analysis and operations research (Rubinstein and Kroese 2017). If considered, then, the sustained growth in size and complexity of emerging real-world systems and problems, we cannot expect anything but a parallel growth in popularity of these methods, that may give us a way to peek inside of problems otherwise too big and complex to tackle.

2.1 Simulation models

By *model* it is intended an abstraction of some real *system* that can be used to obtain predictions and formulate control strategies. It is, therefore, useful to clearly define what a system is. By *system* we mean a collection of related and interacting entities, called *elements*, forming a unique complex unit. The elements possess certain characteristics, or *attributes*, that take on logical or numerical values. Typically, the activities of single elements interact over time, causing changes in the system's *state* (Rubinstein and Kroese 2017). For example the system may be considered to describe a firm. Elements in this system may be the employees, the customers or the machinery. The attributes may be the skill level of the employees, inventory's capacity or machinery's efficiency. Then the system's state may be the number of sales in a certain period. Clearly interaction between the workers and customers, the division of work, the production capacity of the machinery and the number of product ready and waiting in inventory all together contribute to the determination of sales for the simulated period.

Given the attributes' numerical or logical nature, the model developed will, in turn, have a mathematical background, an analysable system to emulate the physical one. In order to be useful, two conflicting dimensions will have to be balanced: realism and simplicity. Ideally, the model should be as realistic as possible. The main goal of any model is the one of representing reality as faithfully as possible, incorporating most of the real aspects of the system. On the other hand, some degrees of realism can and should be waived in favour of simplicity. A model too complex to be analysed is of little usefulness since the main goal should be the one of being a tool to simplify interpretation of reality. Furthermore, simplicity leaves the door open to manipulation that could be useful to simulate different possible hypothetical scenarios.

After having defined the model, it is necessary to assess its validity. This may be approached in many ways. One way is to reexamine and reevaluate the formulation of the problem uncovering possible flaws. In doing so it is also useful to verify that all of the mathematical expressions are dimensionally consistent. These two methods involve valuating the mathematical expression of the model. A third way is the one of varying input parameters and check that the model behaves in a plausible manner. It is always useful during the programming phase of a model to spend some time checking this aspect. In these regards, it is useful to analyse degenerate cases, or cases with parameters that are unlikely observed in real life but for which the results may be predictable. If we take the previous example we may analyse the model's behaviour, for instance, in the situation in which there are no customers or no employees and then observing the model's outcome. Finally, the fourth method to verify the validity of a model is the, so called, *retrospective* test, which is a typical way of building and assessing machine learning models. This practice consists in building a model based on historical data and then check, again with a share of past data, how well the model would have predicted reality. In this way it is possible to measure the effectiveness that the model would have had if it would have been used. Comparing, then, this hypothetical accuracy with what actually happens indicates how well the model predicts reality. However, the big disadvantage of retrospective analysis lies in the assumption that past data is a good representation of the future, which not always may be accepted a priori.

Finally, after the model has been set up and its validity verified, it is possible to derive a solution from it. This may be done both *analytically* and *numerically*. In the first case, a solution is obtained directly by working on the formulas and expressions of the model. In the second case, a numerical solution is obtained via a suitable approximation procedure. Here is where simulation comes into play. The use of *stochastic computer simulation* - often called *Monte Carlo simulation* - introduces some randomness in the model, that, therefore, deviates from a deterministic computer model. The randomness introduced in the model, and that lets a computer program be able to compute numerical solutions to a model, also uncovers one of the shortcomings of computer simulation. Simulation, in fact, cannot give exact measures of the model, rather it can derive *sta-*

tistical estimates. Thus, even if we assume we have a model that is faithful to reality, the results that simulation is able to give are subject to a certain degree of uncertainty. The second obvious shortcoming of simulation comes from the assumption, sometimes not so acceptable, of a faithful model. If not a valid representation of reality, then the model will not be able to provide meaningful results. Lastly, simulation modelling is typically *time consuming* due to the random number generation processes needed to introduce randomness, and the large number of iterations that are needed in order to increase precision, reducing the variance of the estimate. This is exactly the flaw that parallel computing is aimed at solving, or at least reduce in relevance.

2.2 Mathematical methods to generate stochastic sequences

As it has been discussed in the previous section, simulation of complex models aimed at the generation of a numerical result often entails the introduction of randomness in the formula. And even if in some cases deterministic algorithms have been found, randomized algorithms are, generally, considerably simpler and more efficient (Vadhan 2012).

The computer, however, is unable, by itself, to achieve true randomness; the very same concept of random is a complex conundrum that mathematicians and philosophers have tried to tackle. In the early days of computer simulation randomness was generated by manual techniques, such as coin flipping, dice rolling, card shuffling up to more sophisticated techniques devised in later years like those based on the universal background radiation or the noise of a PC chip (Rubinstein and Kroese 2017). Mechanical techniques are still widely used in gambling (e.g. roulette) and lotteries but the computer-simulation community abandoned such techniques for three main reasons:

1. mechanical methods were too slow for general, modern use;
2. the generated sequences could not be reproduced;
3. it was found that the generated sequences exhibited both bias and dependence.

Modern mechanical techniques do overcome the first and third issue. The speed has increased considerably and they would pass most statistical tests for randomness, which will be discussed in Section 2.2.1. The main drawback, however, remains the impossibility to reproduce a certain sequence. Some techniques, however, have been devised and developed which lead to the generation to satisfactory and reproducible results. For this reason they are called *pseudorandom numbers.*

Pseudorandomness is the theory of efficiently generating objects that “look random” despite being constructed using little or no randomness at all (Vadhan 2012). These objects may vary in nature but those that will be considered hereafter will simply be numbers.

In general a random process generates unpredictable outcomes. A single event, any event, cannot be predicted in advance with certainty given available information. On the other hand, pseudorandom processes produce outcomes that, given some information, can be predicted. An example of a physical random process can be a six-sided fair die. On a single roll the outcome is supposed to be impossible to predict. There exists no information, given a proper roll, that could permit a prediction with certainty. Now let us consider a logbook in which a million outcomes, a million dice rolls in this case, are transcribed. Then we have a logbook that has been produced randomly and that can be produced to create (pseudo)random sequences of numbers. At the same time, however, if we know the starting position of said sequence, say starting page and row, we could perfectly predict future outcomes. The important aspect of this is that without the information given by the logbook the pseudorandom sequence exhibits statistical randomness. It does not matter that the true randomness is not reached as long as in the sequence there are not present any recognizable patterns or regularities. In fact this example is actually taken from reality. Nowadays many methods have been developed in order to generate random numbers, and computing power has increased such that thousands of random numbers can be generated in the blink of an eye. However, in the past, a logbook with random numbers was something so useful that it was actually published in 1955 by the RAND corporation. The book, *A Million Random Digits with 100,000 Normal Deviates*, was simply a long sequence of random numbers, generated by an electronic simulation of a roulette wheel attached to a computer, culminating in a 400 pages book. The sequences had been carefully filtered and tested before being used to generate the table. A few of these tests are presented in the following section.

2.2.1 Tests for statistical randomness

The first tests to check statistical randomness, which were followed by a prolific literature on the subject but that are still valid today, are four hypothesis tests that take as their null hypothesis the idea that each number in a given random sequence had an equal chance of occurring. consequently, also any other pattern should be distributed evenly with the same probability (Kendal and Babington Smith 1938).

- The frequency test is the first and most basic one. It simply checks the frequency of every single digit. The expectation is that such frequency is the same for each digit as each one has the same probability to occur;
- the serial test does the same thing of checking frequencies but for sequences of two digits at a time (e.g. 00 01 02...). Again the observed frequency is then compared

to the expected one. This test verifies that the sequence is not locally random, i.e. no digit tends to be followed by another specific digit. For instance the sequence formed by repeating 1234567890... passes the first test but not this second one;

– the poker test consists on using all possible categories obtained from poker that uses a hand of five cards. Thus, checks for 5 digits sequences comparing them to the probability for them to happen;

Name	**Pattern**	**Probability**
All different	ABCDE	0.3024
One Pair	AABCD	0.5040
Two Pairs	AABBC	0.1080
Three of a kind	AAABC	0.0720
Full house	AAABB	0.0090
Four of a kind	AAAAB	0.0045
Five of a kind	AAAAA	0.0001

Table 2.1: Probabilities tested by the poker test (Abdel-Rehim et al. 2015).

– the gap tests, lastly, controls the gap between two digits. For instance the sequence 00 has a gap of 0 between the two zero digits. The sequence 050 has a gap of 1, the sequence 093510 has a gap of 4 and so on. The gap itself has a certain expectation. In the example in which we consider the digit 0 then we may expect it to be followed by another 0, having therefore a gap of 0, one tenth of the times.

A sequence that is able to pass all of these tests up to a certain confidence level (usually 5%) is said to be *locally random*. Kendall and Smith introduced this concept differentiating it from the concept of *true randomness*. A truly randomly generated sequence, in fact, may not be locally random to a certain degree. It could happen that these sequences are random overall but do not present randomness in some of their smaller blocks, for instance with many rows with the same number, meaning that they would be useless to certain statistics applications.

2.2.2 Random number generation: the Linear Congruential Generator

The building block to introduce randomness to any stochastic process is the generation of independent uniformly distributed random variables. These variables are then used to generate any random variable that follows a different probability distribution, and these methods will be discussed in Section 2.2.3.

For most applications, these ways of generating random numbers could be just considered a black box since most computer languages already have built-in code to generate them. The user is usually only requested to input the starting seed, X_0, which if not specified is picked by the computer itself, usually based on the current date and time when the program is run. In MATLAB uniformly distributed random numbers can be generated by means of the function `rand(...)` and the seed set with `rng(<seed>)`. In R the function `runif(...)` generates the numbers while `set.seed(<seed>)` sets the seed.

For some applications, however, understanding the generation of random numbers is very useful, and for the parallel computing purpose can prevent some mistakes that could lead to unwanted results. The simplest method for generating pseudorandom sequences is to make use of *linear congruential generators* (Lehmer 1951). This method uses a recursive formula to generate a deterministic sequence $\mathbf{X}$ of pseudorandom numbers:

$$X_{t+1} = (aX_t + c) \pmod m$$

where the initial value X_0 is the seed, a, c and m are all positive integers and are respectively called the *multiplier*, the *increment* and the *modulus*. The function modulo-m (mod m) means that the result of the formula is divided by m and then only the remainder is taken, becoming the element X_{t+1} in the sequence. These components are the variables shaping the resulting sequence. If $c = 0$ then the generator is called a Lehmer Random Number Generator. The key component is the variable m that is the component of the modulo function. This variable, in fact, defines the *period length* of the sequence. Each component of the sequence can only be, in fact, smaller than m, assuming uniquely values from the set $\{0, 1, ..., m - 1\}$. We have that this sequence is uniformly distributed and we can transform it such that its support becomes [0, 1], as most of the uses of uniform distributed random variables require, simply by:

$$U_t = \frac{X_t}{m} \sim \mathcal{U}\ (0, 1)$$

The random variable $U_t \sim \mathcal{U}\ (0, 1)$ is easy to generalize to have any support, so that $\tilde{U}_t \sim \mathcal{U}\ (a, b)$. To expand, or shrink, the support of the variable it is sufficient to multiply the random variable by a constant c, i.e. $U_t \cdot c = U_t{}' \sim \mathcal{U}\ (0, c)$. By setting $c = b - a$ it is possible to create a random variable with the desired gap: the difference between a and b. The second step is the one of translating the gap, which can be done by simply adding, or subtracting, a constant from the variable, i.e. $U_t{}' + a = \tilde{U}_t \sim \mathcal{U}\ (0 + a, c + a)$

This sequence of random numbers is periodic, meaning that it will repeat itself at most every m iterations. For example, let $a = c = X_0 = 2$ and $m = 7$. Then the recursive formula will be $X_{t+1} = 2X_t + 2 \pmod 7$ which will produce the sequence, including the starting point, 2, 6, 0, 2 which has period 3. The parameters will, therefore, have to be carefully chosen to yield an optimal result without involving computations that result too computationally expensive. For instance prime numbers are generally good m candidates to produce long sequences. Prime numbers, in fact, reduce the likelihood of obtaining as result the starting point, the seed, which would restart the sequence. The more this event can be delayed by choosing the right numbers, so the larger the period of the sequence, the better generation of random numbers we can aim to achieve.

More in general good generators are those that pass a large part of statistical tests. Two examples, taken from Palczewski's notes of a course in Computational Finance[1], are presented in Figure 2.1. The example shows the distribution of the sequence using two different sets of parameters, $a = 1229, c = 1, m = 2048$ on the left and $a = 1597, c = 51749, m = 244944$ on the right. While the histogram depicting the distribution seems fairly unbiased, considering the small sample size, and in any case no clear difference among the two examples can be highlighted, the scatter plot presents clear differences in the randomness. In particular, each point represents two consecutive elements in the sequence, one in the x coordinate, and the second in the y. Thus, the plot represents the relation between each point and its predecessor. In the first example this relationship is biased, meaning that the sequence is not statistically random, and, therefore, not usable for many simulation purposes. The second example presents no correlation between subsequent numbers, denoting a more random behaviour. It is important to note that these considerations are different to the ones about the period of the sequence. These are two different observations that need to be simultaneously evaluated in order to obtain good generators. In this case, since the period of the sequences was not the relevant aspect, two non-prime m variables have been chosen in order to display their different behaviour.

2.2.3 Random variable generation: the Inverse-Transform Method

In the previous section it has been discussed the generation of uniformly distributed random numbers. In this section good uniformly distributed generated sequences will be taken as given and will be used in the generation process of sequences of random variables following distributions that are not uniform. For this purpose we will employ the *Inverse Transform Method.* This method allows us to generate one-dimensional random variables from a prescribed distribution.

Let X be a random variable with cumulative distribution function (CDF) F_X. X is the random variable that we want to simulate. Since F_X is non-decreasing then F_X^{-1} may be defined as

$$F_X^{-1}(u) = inf\{x : F_X(x) \geq u\} \ , \quad 0 \leq u \leq 1.$$

The infimum function is used since cumulative distribution functions are weakly monotonic and right continuous. If U is a uniform random variable on $(0, 1)$, $U \sim \mathcal{U}(0, 1)$, we need to show that by setting $F_X^{-1}(U) = X$ we get a random sequence with CDF F_X.

[1]Lecture notes accessed on 12/05/2020 from: https://www.mimuw.edu.pl/∼apalczew/CFP_lecture3.pdf

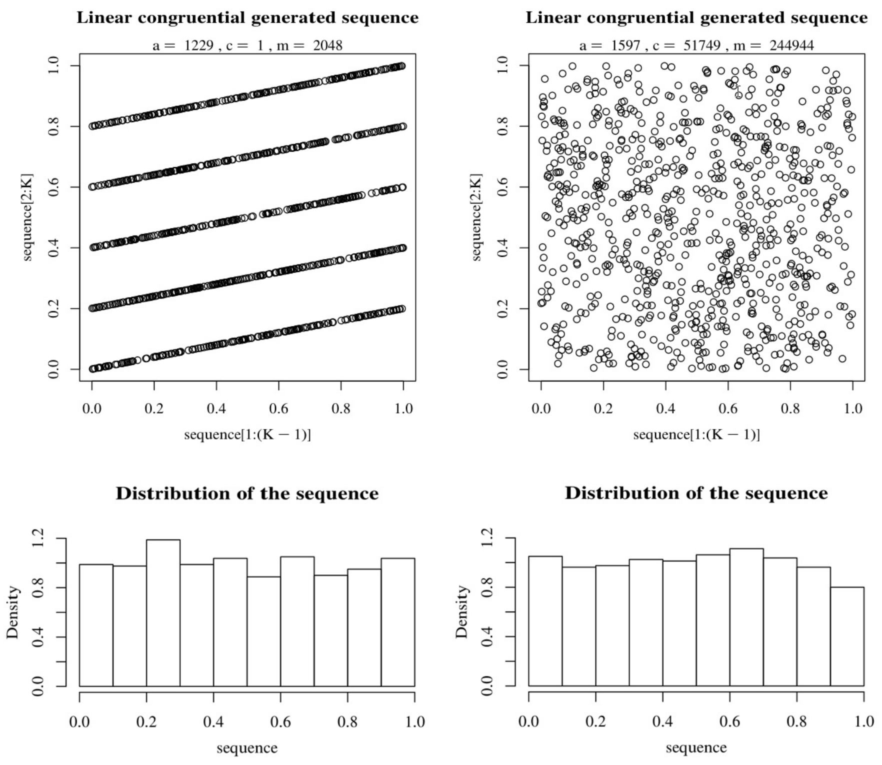


Figure 2.1: Two pseudo random sequences. The one on the left presents a correlation between subsequent numbers. The one on the right, instead, is locally random in addition to being globally random.

To show this, assuming F_X is invertible, we have:

$$\begin{aligned}\mathbb{P}(F_X^{-1}(U) \le x) &= \mathbb{P}(U \le F_X(x)) && \text{by applying } F_X \text{ on both sides}\\ &= F_X(x) && \text{since } \mathbb{P}(U \le y) = y \text{ when } U \sim \mathcal{U}(0,1)\end{aligned}$$

$$\Rightarrow F_X(x) = \mathbb{P}(F_X^{-1}(u) \le x)$$

Thus, if we have F and we can compute its inverse F^{-1}, by drawing uniform random variables U such that $U \sim \mathcal{U}(0,1)$ we can get X to be a random variable having CDF F by setting $X = F^{-1}(U)$. This method is intuitive to understand through a graphical visualization, as the one in Figure 2.2.

The algorithm of this methods is, therefore, the following:

Inverse-Transform Method

Needs: inverse cumulative distribution function F.

1. Generate u_i from $\mathrm{U}(0, 1)$;
2. compute $x_i = F^{-1}(u_i)$;
3. repeat steps 1 and 2 n times;
4. return X.

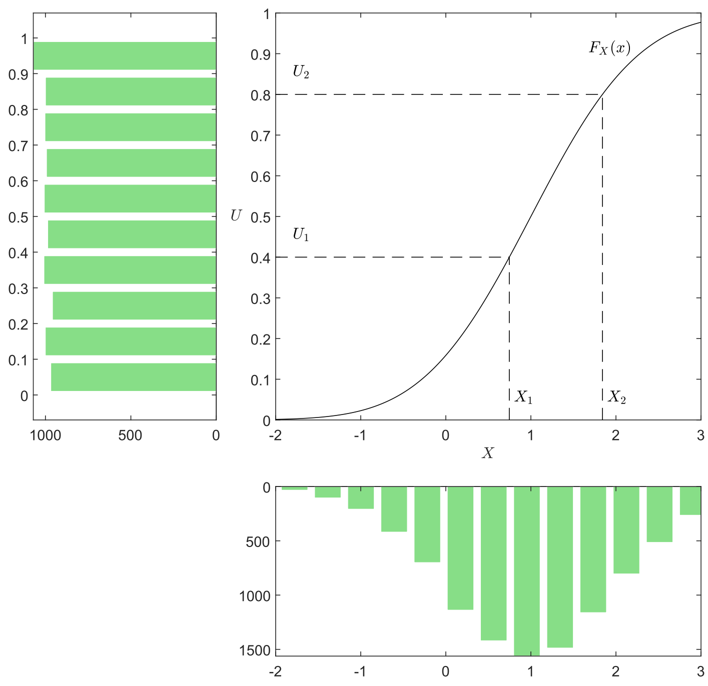


Figure 2.2: Graphical representation of the Inverse Transform Method

A strength of the Inverse-Transform Method is the fact that it can be used to draw from a discrete distribution. Let X be a discrete random variable with $\mathbb{P}(x = x_i) = p_i$, with $i = 1, 2, \ldots$, such that $\sum_i p_i = 1$ and $x_1 < x_2 < \ldots$. The CDF F_X of X is given by $\sum_{i:x_i \leq x} p_i$.

The algorithm will be slightly different and can be rewritten as such:

Inverse-Transform Method: discrete case

Needs: inverse discrete cumulative distribution function F.

1. Generate u_i from U$(0, 1)$;
2. Find smallest positive integer, k, such that $u \leq F(x_k)$. Let $x_i = x_k$;
3. repeat steps 1 and 2 n times;
4. return X.

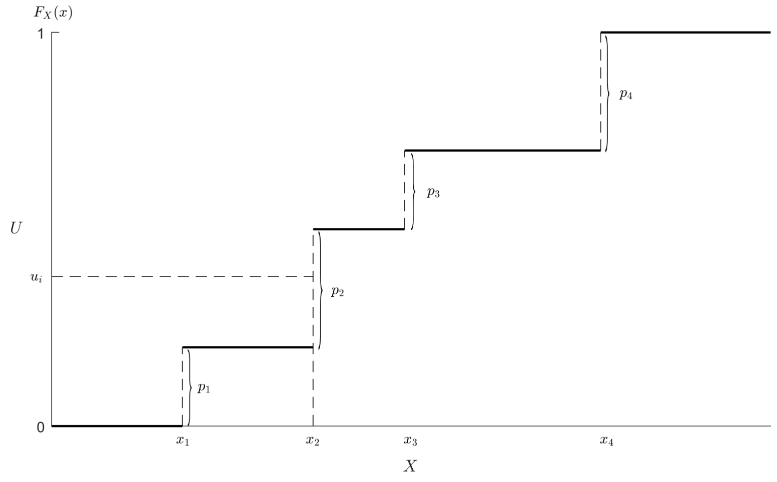


Figure 2.3: Graphical representation of the Inverse Transform Method applied to a discrete distribution

What is important to bear in mind when using this method is the fact that in order to apply it we need to have a CDF F for which it is possible to compute the inverse, either analytically or algorithmically. We need to be able to solve $F(x) = \int_{-\infty}^{x} f(t)dt = u$ with respect to x and this is not always easy to achieve. In order to generate random variables from distributions for which we cannot compute the inverse cumulative density function, other methods are used. In this thesis the Metropolis-Hastings method is discussed in depth in Chapter 3.

2.3 Variance reduction techniques

The concepts that this thesis presents are focused on the implementation of simulations in parallel. Regardless, when writing about computer simulation techniques it is relevant to talk about variance reduction. Parallel computing and variance reduction are, fundamentally, two extremely different topics but they share the common objective of increasing simulation's performance. With the parallel computing approach, performance is improved by exploiting the unused computational potential of computers. On the other hand, variance reduction enhances the simulation's performance by improving the accuracy of its estimators. It does so by means of utilising known information about the model. These are, in fact, techniques that take advantage of knowledge about the system that is available.

A simple technique is the use of *common random variables.* Rubinstein (2017) uses a simple example to show situations in which it could be useful and how it has an impact on the reduction of variance. Let X and Y be random variables with known cdfs, F and G. Suppose the need to estimate $l = \mathbb{E}[X - Y]$ via simulation. The simplest unbiased estimator for l is $X - Y$. The random variables X and Y can be simulated by means of the Inverse Tranform method:

$$\begin{aligned} X &= F^{-1}(U_1), \quad U_1 \sim \mathcal{U}(0,1), \\ Y &= G^{-1}(U_2), \quad U_2 \sim \mathcal{U}(0,1). \end{aligned}$$

In this setting the precision on the simulated estimator can be measured by the variance,

$$\text{Var}(X - Y) = \text{Var}(X) + \text{Var}(Y) - 2\text{Cov}(X, Y)$$

and since the marginal cdfs of X and Y have been prescribed, it follows that the variance can be minimized by maximizing the covariance of the two variables. Thus the random variables X and Y need to be not independent. In particular, it is said that common random variables are used if it is set $U_2 = U_1$. Since both F^{-1} and G^{-1} are nondecreasing functions, then

$$\text{Cov}(F^{-1}(U), G^{-1}(U)) \geq 0.$$

Furthermore, using common random variables proves to be maximizing the covariance between X and Y (Whitt 1976). Thus, variance reduction is achieved: the variance of the estimator $F^{-1}(U) - G^{-1}(U)$ is, in fact, smaller than the *crude Monte Carlo* estimator $X - Y$.

In the case in which the model requires the estimation of $\mathbb{E}(X + Y)$ a different technique offers the same advantages of variance reduction: the generation of *antithetic random variables.* This technique sets $U_2 = 1 - U_1$ and this, just as with common variables in the first case, minimizes the variance of the estimator.

Variance reduction can also be employed to drastically improve the simulation of rare events. In this case the general idea is to modify the selection of random samples in such a way that the desired events occur more frequently (effectively biasing them) than they would normally, while simultaneously taking these changes into account in order to obtain unbiased estimates (Biondini 2015). This method is called *importance sampling* and is the most fundamental variance reduction technique (Rubinstein and

Kroese 2017).

Let

$$\mu = \mathbb{E}_f[H(X)] = \int H(x)f(x)dx$$

where H is the sample performance and f is the probability density for X. A modified density g, called the *importance distribution* or the *instrumental distribution*, is introduced to apply a change of measure:

$$\mu^{IS} = \int H(x)\frac{f(x)}{g(x)}g(x)dx = \mathbb{E}_g\left[H(X)\frac{f(X)}{g(X)}\right].$$

The resulting integral is evaluated numerically by using a i.i.d. sample $X_1, \ldots, X_n$ from g:

$$\hat{\mu}_n^{IS} = \frac{1}{n}\sum_{i=1}^{n} H(X_i)\frac{f(X_i)}{g(X_i)} \tag{2.1}$$

The ratio $W(x) = \frac{f(x)}{g(x)}$ is called the *likelihood ratio* or *importance weights* and it is the term controlling for the bias generated by the introduction of the distribution g.

The choice of the importance sampling density g is linked with the resulting variance of the estimator $\hat{\mu}$ in 2.1. From this derives the problem of variance reduction of $\hat{\mu}$ with respect to g:

$$\min_g \text{Var}_g\left(H(X)\frac{f(X)}{g(X)}\right). \tag{2.2}$$

The solution to this problem has been proved (Rubinstein, Melamed, and Shapiro 1998) to be:

$$g(x)^* = \frac{|H(x)|f(x)}{\int|H(x)|f(x)dx}$$

and the resulting density is called the *optimal importance sampling density.*

These techniques are particularly relevant because their use may determine a significant reduction in the number of different simulations that are needed in order to achieve a certain result. Also parallel computing may provide a tool to achieve the same, or even better, results compared to the sequential version reducing the time of computation. Nevertheless, it may not be considered a substitution to a model that is properly designed and for which these techniques, if possible, have been employed.

Chapter 3

Introduction to Metropolis-Hastings algorithm

The Metropolis-Hastings is an algorithm that allows us to sample from an arbitrary generic probability distribution, our target distribution, even if we don't know the normalizing constant or, in general, we are not able to compute such function easily, making it possible to use other methods to generate such sample.

Creating a sample from a given distribution "resembles" the mere computation of an integral. The histogram of a well built sample, in fact, follows the shape of the probability density function. In some cases the computation of such integral may prove, however, impossible to perform analytically and numerical methods may be hindered as well. In situations like these the Metropolis-Hastings algorithm may provide a solution, producing as output a sample sequence of the target distribution.

3.1 Motivations of the method

The Metropolis-Hastings (M.-H. hereafter) is an algorithm that allows us to sample from a generic probability distribution, the *target distribution*, that may otherwise be difficult to sample from. There may be many reasons why computing an integral like

$$\mathfrak{I}(h) = \int_{\mathcal{X}} h(x) \mathrm{d}\pi(x)$$

may be difficult or impossible. In this example $\mathrm{d}\pi$ is a probability measure while $\mathcal{X}$ is the domain, or support, of the function. Each of these elements may be source of complexities difficult to overcome. In a scenario like this Monte Carlo methods could provide a solution: exploiting the probabilistic nature of π and its weighting over the domain $\mathcal{X}$ is the most natural and most efficient way to produce approximations to integrals connected to π and to determine the regions of the domain $\mathcal{X}$ that are more heavily weighted by π (Robert 2016). As we have seen in the previous chapter, the Monte Carlo approach, relying on the ability to produce a large number of simulations of a random variable following a specific distribution, takes advantage of the stabilisation of the empirical average, property given by the Law of Large Numbers (Rubinstein and Kroese 2017). However, given the difficulty, or in some cases impossibility, of sampling from a specific distribution, then the (standard) Monte Carlo methods is not able to provide a solution.

In this scenario an indirect approach to the simulation of complex distributions is necessary. We need to take a step back and observe the fact that we can evaluate each point on the support space $\mathcal{X}$ if compared to a second one. To do so a Markov chain associated to the target distribution π is used, taking advantage of it to validate the convergence of the chain to the distribution of interest.

Let π be a probability density function, the *target distribution*, defined on a state space $\mathcal{X}$. π is computable up to a multiplying constant, so that $\pi(x) \propto \tilde{\pi}(x)$. We are able to compute $\tilde{\pi}(x)$ but we do not know the normalizing constant. The M.-H. algorithm, developed from the work of Metropolis et al. (1953) and the work of Hastings (1970), proposes a Markov chain that is ergodic and stationary with respect to π, meaning that if $X^{(t)} \sim \pi(x)$, with $X^{(t)}$ being the tth element of the Markov chain, then $X^{(t+1)} \sim \pi(x)$ as well. Therefore, the chain will converge in distribution to π (Robert 2016).

If the Markov chain is ergodic, meaning that it forgets its starting value, it is not necessary to determine when the chain reaches stationarity since the empirical average

$$\hat{\mathfrak{J}}(h) = \frac{1}{T}\sum_{t=1}^{T} h(X^{(t)}) \xrightarrow{a.s.} \mathfrak{J}(h) \ ,$$

so it converges almost surely to the value that we are looking for. This implies that, in theory, simulating a Markov chain is intrinsically equivalent to a standard i.i.d. simulation from the target, the difference being in a loss of efficiency (Robert 2016). It is, in fact, necessary to produce a high number of simulation to reach a given variance with the Monte Carlo estimator.

The advantage of this methods lies in the approach that differs from other methods, as the accept-reject method (Robert and Casella 2009), that aims directly at the "big picture" by accepting or discarding each proposal subject to a passing test. A similar process to this is also performed by the M.-H. algorithm but the approach differs in the fact that the "picture" is built progressively: the target distribution's shape is gradually formed by local exploration of the state space $\mathcal{X}$, ideally until all the regions of interest have been uncovered.

An analogy for this method may be a hound employed in the research of missing people. We can imagine, in fact, a hound that is instructed by making it smell a piece of clothing of the missing individual. Then we can assume that the hound is let free to roam from a arbitrary, almost random, point; let's say the edge of a wood. The hound will wander, taking small steps, and at smelling the ground. Then it will evaluate how the smell changed compared to the previous step and decide (here we should assume a degree of randomness in the decision) whether to continue from there or go back to the previous step. Slowly the hound will wander around enough to have a clear idea of the smells in the territory, that we defined as the support of the function, and it will eventually converge to the mean, saving the missing person.

In addition to these convenient characteristics of the method, the Markov chain that the algorithm produces, $X^{(1)}, X^{(2)}, \cdots, X^{(t)}, \cdots$, is such that $X^{(t)}$ is converging to π. The result is a chain that can be considered as a sample from π, the distribution itself. Due to the Markovian nature of the chain, the transition from one element to the following one in the sequence, that we described as a small step, is highly dependent from the previous one. In particular this is relevant for the initial values, extremely dependent on the starting value $X^{(1)}$ which mat prove to be very far from the relevant areas of the distribution. For this reason this aspect must be taken into account when setting the algorithm. This, among other considerations regarding the practical application, will be expanded in Section 3.3.

3.2 The Metropolis-Hastings algorithm

Resuming the notation from the previous section, let π be our target distribution. The M.-H. algorithm requires the choice of a conditional density q, also called *proposal distribution* or *candidate kernel*. Let $\{X^{(t)}\}_{t=1}^{T}$ be a sequence of random variables. Such sequence is generated with a M.-H. algorithm equipped with the proposal $q(\cdot|X^{(t)})$. The transition from one element of the Markov chain to the following, from $X^{(t)}$ to $X^{(t+1)}$, proceeds by means of the following algorithm:

Metropolis-Hasting algorithm

At the t-th iteration, given $X^{(t)}$

1. Generate $Y^{(t)} \sim q(y|X^{(t)})$
2. Take

$$X^{(t+1)} = \begin{cases} Y^{(t)} & \text{with probability} \quad \alpha(X^{(t)}, Y^{(t)}) \\ X^{(t)} & \text{with probability} \quad 1 - \alpha(X^{(t)}, Y^{(t)}) \end{cases}$$

where
$$\alpha(x,y) = \min\left\{\frac{\pi(y)}{\pi(x)}\frac{q(x|y)}{q(y|x)}, 1\right\}$$

The chain is irreducible, meaning that, over time, it will eventually explore the entirety of the support $\mathcal{X}$. This determines that, thanks to the accept-reject step of the algorithm, the simulation from an almost arbitrary proposal q is turned into a sequence that preserves π as the stationary distribution. Of course this is true in theory, but results depend on the actual choice of q and its parameters. This is the topic of Section 3.3, in which the different errors leading to a wrong output are analysed.

3.3 Diagnostics and calibration of practical implementation

In order to capture the mechanisms and to visualize the different problems that may arise when applying the M.-H. algorithm an example is presented taken from the article of Robert (2016). In particular it is a *random walk Metropolis-Hastings algorithm.*

Our target density is a perturbed version of the normal $\mathcal{N}(0, 1)$ density, $\varphi(\cdot)$,

$$\tilde{\pi}(x) = sin^2(x) \times sin^2(2x) \times \varphi(x).$$

The proposal distribution function is a uniform $\mathcal{U}(x - \theta, x + \theta)$:

$$q(y|x) = \frac{1}{2\theta}\mathbb{I}_{(x-\theta,x+\theta)}(y).$$

It is a function that exploits as little as possible of the target distribution, proceeding to a local exploration each step potentially spanning from $X^{(t)} - \theta$ to $X^{(t)} + \theta$. The proposed value $Y^{(t)}$ is simulated as:

$$Y_t = X^{(t)} + \varepsilon_t,$$

where ε_t is a random perturbation with distribution q. It is called *random walk M.-H.* all the cases when q is symmetric, $q(y|x) = q(x|y)$, thus when computing α the formula gets simplified to:

$$\alpha(x, y) = \min\left\{\frac{\tilde{\pi}(y)}{\tilde{\pi}(x)}, 1\right\}.$$

The generic nature of the algorithm determines that it remains valid for almost every choice of proposal q, regardless it being symmetric or not. This also means that no indication, about proper proposal functions and/or calibration of such proposals, is given by the theory. In the example given, in fact, the method is valid for every choice of θ but it is actually a critical decision to do. We can see the proposal distribution as a random walk kernel, and θ is the parameter determining how far the random walk will oscillate. Furthermore, since the computation $\alpha(x, y)$ is independent of θ, a poor choice of it can impact even more the results. The different behaviour of the algorithm is shown in Figure 3.1. In it the M.-H. algorithm given from the example presented above is repeated for the same number of iterations, $T = 10^4$, with different parameters of the proposal distribution θ. The b) plot shows the results when using the parameter $\theta = 1$, which is the value yielding the better results. In this ideal case the chain is able to explore the whole support giving a result that represents pretty accurately the target distribution $\pi(x)$, which has been plotted after being appropriately normalised by numerical integration. If the parameter is too low, as in plot a) where $\theta = 0.1$, then the chain will not be able to explore the whole support. The random walk makes steps that are too small and with which it is too difficult to overcome the attracting power of a mode, thus being trapped in it. Eventually, since the chain is, from the theory, ergodic it will explore the entire support. But it requires too many steps loosing in efficiency. On the other side of the spectrum there is the case visualized in plot c). In this third example we have a parameter $\theta = 100$ that is too big, leading to candidates y that are very often relatively far from the points of high density of the target distribution. Therefore, when computing $\alpha(x, y)$ we have that the result is too often very small. What happens is that the vast majority of candidates are rejected. In this way the support is not explored properly leading to biased results that would require, as in the previous case, a higher number of iteration to produce an accurate result.

Figure 3.2 illustrates the difference in performance when the θ parameter varies, via the autocorrelation graphs of three chains. The parameters differ slightly from the previous ones, $\theta = 0.3, 3, 30$, in order to better illustrate the variation in performance. This graph shows that the chain produced with $\theta = 3$ should be preferred, measured as the autocorrelation between different steps in the sequence. An autocorrelation that faster approaches zero is a sign that the sequence it measures is properly exploring the support, having therefore more information overall (Robert 2016).

Another parameter that needs to be taken into consideration when implementing the algorithm is the starting point X_0. The Markov chain, as it has been said ear-

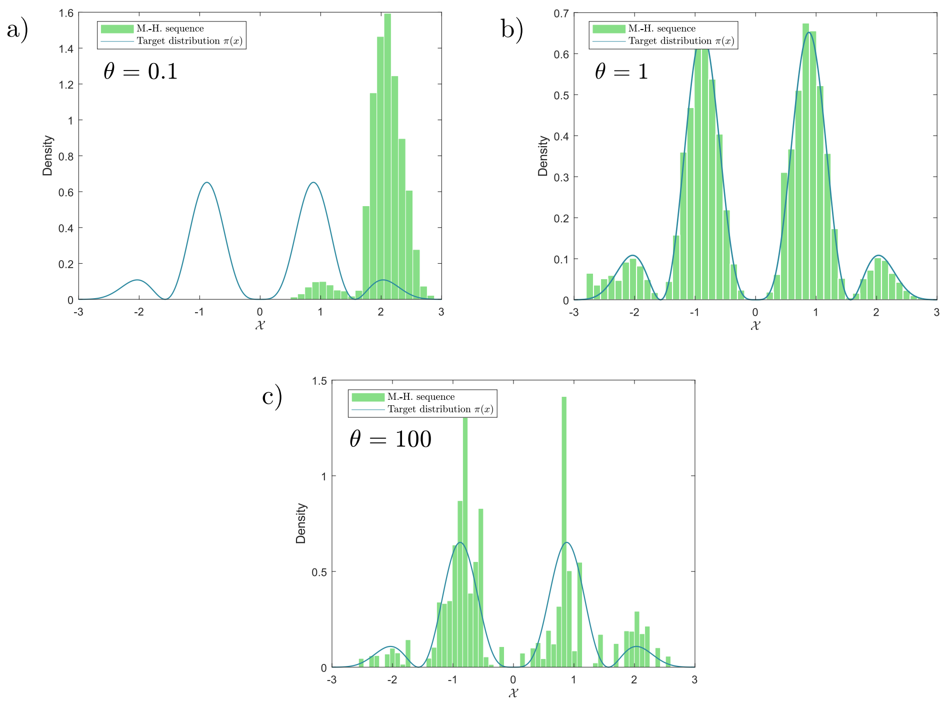


Figure 3.1: Plot showing the different sequences, plotted as histograms, formed by the M.-H. algorithm with proposal distributions varying in their parameter θ. We can observe that if this parameter, determining how far the proposal Y_t can be from X_t, is too low or too high then the sequence is not able to explore the support properly, producing biased sequences.

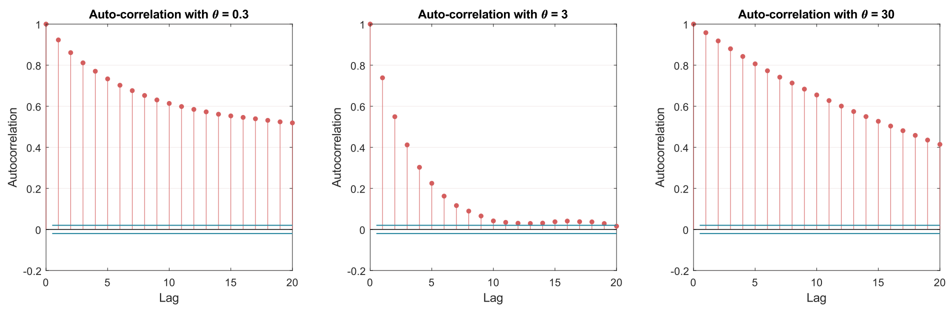


Figure 3.2: Autocorrelation between lagged elements in the sequence with different θ parameters. The better result is for $\theta = 3$ since it contains more information as the autocorrelation decreases faster.

lier, is ergodic, so it will eventually explore the whole support space $\mathcal{X}$. However, the starting point is a relevant parameter affecting the efficiency of the algorithm. Since the transition kernel is a function that “translates” the chain one step at a time, if the starting point is very far from the points in which the density accumulates then a share of the overall iterations will be devoted to reaching stationarity. Usually this part of the sequence is eliminated; it is called the *burn-in*. This is done in order to leave the time to the sequence to reach the point in which the exploration becomes more meaningful. This behaviour is illustrated in Figure 3.3 where the values of the sequence are plotted in a sequential manner. This plot is called the traceplot of the M.-H. sequence. The simulation has been performed starting with $X_0 = 20$ to show the path from that point to where the higher values of the target distribution function lie. It is easy to see in this representation the transition period which is not a real representation of the target distribution. More meaning is yield when the sequence gets closer to the value 0, around which the function assumes higher density and the sequence displays a different behaviour, due to the stationarity of the sequence.

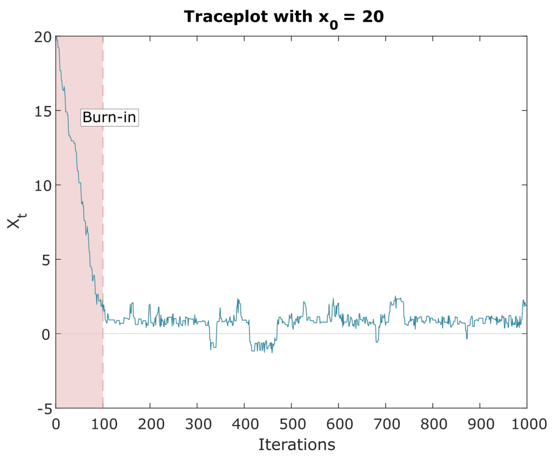


Figure 3.3: Traceplot of a M.-H. sequence starting from the area of stationarity, where the target probability density function is higher. The share of the sequence needed to reach stationarity, referred to as the *burn-in*, gets discarded as it does not contain meaningful information.

Finally, one of the parameters giving information about the calibration of the M.-H. algorithm is the acceptance rate. Since there is, for each iteration t of the algorithm, an acceptance step in which the next candidate, $Y^{(t+1)}$, can be accepted becoming $X_{(t+1)}$ or rejected making the previous value be held for the following step, $X_{(t+1)} = X_t$, the acceptance rate can be defined as the share of candidates that are being accepted, on average. It is, in fact, computed as the empirical average:

$$a = \sum_{t=1}^{T} \mathbb{1}_{\{u<\alpha(x_t,y_t)\}}$$

where $y_t \sim q(y|X^{(t)})$ and $u \sim \mathcal{U}(0,1)$. This gives us some information about the quality of the candidates that are being generated. If the acceptance rate is too high then the proposals $Y^{(t)}$ generated are too close to the previous term in the sequence.

Consequently the ratio $\tilde{\pi}(y)/\tilde{\pi}(x)$ will be close to 1, possibly even higher, making almost every proposal be accepted. Consequently, a sequence generated as such gives very little information. It does not differ too much from a sequence that may simply be generated from the proposal distribution and the result is that it does not explore the support properly. On the other side of the spectrum, if the acceptance rate is too low, it means that most of the proposal are very far from the previous element in the sequence, and they would not be accepted. They are not meaningful and do not let the sequence to explore the support since the sequence will be composed by only few different elements. Figure 3.4 illustrates the different behaviour of the sequences when exploring the support space formed by proposals with different parameters θ.

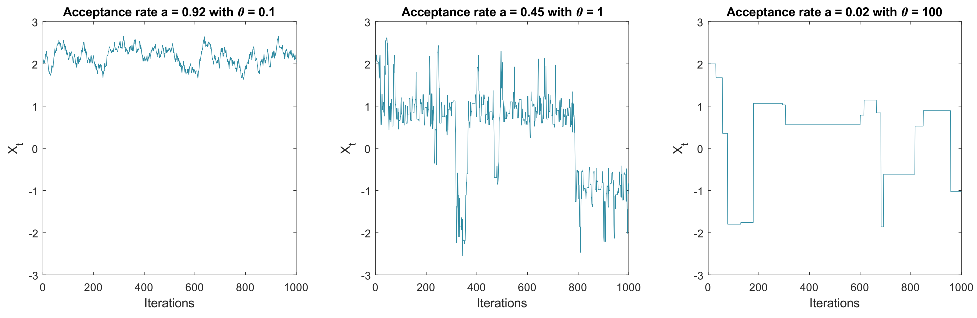


Figure 3.4: Different traceplots of M.-H. sequences with different values of θ, illustrating the different exploration that it determines. With $\theta = 0.1$ the sequence is not able to explore fully the support. With $\theta = 100$ very few proposal get accepted by the algorithm, determining a biased exploration. The better result comes from the sequence generated with $\theta = 1$ where the sequence is able to jump from one mode to the other

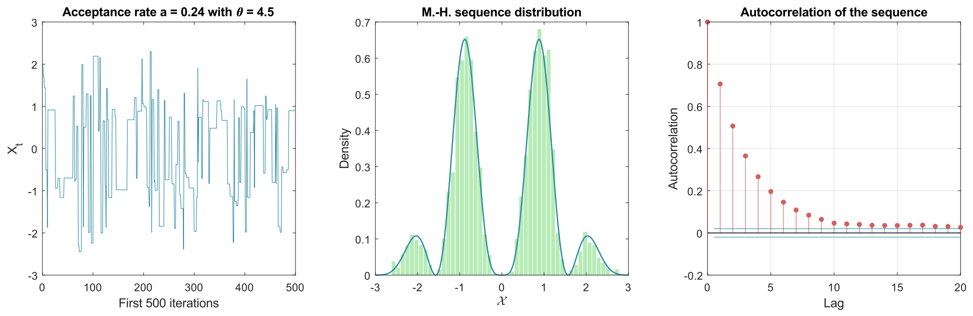


Figure 3.5: The results of the Metropolis-Hastings algorithm with optimal calibration of its parameters.

It is difficult to determine an optimal value that the acceptance rate should reach in order to achieve a high degree of efficiency, reducing the Monte Carlo variance. Roberts et al. (1997) studied a formal Gaussian setting aiming at the ideal acceptance rate that would maximize the efficiency of the algorithm. Their result, under fairly general conditions gave rise to a very useful heuristics for random walk M.-H. applied in practice: "tune the proposal variance so that the average acceptance rate is roughly 1/4". Figure illustrates this in the problem that has been taken as example for the section and with the tools that have been explored: the plot of the first 500 elements of the sequence (the number of elements plotted has been reduced for visual clarity), the histogram showing the distribution of the generated sequence and finally the autocorrelation function of the sequence.

Chapter 4

Application of the M.-H. algorithm in parallel

4.1 Problem applied in the parallel analysis

This chapter will revolve around the discussion of the practical application of the M.-H. algorithm, in particular taking advantage of the parallel capabilities that modern computers have and that have been discussed in Chapter 1. All of the topics will be explored starting from a specific example that is now presented and that will be used for the final analysis of the improvements in computation time with parallel implementation[1].

The target distribution function $\tilde{\pi}(x, y)$, which will be the function that we aim to simulate, comes from a bivariate mixture of normal distributions, $\frac{1}{3}\mathcal{N}_2(-\boldsymbol{\iota}, I_2) + \frac{2}{3}\mathcal{N}_2(\boldsymbol{\iota}, I_2)$ where $\boldsymbol{\iota} = (1,1)'$ and I_2 is the 2-dim identity matrix. Boldface variables represent two-dimensional vectors, i.e. $\mathbf{x} = (x_1, x_2)' \in \mathbb{R}^2$, which represent x and y coordinates on the support space $\mathcal{X}$. Thus, the target distribution will be:

$$\tilde{\pi}(\mathbf{x}) = \frac{1}{3}\varphi(\mathbf{x}|-\boldsymbol{\iota}, I_2) + \frac{2}{3}\varphi(\mathbf{x}|\boldsymbol{\iota}, I_2)$$

where $\varphi(\cdot|\boldsymbol{\mu}, I_2)$ is the normal density function given mean $\boldsymbol{\mu}$ and standard deviation I_2.

The problem at hand is a *random walk Metropolis-Hastings problem* since the proposal distribution $q(\cdot|\mathbf{x}^{(t)})$ is a symmetric function, not depending on the target function:

$$\mathbf{y} \sim \mathcal{N}_2(\mathbf{x}, \sigma^2 I^2).$$

[1]Example taken from Robert Casarin's lecture notes for PhD the course on Advanced Econometrics (September 17, 2019) at Ca' Foscari University of Venice

The proposal distribution is a myopic random walk generating random paths in the two-dimensional support $\mathcal{X}^2$. Its symmetry guarantees that the computation of $\alpha(x, y)$ in the algorithm gets simplified to:

$$\begin{aligned}\alpha(\mathbf{x}, \mathbf{y}) &= \min\left\{\frac{\pi(\mathbf{y})}{\pi(\mathbf{x})}, 1\right\} \\ &= \min\left\{\frac{\frac{1}{3}\exp\left\{-\frac{1}{2}(\mathbf{y}+\boldsymbol{\iota})'(\mathbf{y}+\boldsymbol{\iota})\right\} + \frac{2}{3}\exp\left\{-\frac{1}{2}(\mathbf{y}-\boldsymbol{\iota})'(\mathbf{y}-\boldsymbol{\iota})\right\}}{\frac{1}{3}\exp\left\{-\frac{1}{2}(\mathbf{x}+\boldsymbol{\iota})'(\mathbf{x}+\boldsymbol{\iota})\right\} + \frac{2}{3}\exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\iota})'(\mathbf{x}-\boldsymbol{\iota})\right\}}, 1\right\}\end{aligned}$$

4.1.1 A note on the parallelisation of the algorithm

Since each step of Markov chains depend on, at least, the previous one in the chain, then the M.-H. algorithm is not parallelisable. At least it is not parallelisable if applied in a single chain. But it is possible to run multiple different chains at the same time, one completely independent from the others. This results in is a different approach in the calibration of the implementation which will be explored in Section 4.2.2.

4.2 Calibration of the parallel algorithm

As discussed in Section 3.3, the calibration of the practical implementation of the algorithm is key to produce more precise and unbiased results. The objective, however, is to perform a parallel implementation of the M.-H. algorithm. Consequently, a few different considerations need to be made beforehand:

- since the parallel implementation entails multiple different chains, the impact of the starting points is greater on the overall amount of iterations. It has been shown the fact that each chain needs a certain amount of steps in order to reach the point of interest of the chain, to reach, therefore, stationarity. This number of steps is called *burn-in*. If there are more chains, then, each of these will need this number of iterations. Therefore, more attention needs to be put in the choice of the first element $\mathbf{x}^{0,j}$, for each different chain j.
- if the starting point $\mathbf{x}^{0,j}$ of each chain assumes greater importance, the parameters of the proposal distribution may have a smaller impact. This statement depends heavily on the proposal distribution adopted by the model, but the possibility to have different chains with different, random, starting points may help find a solution to one of the problems of the implementation of the M.-H. algorithm: the risk of the chain not exploring the whole support space, for instance because stuck in a mode of the distribution and not being able to "get out" of it and explore eventual others. This behavior is easily visualized in a random walk M.-H. when the candidates proposed are very close to the the previous step in the chain. If

this is the case then the chain will not be able to explore the whole space, at least not in a reasonable/efficient amount of steps, and a symptom of this is the high acceptance rate as discussed in Section 3.3. This aspect, in the example on which this chapter builds upon, is further analysed in Section 4.2.2.

4.2.1 Parameters used in the analysis

Keeping in mind these considerations, it is possible to analyse and calibrate the specific problem at hand. First, it is useful to visualize the target distribution $\pi(x, y)$. In Figure 4.1 it is possible to observe the main areas of the support $\mathcal{X}^2$ in which the distribution is concentrated. From the knowledge gained analysing the plot, given that many different starting points may yield better results as will be discussed in Section 4.2.2, an optimal area of deploying the numerous M.-H. chains can be $(x_1^{0,j}, x_2^{0,j})$ where $x_1^{0,j}, x_2^{0,j} \sim \mathcal{U}(-3, 3)$, for every different chain j.

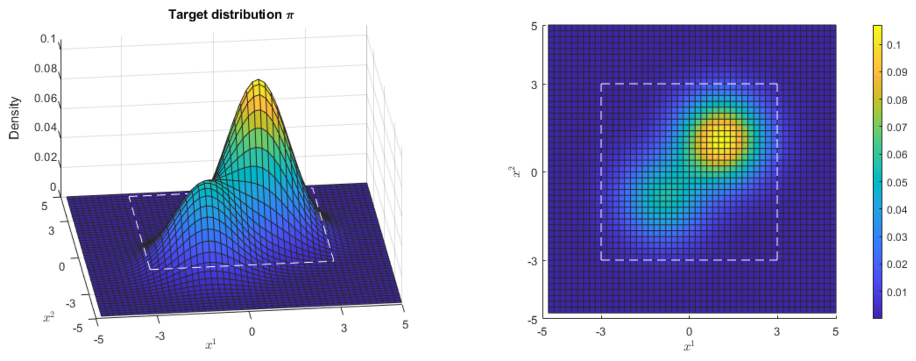


Figure 4.1: plot of the target distribution $\pi(x_1, x_2)$ with the optimal starting area highlighted.

Having defined the area in which all of the parallel chains will start, it is important to set the parameters of the proposal distribution. In this example the proposal distribution is a random walk depending on the parameter σ which defines the variance of the normal random variable generating the new step. As it has been done in Section 3.3, various different parameters have been tested to see the one yielding the better results. Figure 4.2 shows the different behaviour of the M.-H. sequence generated from the proposal distribution when σ is varied. It is interesting to see the leftmost sequence, equipped with $\sigma = 0.08$, exploring almost exclusively the lower mode and having difficulty to get out of it. It remains trapped. The middle plot, on the other hand, being equipped with $\sigma = 8$, takes bigger steps and this lets it explore both of the modes more uniformly. The last plot displays a sequence with $\sigma = 80$. Here the parameter seems to be too large, determining very few steps to be accepted. The number of iterations of the algorithm is, in fact, the same for all of the sequences even if the number of actual unique steps is different. This leads to the analysis of the other important aspect that this representations can give: the number of steps accepted, or *acceptance rate*. The discussion of the optimal acceptance rate in the case of random walk M.-H. algorithms has lead to the heuristic of aiming at a rate of roughly 1/4 (Roberts, Gelman, and Gilks 1997). This second consideration leads to the same conclusion, namely that the optimal parameter σ is 8. The traceplots of a sample sequence with $\sigma = 8$ are visualized in Figure 4.3.

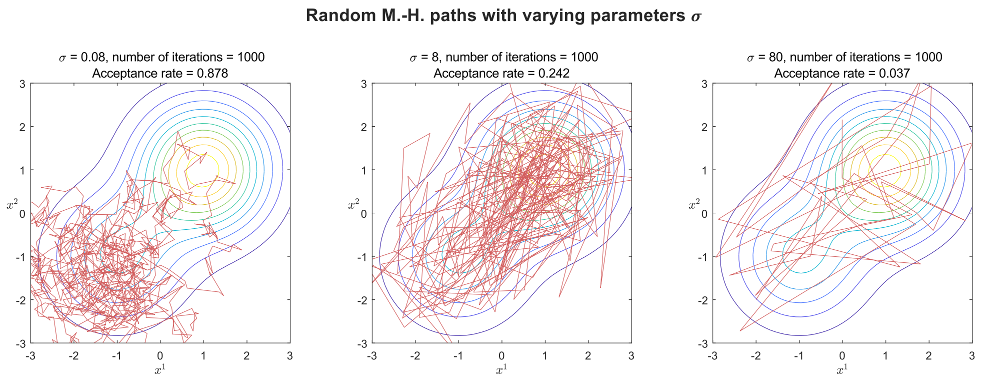


Figure 4.2: Different behaviour of the M.-H. sequences generated with different parameters σ.

4.2.2 Improving the results with different starting points: the pathological case

In a situation in which the model has not been well calibrated, i.e. the parameter of the proposal distribution σ is too small and the path has difficulties exploring the whole support, breaking the sequence into multiple, independent, sequences that have random starting points may yield better results. This may prevent the situation in which the sequence gets stuck into a mode. In Figure 4.4 this pathological case is visualized. The two plots have been generated with the same number of total M.-H. iterations, as the same is also the parameter $\sigma = 0.02$. On the left plot a single chain is generated and does explore only one mode. On the other hand, on the right plot, four independent sequences, each composed of one fourth of the steps of the unique one, are deployed in different points of the support. In the histograms below, the distribution of all of the components of the sequences are displayed and it is easy to see the bias that the first sequences has fallen into, by exploring only one mode. On the other hand the set of four sequences better simulate the target distribution by appreciating both of the modes, even though not in a precise way. This, in fact, can not be taken as a solution making a proper calibration of the model superfluous. That is, in any situation, the first thing to be done and the one guaranteeing better results.

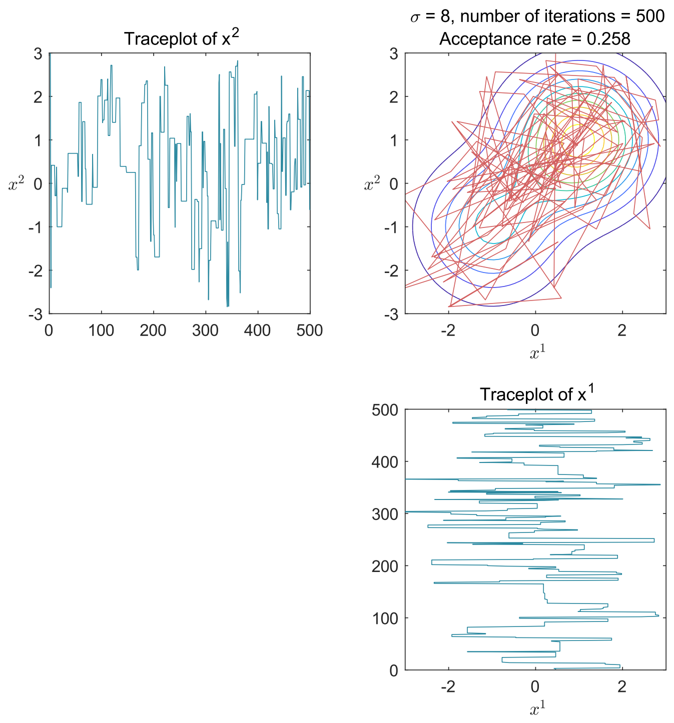


Figure 4.3: Traceplots of the components of the optimal sequence, generated with $\sigma = 8$.

4.3 Code

This section presents the implementation of the discussed problem in the two different programming languages for which parallel capabilities have been presented: MATLAB and R. The code has been written to be as symmetrical as possible in order to be able to perform a comparison of the results produced by the two languages.

The functions presented below have been written to directly manage the parallel implementation of the algorithm. They require the same 4 inputs:

- `nMH`: variable specifying the number of iterations of the algorithm that each different path needs to do. This is, therefore, the length of each individual parallel sequence;
- `nPaths`: variable controlling the number of different and parallel M.-H. sequences;
- `sig`: parameter σ of the proposal distribution;
- `nCores`: variable controlling the number of cores that the computation will involve. Each core will, in parallel, perform the computations needed to generate a single M.-H. sequence at a time, until the number of different sequences, `nPaths`, is reached.

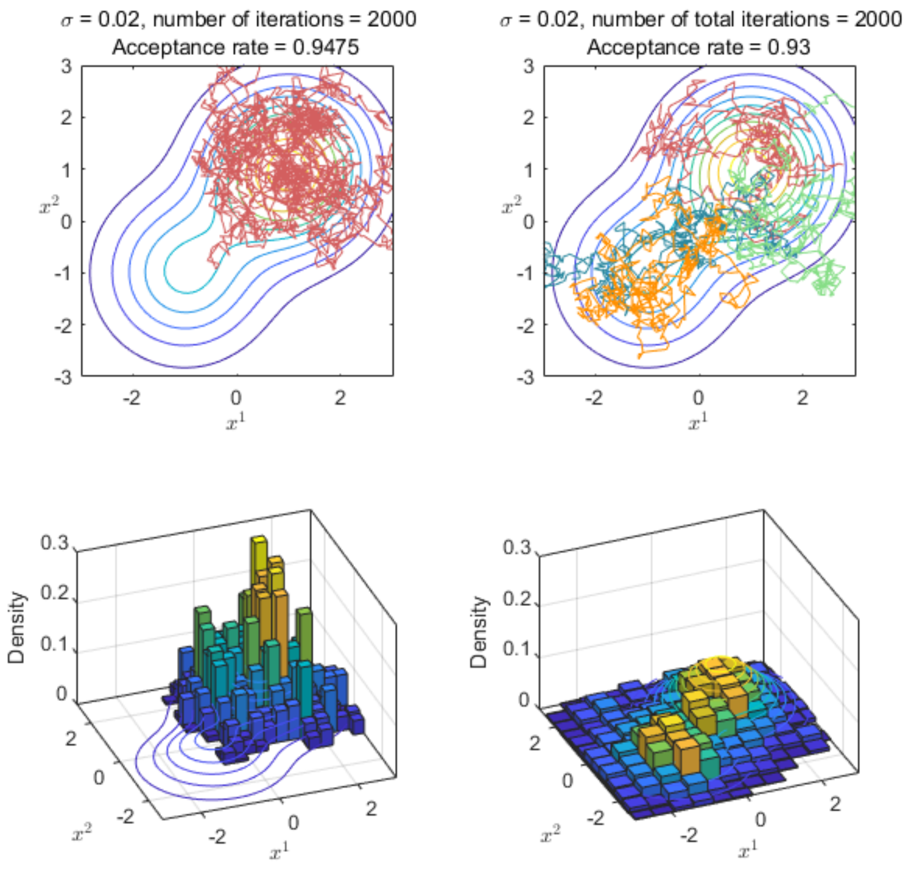


Figure 4.4: Illustration of how the simulation by M.-H. can be improved by deploying many different chains vs. a single, longer, one.

As in the case of the functions used as benchmark in Section 1.5, these functions can be recalled from a different script which can, in turn, control the behaviour the parameter of the function. This is what has been done for the scale-up study, both in Sections 1.5.2 and 4.4.

4.3.1 MATLAB code for the Metropolis-Hastings algorithm

```
function [time] = BenchMH(nMH, nPaths, sig ,nCores)
% MHtime function that evaluates in parallel nPaths different MH paths.
%   NOTE: this function evaluates a specific pdf.
%   nPaths: number of different paths to be computed
%   nMH:    number of steps for each path
%   sig:    variance of acceptance alpha
%   nCores: number of cores employed in the computation

delete(gcp('nocreate'));
parpool(nCores);
tic

z1sequence = zeros(nMH, nPaths);
z2sequence = zeros(nMH, nPaths);

parfor idx = 1:nPaths
    % we create support variables to be used inside of the nested for
        loop
        % this avoids errors due to the access of the same variable in
        % multiple workers simultaneously
    z1 = zeros(nMH,1);
    z2 = zeros(nMH,1);
    z1(1) = rand(1)*6-3; % starting x~unif[-3,3]
    z2(1) = rand(1)*6-3; % starting y~unif[-3,3]
    for i=2:nMH
        z1(i) = z1(i-1);
        z2(i) = z2(i-1);
        z1star=z1(i-1)+sqrt(sig)*randn(1,1);
        z2star=z2(i-1)+sqrt(sig)*randn(1,1);
        u=rand(1,1);
        alfaxy=min([exp(...
            log(1/3*pdf('normal',z1star,-1,1)*pdf('normal',z2star,-1,1)+...
                2/3*pdf('normal',z1star,1,1)*pdf('normal',z2star,1,1))-...
            log(1/3*pdf('normal',z1(i-1),-1,1)*...
                pdf('normal',z2(i-1),-1,1)+...
            2/3*pdf('normal',z1(i-1),1,1)*...
                pdf('normal',z2(i-1),1,1))...
            ),1]);
        if u<alfaxy
            z1(i) = z1star;
            z2(i) = z2star;
        end
    end %for

    z1sequence(:, idx) = z1;
    z2sequence(:, idx) = z2;
end % parfor

```

```
time = toc;
delete(gcp('nocreate'));
end
```

4.3.2 R code for the Metropolis-Hastings algorithm

```
function(nMH, nPaths, sig, nCores) {
  # MHtime function that evaluates a single M.-H path with nMH steps.
  #   NOTE: this function evaluates a specific pdf.
  #   nPaths: number of different paths to be computed
  #   nMH:    number of steps for each path
  #   sig:    variance of acceptance alpha
  #   nCores: number of cores employed in the computation

  cl <- makeCluster(nCores)
  registerDoParallel(cl)

  time <- system.time(
    MHPaths <- foreach (i = 1:nPaths, .combine='cbind') %dopar% {
      z1 <- rep(0, nMH) # Vector representing a column of the results
          variable
      z2 <- rep(0, nMH)
      z1[1] = runif(1)*6-3; # starting x_unif[-3,3]
      z2[1] = runif(1)*6-3; # starting y_unif[-3,3]

      u <- runif(nMH)

      # M.-H. Algorithm:
      for (i in 2:nMH) {
        # next step in the random walk:
        z1star <- z1[i-1] + sqrt(sig)*rnorm(1);
        z2star <- z2[i-1] + sqrt(sig)*rnorm(1);
        alpha <- exp( log(1/3 * dnorm(z1star,-1,1) * dnorm(z2star,-1,1) +
                          2/3 * dnorm(z1star, 1,1) * dnorm(z2star, 1,1) ) -
                        log(1/3 * dnorm(z1[i-1],-1,1) *
                            dnorm(z2[i-1],-1,1) +
                            2/3 * dnorm(z1[i-1], 1,1) * dnorm(z2[i-1],
                                1,1) ) )
        if (is.na(alpha)) alpha = 0
        alphaxy <- min(alpha, 1)


      if (u[i] < alphaxy) {
        z1[i] = z1star
        z2[i] = z2star
      } else {
        z1[i] = z1[i-1]
        z2[i] = z2[i-1]
      }
    } # for

  Path <- z1
  Path <- cbind(Path, z2)
  return(Path)
  } #foreach
)[3] #system.time

stopCluster(cl)

return(list(MHPaths,time))
}
```

4.4 Results

Using the programs presented in the the previous sections, a scale-up study similar to the one with the benchmark problems, done in Section 1.5 has been performed. The machine used is equipped with Linux Ubuntu with a 24 core Intel® Xeon® Gold 6148 CPU @ 2.40GHz, the same VERA's machine used in the previous experiment.

The experiment consisted of running both R and MATLAB versions of the M.-H. algorithm with varying number of cores, from a single one (the sequential version) to the employment of all 24. The experiment, then, has been repeated 25 times and the average value has been computed.

The outcomes obtained show a different parallel potential: MATLAB's speedup result stays much closer to the ideal speedup which, as previously defined, is the speedup that a parallel program can potentially achieve in optimal conditions, i.e. when the addition of a core to the pool grants a proportional decrease of computation time, or a constant increase in speedup. The parallel program that has been designed in R, on the other hand, seems to be losing efficiency faster as the number of cores employed increases, as shown by the speedup curve that is lower than MATLAB's across the entire spectrum of the experiment.

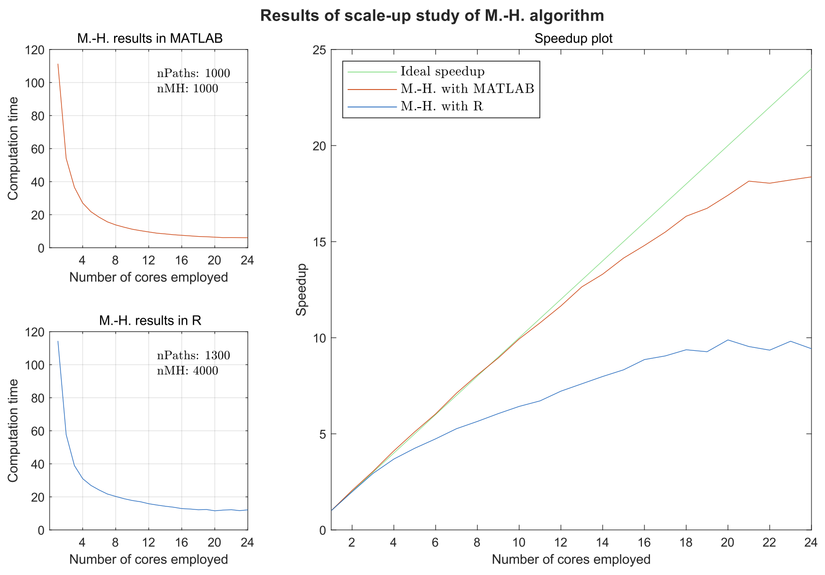


Figure 4.5: Results, in seconds, of the M.-H. computation times and the speedup in MATLAB and R. The speedup graph illustrates an interesting difference in the optimization of parallel problems in R and MATLAB.

This result is in-line with the results of the scale-up study performed with the benchmark problems, showcased in Section 1.5. Also in the previous case the R results highlighted a lower potential to be gained compared to MATLAB.

Nevertheless, these experiments show that in both cases some improvement is to be gained through parallelisation, with a relatively small cost of conversion from the sequential version of the code since the functions used in both languages retain the same structure as the sequential counterparts with only few considerations and, possibly, changes needed in order to make the problem parallelisable.

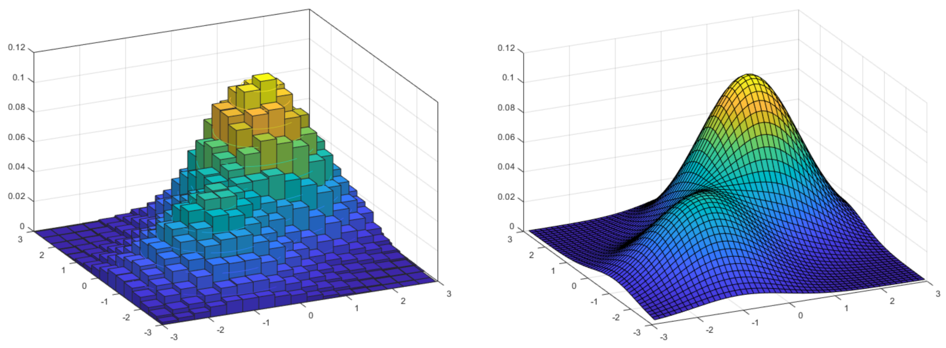


Figure 4.6: Result obtained by the M.-H. algorithm in the simulation of the target distribution (represented in the right graph for reference). The histogram represents the density that the points generated by the M.-H. simulation form.

R			MATLAB		
Cores	MH time	Speedup	Cores	MH time	Speedup
1	114.36	1.00	1	111.35	1.00
2	57.71	1.98	2	54.12	2.06
3	38.93	2.94	3	36.58	3.04
4	31.02	3.69	4	27.06	4.11
5	26.92	4.25	5	21.81	5.10
6	24.14	4.74	6	18.45	6.04
7	21.72	5.27	7	15.65	7.12
8	20.26	5.65	8	13.81	8.06
9	18.90	6.05	9	12.45	8.94
10	17.80	6.43	10	11.21	9.94
11	17.03	6.71	11	10.34	10.77
12	15.84	7.22	12	9.56	11.65
13	15.04	7.60	13	8.81	12.64
14	14.33	7.98	14	8.37	13.31
15	13.73	8.33	15	7.87	14.14
16	12.90	8.86	16	7.52	14.80
17	12.63	9.05	17	7.18	15.50
18	12.20	9.37	18	6.82	16.32
19	12.33	9.27	19	6.65	16.73
20	11.57	9.89	20	6.39	17.41
21	11.98	9.54	21	6.14	18.15
22	12.23	9.35	22	6.17	18.04
23	11.65	9.82	23	6.11	18.21
24	12.13	9.43	24	6.06	18.37

Table 4.1: Detailed results of the scale-up study of the M.-H. algorithm in seconds.

Conclusion

Aimed at the exploration of a possible approach to parallel computing in programming, with the objective of representing an introduction of parallel computing and a guide for further practical implementations, this thesis has explored its potential, firstly from a theoretical standpoint and then going through the practical applications, in both MATLAB and R.

Parallel computing offers an alternative way to increase computational power of computers, nowadays more and more required for modern applications. From the huge quantities of data to analyse in complex statistical models to the many artificial intelligence applications that are becoming more and more popular and accessible, technological improvements in hardware components has not been able to keep-up with the raising computing power demand.

The thesis takes the knowledge of the parallel computing paradigm and studies its application on the Metropolis-Hastings algorithm, an optimal case due to its computational requirements that may determine long computation times. The algorithm also required a rearrangement to be made parallelisable. In the original form it does not meet, in fact, the requirement of having various sub-tasks independent from each other. Thus, it is needed a small rework in order to make it parallelisable, namely by breaking the Markov chains into many separate and independent chains. By running the algorithm in parallel, significant reduction in the computation time has been achieved. More specifically, MATLAB seems to be more optimized to work in parallel and takes greater advantage from it, obtaining a greater reduction in the overall computation time. On the other hand R, although achieving a sensible reduction, with the increase of the number of cores employed, its efficiency reduces at a faster pace compared to MATLAB.

Furthermore, MATLAB has proved to be more easy to set-up for parallel computing, as long as the Parallel Computing Toolbox is installed. The functions themselves are more similar to the usual, sequential, way of programming for simpler constructs, such as `for` loops. R requires the installation of few libraries and, even if with a few extra steps, it is a process that has been made fairly easy, not too different from the sequential programming constructs.

In conclusion, the aim of this thesis is the one to evaluate the implementation of parallel computing in the workflow of computer simulation, in the simplest way, achieving the translation of sequential programs into parallel ones, reducing the effort required as much as possible. Of course doing so is expensive in terms of knowledge needed and programmer's efforts of modifying the program, making it parallel and/or dealing with possible new bugs and errors in the code that arise during the process. This is, in fact, a way of thinking about programming that is inherently different from what non-experts in the subject are used to. Nevertheless, in recent years a lot of effort has gone into making parallel computing easier to approach, both from MathWorks in the development of MATLAB and the open-source community developing libraries for R. In conclusion, in light of the results achieved by the experiments performed in this thesis, it is reasonable to state that parallelisation has become a feasible solution which can be able to provide improvements that are worth the extra effort.

Bibliography

Abdel-Rehim, Wael M.F. et al. (2015). "Testing Randomness: The Original Poker Approach Acceleration Using Parallel MATLAB". In: *Journal of Computer Science and Applications*, pp. 52–57.

Adve, V. Sarita et al. (2008). *Parallel Computing Research at Illinois. The UPCRC Agenda.* University of Illinois at Urbana-Champaign.

Amdahl, Gene M. (1967). "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference.* Association for Computing Machinery.

Anantha, P. Chandrakasan, Sheng Samuel, and W. Brodersen Robert (1992). "Low-Power CMOS Digital Design". In: *IEEE Journal of Solid-State Circuits.*

Asanović, Krste et al. (2006). *The Landscape of Parallel Computing Research: A View from Berkeley.* Tech. rep. EECS Department, University of California, Berkeley.

Backus, John (1978). "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs". In: *ACM Turing Award Lectures.* New York, NY, USA: Association for Computing Machinery.

Biondini, Gino (2015). "An Introduction to Rare Event Simulation and Importance Sampling". In: *Big Data Analytics.* Ed. by Venu Govindaraju, Vijay V. Raghavan, and C.R. Rao. Vol. 33. Handbook of Statistics. Elsevier.

Catalant Staff (2015). "The Slowing of Moore's Law, and What It Really Means for Computing". In: *gocatalant.com.* Accessed on: 12/05/2020. URL: `https://gocatalant.com/blog/the-slowing-of-moores-law-and-what-it-really-means-for-computing/`.

Gustafson, John L. (1988). "Reevaluating Amdahl's Law". In: *Commun ACM* 31.5.

Hastings, W. K. (Apr. 1970). "Monte Carlo sampling methods using Markov chains and their applications". In: *Biometrika* 57.1.

Kendal, M. G. and B. Babington Smith (1938). "Randomness and Random Sampling Numbers". In: *Journal of the Royal Statistical Society.*

Lehmer, Derrick H. (1951). “Mathematical Methods in Large-scale Computing Units”. In: *Proceedings of the Second Symposium on Large Scale Digital Computing Machinery*. Cambridge, United Kingdom: Harvard University Press.

MathWorks (2020). *Parallel Computing Toolbox™ User's Guide*. The MathWorks, Inc.

McCool, M., J. Reinders, and A. Robison (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science.

Metropolis, N. et al. (1953). “Equations of state calculations by fast computing machine”. In: *Journal of Chemical Physics* 21.

Microsoft and Steve Weston (2019a). *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*. R package version 1.0.15.

Microsoft and Steve Weston (2019b). *foreach: Provides Foreach Looping Construct*. R package version 1.4.7.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria.

Rauber, Thomas and Gudula Rünger (2013). *Parallel programming for multicore and cluster systems*. Second. Springer.

Robert, P. Christian (2016). “The Metropolis Hastings algorithm”. In: ArXiv preprint arXiv:1504.01896.

Robert, P. Christian and George Casella (2009). *Introducing Monte Carlo Methods with R*. Springer-Verlag.

Roberts, G. O., A. Gelman, and W. R. Gilks (1997). “Weak convergence and optimal scaling of random walk Metropolis algorithms”. In: *Ann. Appl. Probab.* 7.1.

Rossini, Anthony, Luke Tierney, and Na Li (2003). “Simple Parallel Statistical Computing in R”. In: *UW Biostatistics Working Paper Series*.

Rubinstein, R. Y., B. Melamed, and A. Shapiro (1998). *Modern Simulation and Modeling*. A Wiley-Interscience publication. Wiley.

Rubinstein, Reuven Y. and Dirk P. Kroese (2017). *Simulation and the Monte Carlo Method*. John Wiley & Sons, Inc.

Vadhan, Salil P. (2012). “Pseudorandomness”. In: *Foundations and Trends in Theoretical Computer Science* 7.

Whitt, Ward (Nov. 1976). “Bivariate Distributions with Given Marginals”. In: *Ann. Statist.* 4.6.