UNIVERSITY OF MONTENEGRO

FACULTY OF ELECTRICAL ENGINEERING

PODGORICA, MONTENEGRO

SIDITA DULI

AN APPROACH FOR THE PARALLELIZATION OF THE WEIBULL DISTRIBUTION PARAMETERS ESTIMATORS WITH APPLICATIONS

- Doctoral Thesis -

Podgorica, June 2018

INFORMATION ABOUT PHD CANDIDATE

• First name / Surname	Sidita Duli
• Date of birth and birthplace	22.08.1983, Tiranë
• The title of the completed postgraduate study program and the year of completion.	Master in Computer Science, 2008

INFORMATION ABOUT DOCTORAL DISSERTATION

• Title of doctoral studies	Doctoral studies
• Title of doctoral thesis	An approach for the parallelization of the Weibull distribution parameters estimators with applications
• Faculty where the dissertation is defended	Faculty of Electrical Engineering

INFORMATION ABOUT DEFENSE OF DOCTORAL THESIS

• Date of doctoral dissertation	18.12.2017
• Date of session of the Senate of the University of Montenegro where the dissertation is accepted	06.06.2018
• Commission for the evaluation of the topic and eligibility of doctoral dissertation	Prof. Dr. Božo Krstajić Prof. Dr. Igor Djurovič Prof. Dr. Miloš Daković
• Supervisor	Prof. Dr. Božo Krstajić
• Commission for the evaluation of the doctoral dissertation	Prof. Dr. Milutin Radonjić Prof. Dr. Božo Krstajić Doc. Dr. Slavko Gajin
• Commission for the PhD thesis defense	Prof. Dr. Milutin Radonjić Doc. Dr. Slavko Gajin Prof. Dr. Božo Krstajić Prof. Dr. David Kaljaj Prof. Dr. Slobodan Djukanović
• Day of the defense:	15.06.2018

To my parents, Bahri Duli and Shpresa Duli

ABSTRACT

A distributed system is a collection of independent computers that appears to its users as a single coherent system. This definition has several important aspects. The first one is that it consists of autonomous computers. A second aspect is that users think they are dealing with a single system. This means that, one way or the other; the autonomous components need to collaborate. How to establish this collaboration lies at the heart of developing distributed systems.

One of the benefits of parallel computing is processing a statistical problem in less time, thus leading to a faster result. Many multidisciplinary scientific fields, such as bioinformatics, biochemistry, electrical engineering and physics, use parallel computing and distributed computing resources for simulations of experiments. The primary focus of many researches in the area of distributed computer scheduling is finding a way to distribute tasks among the CPU cores in order to achieve better performance, such as minimizing job execution time, minimizing communication and maximizing resource utilization. Complex tasks are numerical modeling or numerical weather prediction and the specific work is to analyze the data collected from the weather stations. The most important and intensive task is the wind analysis, with its two parameters, the speed and the direction.

Different tests should be performed to determine the Weibull parameters for a particular location. In this research, the specific locations are two cities, Podgorica and Shkodër. A parallel version for estimating the Weibull parameters could be necessary to improve the quality of performing the calculations by processing them faster. It is important to compare and analyze different methods of parallelization, in order to find the adequate one which executes faster the specific algorithm.

A parallel version is built in C by using the MPI library, which implements the parallelization using the message passing architecture. Another implementation of parallel version is by using the concept of threads implemented in shared memory architecture. A third implementation in parallel of estimation of Weibull parameters uses the OpenMP directives to synchronize tasks split to different threads. A forth version is the hybrid MPI/OpenMP version. The main reason this version is implemented as the last one, is the need to compare a hybrid mode with the pure MPI implementation.

The main aim of this thesis is to analyze and compare the methods that enable the parallelization of the Weibull parameter estimation. This analysis leads to an improved version of the Weibull parameter estimation, by using the most efficient method among those compared. Also, these methods are compared with the serial version of the Weibull distribution parameter estimator, by analyzing the speed-up and the time spent in calculations.

Acknowledgments

First and most important, I would like to thank Prof. Dr. Božo Krstajić, professor at University of Montenegro, my supervisor, for guiding my scientific development and for his ideas regarding this dissertation.

Additional thanks to the employees of Hydro meteorological Institute of Montenegro, also the Institute of Geo-sciences, Energy, Water and Environment in Tirana and the team of Technical and Meteorological station of University of Shkodra, for having provided the wind speed data used in this research, and very necessary to complete this dissertation.

I would like to thank Elvis Tavasja, for being so supportive while I was working on my thesis. Thanks for all your constructive suggestions.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment is dedicated to my parents. Thank you.

Table of Contents

Table of figures

Index of Tables

Chapter 1

Introduction

In our sophisticated modern world, time is perhaps the most precious commodity. We live our lives in the fast lane, always trying to buy more time. In this world, speed is essential and it is translated naturally into how fast we can solve the problems we face. To gain time, parallel computing is perhaps our greatest ally [1].

A distributed system is a collection of independent computers that appears to its users as a single coherent system. This definition has several important aspects. The first one is that it consists of autonomous computers. A second aspect is that users think they are dealing with a single system. This means that, one way or another, the autonomous components need to collaborate. How to establish this collaboration lies at the heart of developing distributed systems [1] [2].

One important characteristic of distributed system is that differences between the various computers and the ways in which they communicate are mostly hidden from users. Another important feature is that users and applications can interact with a distributed system in a consistent and uniform way, regardless of where and when interaction takes place [3]. A distributed system should make resources easily accessible; it should reasonably hide the fact that resources are distributed across a network; it should be open; and it should be scalable. A distributed system that is able to present itself to users and applications as if it were only a single computer system is defined as transparent [2] [4].

The main motivation for parallel computing is to speed up computation. The pervasive nature of computers nowadays makes it possible for huge amounts of data to be acquired and stored in large databases for future analysis, data mining, referencing, etc. In some cases, the amount of information that needs to be processed is so huge, that the time required to complete the job becomes prohibitively long [5] [6]. But time is not the only measure for the advantage gained by using a parallel approach. Sometimes, it is the quality of the solution computed that is greatly improved if more processors are available, in a fixed amount of time. Furthermore, computational scenarios have been identified, in which the only chance to terminate a computation and reach a solution is to have the

required number of processors working simultaneously on that respective task. We call such problems inherently parallel, because the ability of a parallel computer to be “in more than one place at a time” through its multiple processing elements is a necessary condition to successfully tackle these problems [1] [7].

Parallel computing requires that the program is segmented, divided into sections that can run simultaneously, and the division of the program takes into account the different environments on which the different sections of the program will be running. Organizing the interaction between each segment is of utmost importance [8]. To reach the widest possible range and types of computers, the protocol or communication channel should not contain or use any information that may not be understood by certain machines. The messages should be delivered correctly. The invalid messages should be rejected; otherwise they will bring down the system and perhaps the rest of the network [9].

Why parallel computing? The answer is simple: because there exists a need to solve large problems, which often arise, among others, in the application of statistical methods to large data sets [9].

One of the benefits of parallel computing is processing a statistical problem in less time, thus leading to a faster result [10]. Many multidisciplinary scientific fields, such as bioinformatics, biochemistry, electrical engineering and physics, use parallel computing and distributed computing resources for simulations of experiments. The primary focus of many researches in the area of distributed computer scheduling is finding a way to distribute tasks among the CPU cores in order to achieve better performance, such as minimizing job execution time, minimizing communication and maximizing resource utilization. Complex tasks are numerical modeling or numerical weather prediction and the specific work is to analyze the data collected from the weather stations. The most important and intensive task is the wind analysis, with its two parameters, the speed and the direction.

Among various renewable energy resources, wind power energy is presently one of the most popular and most promising one. At a specific wind farm, the available electricity generated by a wind power generation system depends on the mean wind speed and the standard deviation of mean speed. The wind speed variations during a year can be characterized in terms of a probability distribution function [11]. It is a matter of common

observation that the wind is not steady. In order to calculate the mean power delivered by a wind turbine from its power curve, it is necessary to know the probability density distribution of the wind speed. This is the distribution of the proportion of time spent by the wind within narrow bands of wind speed. The tool used to calculate the wind speed is the Weibull distribution. The reason why the Weibull Distribution would be important in calculating wind speeds is that eventually the wind speeds are going to die down. Therefore, this is something that is decaying or vanishing. Hence, a Weibull distribution for the wind speed distribution is going to be very close to the overall Weibull shape [6].

The most general form of the Weibull is the three-parameter Weibull distribution expression. It includes the location parameter, the scale parameter and the shape parameter. Frequently, the location parameter is not used. The value for this parameter can be set to zero. In this way the Weibull distribution is represented as the two-parameter form of Weibull distribution. There is also a form of the Weibull distribution known as the one-parameter Weibull distribution.

There are different methods for estimating the Weibull parameters. Choosing the right algorithm is an important issue of the whole process of parallel design and implementation [12] [13] [14] [15].

In this research, the maximum likelihood method is implemented. According to [16], the method of moments is the best one, taking into consideration the total deviation as criteria for comparison. But, as concluded in [16] and [17], the maximum likelihood method can be a close substitute of method of moments in estimation of the two-parameter Weibull distribution.

Different tests should be performed to determine the Weibull parameters for a particular location. In this research, the specific locations are two cities, Podgorica and Shkodër. A parallel version for estimating the Weibull parameters could be necessary to improve the quality of performing the calculations by processing them faster. It is important to compare and analyze different methods of parallelization, in order to find the adequate one which executes faster the specific algorithm.

A parallel version is built in C by using the MPI library, which implements the parallelization using the message passing architecture [18] [19].

Another implementation of parallel version is by using the concept of threads implemented in shared memory architecture [20] [21].

A third implementation in parallel of estimation of Weibull parameters uses the OpenMP directives to synchronize tasks split to different threads [22] [23].

A forth version is the hybrid MPI/OpenMP version. The main reason this version is implemented as the last one, is the need to compare a hybrid mode with the pure MPI implementation [24].

The performance of these versions is compared with each other, as well as with the serial version. What is more important, the parallel applications that are built can be used in all the related fields that apply the estimations of the Weibull distribution parameters. The time needed to perform the calculations is reduced depending on the number of cores and the architecture used in the specific case [25] [26].

This research used one serial algorithm might extend by adding another serial algorithm of Weibull parameter estimation and by analyzing the interference of the algorithm in the speed-up metric.

Parallel programs which estimate the Weibull parameters are tested in a distributed environment. The process of adding parallel code in a serial version leads to a faster result of calculating complex mathematical formulas.

These four parallel versions are an improvement of an existing serial program. They contain extra parallel code that makes it possible to be implemented in a distributed system. In this way, each of them reduces the time to calculate the parameters for different set of data. This reduction of time is not the same for all the methods used in parallelization. The methods are compared with each other, based on two parameters: time and speed-up.

The main aim of this thesis is to analyze and compare the methods that enable the parallelization of the Weibull parameter estimation. This analysis leads to an improved version of the Weibull parameter estimation, by using the most efficient method among those compared. Also, these methods are compared with the serial version of the Weibull distribution parameter estimator, by analyzing the speed-up and the time spent in calculations.

Note that, the reduction of time depends on certain factors, such as number of processors involved in the tests, network bandwidth, etc. The parallel applications also calculate the mean, the average, the mode, and other statistical parameters for a set of data.

For the specific location, it might be interesting to compare the Weibull parameters from Podgorica and Shkodër. Tests can be performed with wind data for the whole interval of time, the whole year or some other periods of time. The application will help all environmental engineers in their studies of comparison of the weather elements in neighboring locations. For these specific locations, there are only a few comparative studies. The weather stations already exist, but the way of analyzing these data should be improved. By introducing the parallel computations, the data collected by the weather stations will be processed faster. This is what this research proposes: by running a parallel version, the parameters of Weibull distribution will be estimated faster than running the serial implementation. Also, this research compares the methods of the parallel programming.

A hypothesis is: the methods that implement the parallelism using the parallel threads, will achieve a better speed-up than the methods that use the parallel processes. Implementing the parallelism by using parallel threads should be a flexible method.

The thesis is organized as follows. The introduction presents an overview of the research, objectives and goals. It gives the key concept of parallel computing, and introduces the main objectives of this research.

The second chapter brings a brief description of architecture of distributed systems. In focus of this chapter are the Grid parallelization, shared memory architecture and the message passing architecture. Also it describes different levels of parallelism.

The next chapter gives a mathematical overview of Weibull distribution, it described in its three forms. It shows the meaning of three parameters of Weibull. Also, it treats the role of the Weibull distribution in wind speed analysis.

The forth chapter extends in details the parallel programming models and environments. It is focused on describing three main models: the message passing interface, the threads programming and the OpenMP.

The fifth chapter contains data from the tests performed on the implementation of MPI version of Weibull distribution parameter estimation. It concludes with the results of the speed-up and efficiency of MPI version of the application.

The next chapter contains discussions of the tests performed on implementation of Pthread version of Weibull distribution parameter estimation. These discussions are based on speed-up and efficiency of Pthread version of the program.

The seventh chapter includes details of the implementation of OpenMP version of Weibull distribution parameter estimation. This chapter contains discussions and results related to the performance of this parallel version.

The eighth chapter analyses a possible hybrid implementation of MPI/OpenMP version of Weibull distribution parameter estimation. This chapter contains data from the tests performed in this research, including speed-up data and efficiency of this hybrid version.

The next chapter points out to final conclusions, based on results of performance from four methods of parallelization used in this research.

Chapter 2

Architecture of distributed systems

2.1 Computer categorization

In 1966 Flynn categorized the computers' taxonomy. It is referenced to Flynn's taxonomy and it is one of the most known categorization in the computer architecture. The categorization is based on the concept of the stream of information. There are two types of information: instructions and data. The instruction stream is defined as the sequence of instructions performed by the processing unit. The data stream is defined as the data traffic exchanged between the memory and the processing unit. According to Flynn's classification, either of the instruction or data streams can be single or multiple.
This taxonomy classifies each computer in one of following categories [8]:

- SISD- single instruction-single data, which includes most of the Von Neumann type computers.
- MISD-multiple instruction, single data, which describes various special computers, but not any particular class of machines
- SIMD-single instruction, multiple data, the architecture of parallel processor arrays
- MIMD-multiple instruction, multiple data, which includes most modern parallel computers

Single processor computers, that adapt the Von Neumann model, are classified in the SISD category. No well-known systems fit in the MISD designation. It is mentioned for the sake of completeness. No commercial multiprocessor of this type has been built to date [4].

Meanwhile the parallel systems belong to SIMD or MIMD. In cases when there is only one processing unit and all the processors execute the same instruction in a synchrony way, this system is classified as SIMD [7] [8].

2.1.1 SIMD

Computer systems that are part of this category contain several processors. Each processor is connected to its own memory, and they all are connected with a management unit. The management unit distributes the work between processors. Each processor executes the same instruction but with different data. When all processor finish the same calculation, results are collected in the management unit. A main advantage of these computing systems is the synchronic processing of the data. A disadvantage is its cost [8].

2.1.2 MIMD

Computer systems that are part of this category fall into two broad categories: shared memory or message passing [8].

2.1.2.1 Shared memory organization

Shared memory computers contain several processors that share the same memory. The processor and the memory are connected by some network switches. Times ago, these types of computers included just several processors, without memory. The cache memory was later included. In this architecture there are some limitations. One of them is the fact it is not possible to make this architecture scalable with more than 32 processors, without causing bandwidth collisions between processors and global memory [8].

2.1.2.2 Message passing organization

The main purpose of distributed memory parallelization is to create independent modules that can execute in parallel different tasks that might be related to each other. The modules can communicate to each other in different ways, for example by message passing. This way of communication can cause system latency or message delays. In such systems, the main goal of this way of communication is to reduce to a minimum the number of messages sent among modules. When the tasks distributed in different modules are not related to each other, the communication is minimized. But this rarely happens in practice. Often, the modules run different tasks. This is called coarse level parallelization. It is the case when the programmer divides the tasks to distribute to the modules. Such a decision is based on a semantic analysis of the program’s algorithm. One of issues in this case is to

put the modules' work in balance. Each module should run a task with a complexity close to the other modules' tasks. This minimizes the possibility that one module rests while the others are running their tasks [8].

A hybrid version of shared memory and distributed memory might be: shared modules in a distributed configuration. This hybrid form is called distributed shared memory computer. This hybrid system behaves more like a distributed system [7].

2.2 Grid parallelization

Computer Grid is a computer system, homogeneous or heterogeneous, in which the components are connected with a network. These components might be different in architecture. The system might run parallel algorithms which include different tasks that will be run in different units. Whenever the task is finished, it is guaranteed the accuracy of the result [1].

Grid computing gathers resources into one, secure, and quality assuring environment. It represents a significant achievement toward the aggregation of clusters and/or other networked computers for solving large-scale data-intensive or calculation-intensive problems [1]. Grids can be classified into several categories, which vary in target application domain and purpose [1]:

- Computational grids provide distributed computing facilities for executing compute-intensive applications. The role of the Grid is to schedule these applications on available resources.
- Data grids provide the infrastructure to access, transfer, and manage large data sets stored in distributed repositories. They focus on satisfying requirements of scientific collaborations, where there is a need for analyzing large collections of data and sharing the results.
- Application Service Provisioning (ASP) grids provide access to remote applications, modules, and libraries hosted on data centers or computational grids.

- Collaborative grids provide services and platforms for users to interact with each other in a real-time environment. This type of Grid is suitable for multimedia applications, such as video conferencing, and those that require fast networks.
- Knowledge grids work on knowledge acquisition, data processing, and data management. Also, they provide business analytics services driven by integrated data mining services.
- Utility grids focus on providing one or more of the above grid services to end-users as information technology (IT) utilities on a pay-to-access basis. They set up a framework for the negotiation and establishment of contracts, and allocation of resources based on user demands.

It is important to make a comparison between grid computing and cluster systems. The first difference is the expected size of the system. A cluster has a typical size which is usually measured in at most hundreds of nodes. A grid system mostly has a size of thousand of nodes. For this reason, cluster system is a choice for a scalable design of services.

A second difference is the administration. The nodes of a cluster are owned and administered by a single organization, whereas the nodes of a grid can have different owner and different policies of resource management. A third difference is the connection between nodes. Cluster’s nodes are connected by dedicated high-speed networks, whereas grids use public internet [1]. Also, clusters are organized as, in a group of computers connected by a local area network (LAN), whereas grid are more wide scale and can be geographically distributed. A cluster is tightly coupled, whereas a grid is loosely coupled. Clusters are often an operational part of a grid.

2.3 Different levels of parallelism

2.3.1 Instruction level parallelism

The first instruction level parallelism is introduced early, in the 1960s, with the first pipelined processors. Later, this technique was the key to achieve performance improvements [4].

The goal is to create parallelism in between the instructions. This happens only if the instructions are independent of each other. It should be clear the specification when an instruction is defined independent of another instruction. This fact also helps in deciding how to implement this parallelization.

During creation of instruction level parallelism, the programmer should specify which instruction will be run in parallel. If there are two such parallel instructions, they can be run in any pipeline. If two instructions are dependent, they cannot be run in parallel. They will be run in a specific sequence.

There exist three types of dependence: data dependence, name dependence and control dependence [4].

2.3.2 Data dependence

An instruction *j* is defined as dependent to an instruction *i*, if it fulfills one of the conditions below:

-instruction *i* produces a result that will be used from instruction *j*

-instruction *j* is data dependent to instruction *k*, and instruction *k* is dependent to instruction *i* [4].

If two instructions are data dependent, they cannot be run in the same time.

2.3.3 Name dependence

Name dependence occurs when two or more instructions use the same register or memory location, defined as name, but there is no flow of data between the instructions associated with that name.

There are two types of name dependence between instruction *i* and instruction *j*:
-the anti-dependence between instruction *i* and instruction *j* occurs when instruction *j* writes in a register that instruction *i* reads. If these two instructions are run in parallel, then it might happen that instruction *i* is not reading the correct values of register.
-the output dependence between instruction *i* and instruction *j* occurs when both instructions write in the same register or memory allocation. If these two instructions are executed in parallel, the last updated value of the register (or memory address) might be the value of instruction *i* or the value of instruction *j*.

2.3.4 Control dependence

Each instruction in the same block has dependence to other instructions of the same block. An example in this case is the 'if' instruction, which has dependence to other instructions of the block.

Nowadays processors have some pipelines for operations with integers, and for operations with floats. Let's introduce two different types of parallelization
-related to depth: if a pipeline has depth *n*, then it is possible to execute up to *n* instructions in the same time.
-related to number: if there are *i* pipelines that do calculations with integers, each of them in *n* depth, and *j* pipelines that do calculations with float, each of them in *m* depth, and if all these pipelines are running, then for each loop there will be $n * i + j * m$ calculations.

This level of parallelization is called micro-parallelism or instruction level parallelism. It depends on the machine language operations. This parallelism is supported from the logic unit of a processor and from the compiler. The user does not deal with this level of parallelism [4].

2.3.5 Thread level parallelism

The higher-level parallelism is called thread-level parallelism (TLP) because it is logically structured as separate threads of execution. In thread-level parallelism, the multiple control flows needed are called threads and the parallelism is represented by the use of multiple threads of execution that are inherently parallel. Each thread has all the state (instructions,

data, PC, register state, and so on) necessary to allow it to execute. Unlike instruction-level parallelism, which exploits implicit parallel operations within a loop or straight-line code segment, thread-level parallelism is explicitly represented by the use of multiple threads of execution that are inherently parallel. Thread-level parallelism is an important alternative to instruction-level parallelism, primarily because it could be more cost-effective to exploit than the instruction-level parallelism. There are many important applications where thread-level parallelism occurs naturally, as it does in many server applications.

Different programming environments are the field of study in [27], [28], [29], [30], [31]. The study [28] compares the parallel programming models for multi-core architecture. It provides a comparison of new and the most popular software technologies that hide the complexity of multi-core architecture, allowing programmers to focus on algorithm instead on architecture.

The parameters that may serve as a base to compare one model with another are:

- Ease of programming: A subjective term which influences an application developer's choice of programming model.
- Availability: the availability of a programming model on a given approach is a crucial aspect in the widespread use of the programming model.
- Overhead: The overhead associated with parallelization, such as thread creation, termination and management, is calculated as the amount of time it takes the parallel code to run.
- Control and flexibility: the features provided by a programming model influence the effectiveness of the algorithm.
- Types of parallelization: Different programming models are more or less suited for a certain kind of parallelism.
- Scalability: With the increasing number of cores, the scalability of a given application is a major factor while designing the parallel application.

2.4. The performance of parallelization

The processing speed of computers involved in scientific calculations is usually expressed in terms of a number of floating-point operations completed per second, and it is a measure used to describe the computational power of the world's largest supercomputers. For a long time, the basic measure was Mflops expressed as:

$$r = \frac{N}{t} Mflops \qquad (2.1)$$

where N represents a number of floating-point operations executed in t microseconds.

One of the best methods to measure the performance of parallelization of a specific algorithm is to analyze how this algorithm can be divided in parts that will run in parallel, and in other parts that will run in a sequential manner. A good performance is achieved if all possible parallel parts of the algorithm are identified. This analyzing method is mostly developed as contribution of Amdahl. He established the role of the slower parts of an algorithm in its overall performance. Assuming that a given program consists of N floating-point operations, out of which a fraction f is executed with a speed of V Mflops, while the remaining part of the algorithm is executed with a speed of S Mflops, and assuming further that the speed V is close to the peak performance while the speed S is substantially slower ($V>>S$), then the total execution time can be then expressed by using the following formula:

$$t = f\frac{N}{V} + (1-f)\frac{N}{S} = N\left(\frac{f}{V} + \frac{1-f}{S}\right) \quad . \qquad (2.2)$$

From this, it follows that

$$t > \frac{(1-f)N}{S} \quad . \qquad (2.3)$$

If the whole program is executed at the slower speed S, its execution time is calculated as:

$$t_s = \frac{N}{S} \quad . \qquad (2.4)$$

If the execution speed of the part f of the program can be increased to V, then the performance gain can be calculated as:

$$\frac{t_s}{t_v} < \frac{N}{S} \cdot \frac{S}{N(1-f)} = \frac{1}{1-f} \tag{2.5}$$

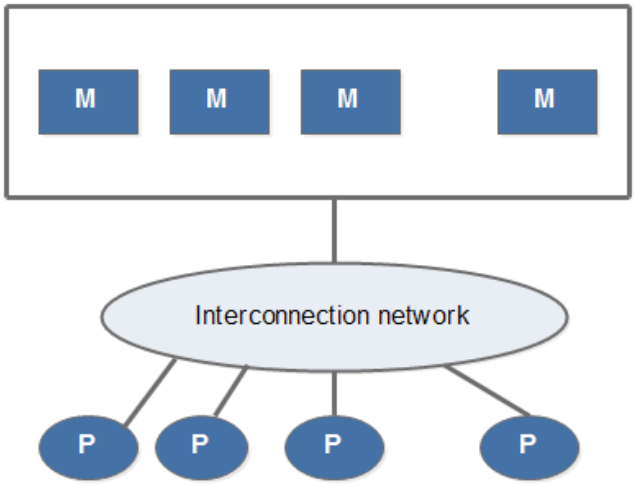
This formula represents the Amdahl's Law and can be interpreted as follows: the speed-up of an algorithm that results from increasing the speed of its fraction *f* is inversely proportional to the size of the fraction that has to be executed at the slower speed. In practice, this law provides an estimate of the overall speed at which the algorithm can be executed [10].

In a parallel system, the total time of execution is calculated as the sum of the times of execution of each process. This total time is larger than the time that a single processor would run the serial version of the algorithm. The goal of the parallel programming is to decrease the time that is needed to get a result. If the algorithm would be divided in *p* parts which can be executed simultaneously in *p* processors, then the ideal case would produce a result in 1/*p* of the time of execution of the serial version. This ideal case is practically impossible. The Amdahl's law shows that the algorithm has several parts that should be executed in serial. These parts decrease the overall performance of the parallel version, making it not ideal anymore [10].

2.5 Shared memory architecture

A major part of multiprocessor systems are shared memory system. This kind of system has processors that share the same global memory. Different tasks that are run in different processors can communicate with each other by reading or writing in this same global memory. Figure 2.1 is an illustration of a shared memory system that contains some processors; it shows *P* nodes, some shared memory modules *M* and an interconnection network. Nodes are connected through a interconnection network with processors and the memory.

Figure 2.1: Shared memory schema



In this system, each process can access the memory by sending an access request. Inside memory, there is an arbitration unit which redirects these requests to the memory controller. If a memory module is free, any request sent for this module will be accepted. After this, the module will be in a busy state. If a module is busy, while the request is sent, he sends a wait signal, through the controller, to the processor that sent the request. As a response to this signal, the processor can wait for this request, until the memory module will be free, or can send a new request. If the arbitration unit receives two different requests in the same time, it will choose one of them, and will send it to the controller. It will send a wait signal to the other request, which means the other processor will wait or will resend a new request [8].

Based on the interconnection used, shared memory can be categorized in:

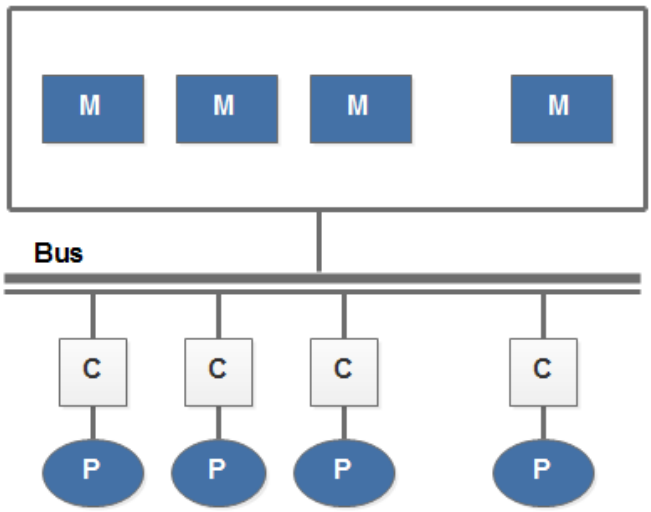
- uniform memory access (UMA)
- non-uniform memory access (NUMA)
- cache -only memory architecture (COMA)

The main difference between the NUMA and UMA memory architecture is the location of the memory. The UMA architecture nodes have first and second cache memory levels joint with the processor, next levels of the memory hierarchy are "in the other side" of the interconnection network [8].

2.5.1 UMA

Members of this category have processors that have the same access memory time. The interconnection network in these cases may be a single bus, multiple bus or crossbar switch. These systems are called also as SMP (symmetric multi-processor) systems.

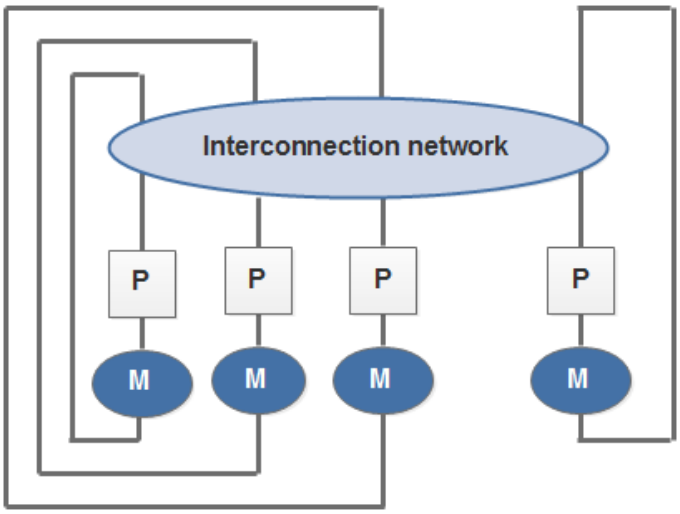
Figure 2.2: SMP schema



2.5.2 NUMA

In NUMA systems, each processor has attached a part of memory. The whole memory has a single address space. Each processor can access any memory location. This happens by using the real unique address of the memory location. The access time of any memory location depends on the processor's distance. This non uniform time of access is the reason of the name of system, non uniform memory access time.

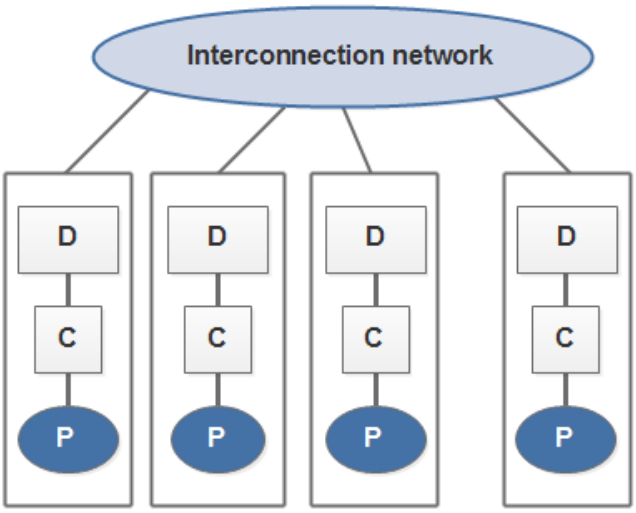
Figure 2.3: NUMA schema



2.5.3 COMA

Members of this category are similar to the NUMA's category, where processors have a module memory. In the COMA systems, this memory is a cache. If any processor needs some data that are located in a memory module, then these data will be transferred to the processor that is connected to this memory module. There is no memory hierarchy. The address space consists of all the caches.

Figure 3.4: COMA schema



2.6 Message passing architecture

A message passing system consists of a local memory and a processor in each node of interconnection network. In this system, there is no shared or global memory. Each node communicates with other nodes by message passing. These messages transfer data from one local memory to another. They are sent by the instructions written in the code [8].

Figure 2.5: Message passing architecture

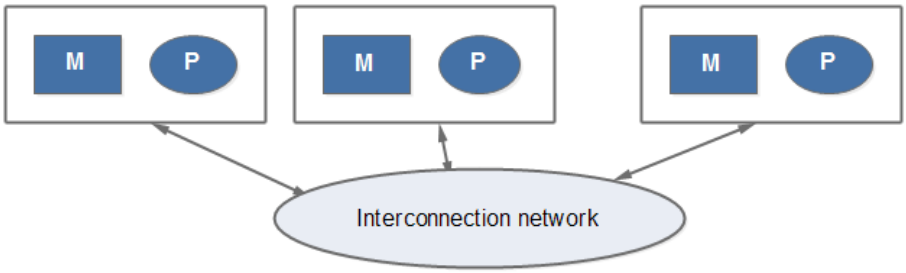


Figure 2.5 shows the components of a message passing system. It contains *N* nodes that contain one processor *P* and one local memory *M*. Each processor has its address space. The nodes communicate to each other through interconnection network.

During the execution, the tasks are divided in concurrent parts. These parts are executed in different processors. If the number of tasks is bigger than the number of processors, then one processor will execute more than one task. Tasks that are executed in the same processor exchange data among them by using the same internal channels.

Data exchanged cannot be shared; they are copied using send and receive instructions. An important advantage of this form of data exchange is the elimination of the need for synchronization constructs, such as semaphores, which results in performance improvement. In addition, a message passing scheme offers flexibility in accommodating a large number of processors in addition to being readily scalable. It should be noted that a given node can execute more than one process, each at a given time [8].

Figure 2.6: Message passing example

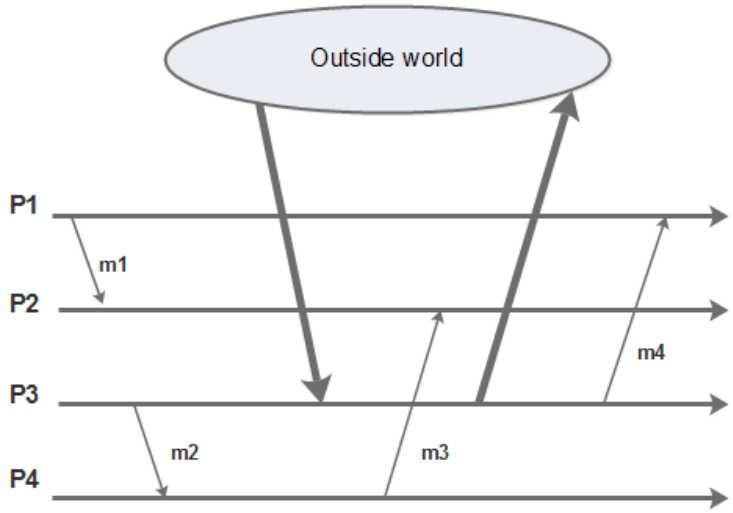


Figure 2.6 shows an example of a message passing system with four processes. In the figure, the horizontal lines represent the execution of each process, *P1* to *P4*. The lines between processes represent a message that is exchanged among these two processes. For example, the message shown as *m3*, is a message passed from process *P4* to the process

P2. In the model shown in figure, a message is a logical unit of the communication among nodes. It is a set of related information that is defined as an entity. A message can be data, instruction, and synchronization or interrupt signal. In the message passing model, a message has a role of interaction with the outside world of the system.

Routing is defined as the techniques used for a message to select a path over the network channels [8].

There are two ways of communication operations in message passing systems. The first one is point-to-point communication. The second is collective communications. In point to point communication, a node can communicate a message to only one destination, which may be one of its neighbors. The techniques used for a message to select a path over the network channels are defined as routing. Two important routing operations are broadcast and multi-cast.

Broadcast is known also as one-to-all operation, because one node sends the same message to all other nodes. It is used to distribute data from one node to others.
Multi-cast is known also as the one-to-many operation, because one node sends a message to n distinct nodes. Multi-cast is used in large-scale multiprocessors and it is appropriate for parallel search algorithms.

The aim of broadcast and multi-cast routing algorithms is the message passing in destination in as little time and using the shortest path possible. One way to manage this is by delivering the message to a common path, to multiple destinations, and then forwarding the message to unique set of nodes. In this way, the message will be delivered in every node. This is a tree-based communication model. Another way is to manage the multi-cast operation is to use separate addressing. A separate copy will be sent to every destination, but this is an inefficient technique [8].

In a comparative view of these models, broadcast is mainly used to distribute data from one node to others during computation of a distributed memory program. Multi-cast has several uses in large-scale multiprocessors, including parallel search algorithms and single program multiple data computation. Practical broadcast and multi-cast routing algorithms must be deadlock-free and should transmit the message to each destination node in as little time and using as short a path as possible.

Chapter 3

Weibull distribution

3.1 Description of Weibull distribution

During the last decades, the Weibull distribution has been the main topic of many researches and studies. Hundreds scientific papers have been written related to this distribution. Weibull is widely analyzed in the field of theory-oriented statistics, because of its large number of special features. In practice, many researchers show interests in its ability to fit to data from many fields, for example in life data, weather data, observations made in business administration or economics, in hydrology, in biology, in engineering sciences, etc [11].

The Weibull distribution is described below based on its characteristic functions.

- For a discrete distribution, the probability density function (DF) is the probability that the variety takes the value *x*. For a continuous function, the DF is the probability that the variety has the value *x*. The formula for the DF of the general Weibull distribution is :

$$f(x|a,b,c)=\frac{c}{b}\left(\frac{x-a}{b}\right)^{c-1}\exp\left\{-\left(\frac{x-a}{b}\right)^{c}\right\};\ x\geq a;\ a\in\Re;\ b,c\in\Re \qquad (3.1)$$

where *a* is the location parameter, *b* is the scale parameter and *c* is the shape parameter. The case where *a* = 0 and *b* = 1 is called the standard Weibull distribution.

- The cumulative distribution function (CDF) is the probability that the variable takes a value less than or equal to *x*. The formula for the cumulative distribution function of the Weibull distribution is:

$$F(x|a,b,c):=\int_{a}^{x} f(u|a,b,c)\,du=1-\exp\left\{-\left(\frac{x-a}{b}\right)^{c}\right\} \qquad (3.2)$$

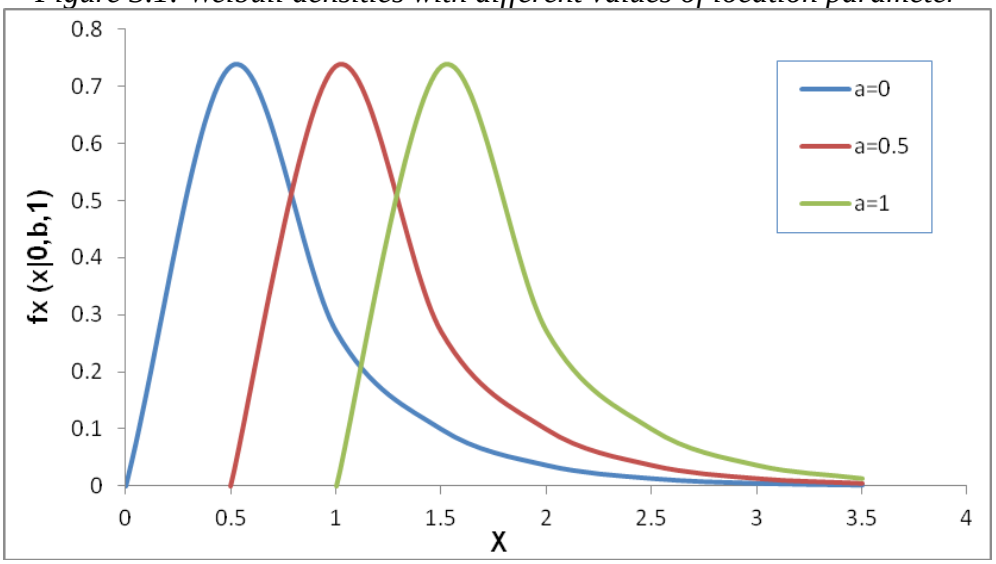
- The hazard rate (HR) is the approximate probability of an *x*–survivor to fail immediately after having spent an amount of time *x*.

$$h(x|a,b,c) := \frac{f(x|a,b,c)}{1-F(x|a,b,c)} = \frac{c}{b}\left(\frac{x-a}{b}\right)^{c-1} \qquad (3.3)$$

3.2 Three-parameter density

A random variable *X* has three-parameter Weibull distribution, with parameters *a*, *b*, *c*, if its density function is the DF in (3.1). The three-parameter density function is the most general form of the Weibull distribution. The *a* parameter is defined on R, so its values range in $-\infty<a<+\infty$. Its unit of measure is the same as the unit of *x* of *X*, normally a unit of time. The *x* variable is duration, which normally is not a negative value. Also the *a* parameter is a duration, exactly the minimum duration, so the domain of *a* is not R, but a smaller interval in [0, ∞).

Figure 3.1: Weibull densities with different values of location parameter

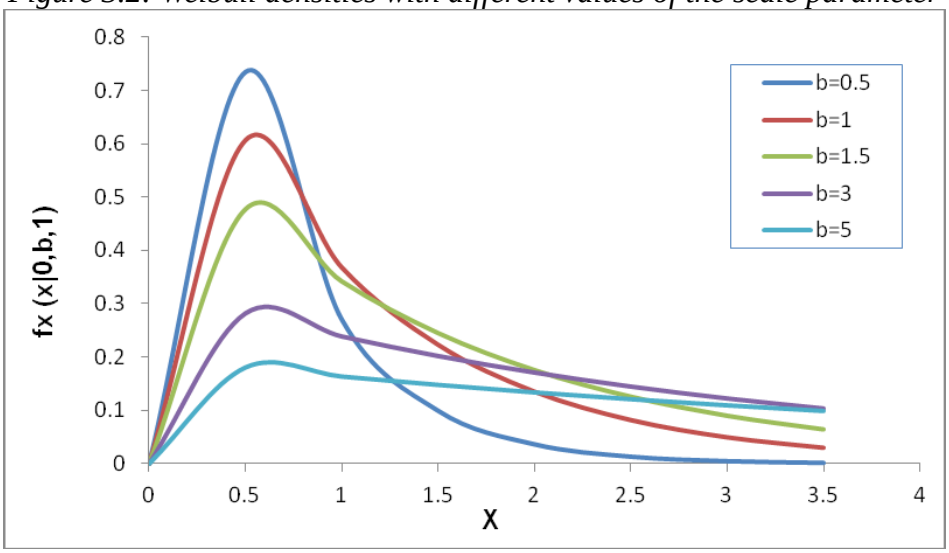


If the other parameters are held constant, and only the parameter *a* is changed, it will result in a parallel shift of the density curve over the abscissa. Increasing the parameter *a* will shift the density curve to the right, and decreasing the parameter *a* will shift the

density curve to the left. This is the reason way that the parameter *a* is often called the location parameter, or the shift parameter.

The *b* parameter in the equation (3.1) has the domain (0,∞). Its unit of measurement is the same as the *x* variable. When the *X* variable is a lifetime, the parameter *b* is called characteristic life. The CDF of all Weibull varies with the same *a* and *b* parameters, but varying the parameter c will intersect at the point with the coordinates *x*=*a*+*b*, and $F_x(a+b|a,b,c) \approx 0.6321$. This means that the chance of surviving the minimum life *a* by *b* units of time is about 63.21%.

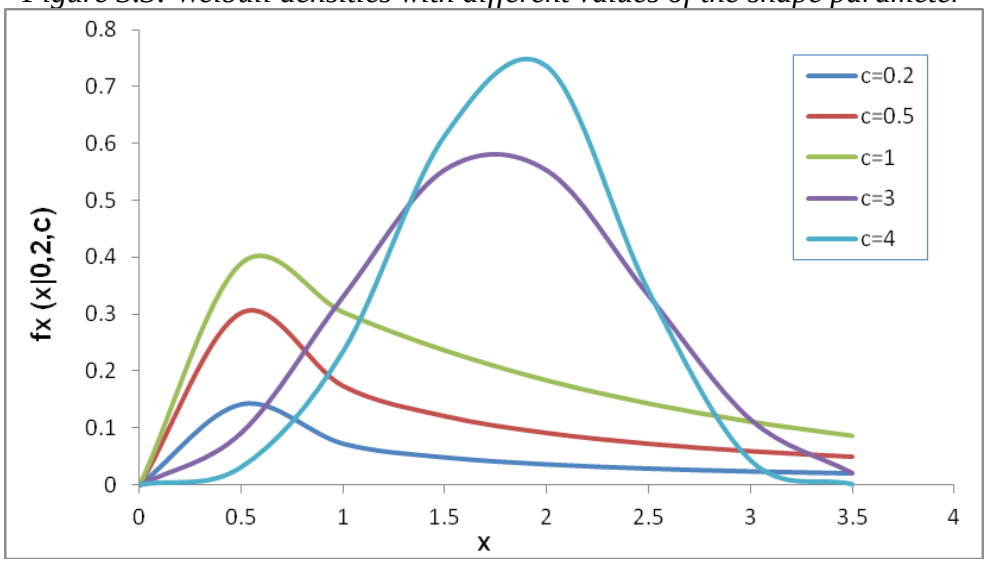
Figure 3.2: Weibull densities with different values of the scale parameter



If only the *b* parameter changes and the other parameters are held constant, it will change the density at *x* in the direction of the ordinate. Enlarging *b* parameter will cause a compression of the density; meanwhile reducing the *b* parameter will stretch the curve. Mostly, *b* parameter is called the scale parameter.

The third parameter in equation (3.1) has the domain (0,∞). It is often called Weibull-slope, because when graphed on Weibull-probability-paper it gives the slope of the CDF.

Figure 3.3: Weibull densities with different values of the shape parameter



If the two other parameters are constant, changing the value of the parameter *c* will change the form of the graph. The parameter *c* is responsible for the shape of the Weibull density graph. In the case when $c<1$, the exponential part of the function dominates, so the density curve is J-shaped. In the case when $c>1$, the polynomial part of the density is more evident, so the density curve becomes uni-modal. The *c* parameter is often called as form parameter, or shape parameter.

3.3 Two-parameters and one-parameter densities

Neutralizing the parameters *a*, *b*, and *c*, by setting $a=0$ and/or $c=1$ results in two-parameter distributions and one-parameter distributions. Not all of them are of equal importance in application.

The densities of the two-parameter versions are:

$$f_x(x|0,b,c)=\frac{c}{b}\left(\frac{x}{b}\right)^{c-1}\exp\left\{-\left(\frac{x}{b}\right)^{c}\right\} \qquad (3.4)$$

$$f_x(x|a,1,c)=c(x-a)^{c-1}\exp\{-(x-a)^{c}\} \qquad (3.5)$$

$$f_x(x|a,b,1)=\frac{1}{b}\exp\left\{-\left(\frac{x-a}{b}\right)\right\} \qquad (3.6)$$

The equation (3.4) is by far the most often found two-parameter Weibull distributions, called the scale-shape-version. The reason is that items normally start to fail after the age of x=0. If a≠0 (but known), it can be transformed in to the (3.4), with the shifted variable X-a. The equation (3.5), the local-shape-version is not scaled, but shifted only. The equation (3.6), the shift-scale-version is nothing but the generalized exponential distribution.

The densities of the one-parameter distributions are:

$$f_x(x|0,1,c)=c\,x^{c-1}\exp(-x^c) \qquad (3.7)$$

$$f_x=(x|0,b,1)=\frac{1}{b}\exp\left\{-\left(\frac{x}{b}\right)\right\} \qquad (3.8)$$

$$f_x(x|a,1,1)=\exp\{-(x-a)\} \qquad (3.9)$$

The functions (3.8) and (3.9) are the special cases of the generalized exponential distribution given by (3.6), either only scaled or only shifted. So c=1 always leads to an exponential distribution. (3.7), the shape version is called reduced or normalized Weibull density [11].

3.4 Maximum likelihood

Many research studies on the Weibull distribution deal with the estimation of its parameters *a*, *b* and *c*. An initial approach, which is appropriate for selecting good estimators for the parameters of a given distribution, is to determine the minimal sufficient–statistic vector. This is the vector of smallest dimension that includes functions

of the observed data yielding all information useful in estimating the parameters and functions thereof. [11]

There are two approaches to estimation and several methods for each of them. The two approaches are point estimation and interval estimation. In point estimation, a numerical value for the k–dimensional vector θ of parameters is calculated. In interval estimation, a k–dimensional region is determined in such a way that this region covers the true vector θ with a specified and predetermined probability $1-\alpha$, $1-\alpha \geq 0.90$ in general, where $1-\alpha$ is the level of confidence [11].

Many methods are available for estimating the parameters, especially those of the Weibull distribution, each having its advantages and disadvantages, and there is no method which is best with respect to all characteristics that may be set up to measure its behavior. Maximum likelihood (ML) is by far the most popular method of estimation. It was introduced by R.A.Fisher (1890-1962). It is an alternative to other estimation methods; the methods of moments and the method of least squares. ML is considered to be one of the most versatile and reliable methods. The maximum likelihood estimators are those parameter values that maximize the likelihood function, given the data available. ML copes with all types of samples, either censored or uncensored, both cases when the data are grouped or not. The likelihood function depends on the distribution of the sampled universe, also on its parameters, on the data and on the way these data are collected. The maximum likelihood estimators are the parameter values, that when it is given the data available, they maximize the likelihood function. Sometimes, there are not known the exact distribution of some maximum likelihood estimators, also the confidence limits of the estimated parameters [11].

3.5 Likelihood functions

Likelihood is a statistical tool used in testing. For a given realized sample, the likelihood expresses the chance of having realized the sample under any set of the parameters governing the sampled universe.

The likelihood is defined as a function of the distributional parameters θ_j (j=1,...,m), collected in the vector θ, given the sampled data [11]. The likelihood might be a joint

density, a joint probability, or a combination of them, depending on the sample, if it's discrete or continuous. The roles of the variables and the parameters in the joint function have been interchanged.

The data consists of a vector of *n* independent observations. The likelihood is a product of *n* simple factors defined as likelihood elements L_i, which express the likelihood of an individual observation.

The likelihood function of an independent sample of size *n* is:

$$L(\theta|data)=K\prod_{i=1}^{n}L_i(\theta) \qquad (3.10)$$

where *K* is a combinatorial constant giving the number of ways the sample might have been observed. It is independent of any parameter in θ.

In a simplified form, it could be writing:

$$L(\theta)=\prod_{i=i}^{n}L_i(\theta) \qquad (3.11)$$

The aim of the maximum likelihood estimation is to find the vector

$$\hat{\theta}:=\hat{\theta}_{ML}$$

that maximizes (3.11). It is rarely done by any procedure of direct optimization. Usually it is done by a gradient method build upon partial derivatives of (3.11), assuming all the parameters $\theta_1,...,\theta_m$ to be continuous. The process of forming these derivatives is made by departing from the log-likelihood function, which is:

$$L(\theta):=lnL(\theta)=\sum_{i=1}^{n}L_i(\theta)\ , \qquad L_i(\theta):=\ln(L_i(\theta)) \qquad (3.12)$$

The partial derivative of $L(\theta)$ with respect to parameter θ_i (i=1,...,m) is denoted

$$\partial L(\theta)/\partial\theta_i \qquad (3.13)$$

and the relation

$$\frac{\partial L(\theta)}{\partial \theta_i}=0 \tag{3.14}$$

is called a likelihood equation. The system of all *m* likelihood equations:

$$\frac{L(\theta)}{\partial \theta}:=\left(\frac{L(\theta)}{\partial \theta_i},\ldots,\frac{L(\theta)}{\partial \theta_m}\right)'=0 \tag{3.15}$$

where *o* is a vector of zeros, is referred to as the system of normal equations.

The solutions $\hat{\theta}_1,\ldots,\hat{\theta}_m$ are stationary points of the log-likelihood function and might include the MLE as global maximum.

The natural wind has always been the main energy source. It is extremely variable in speed and in direction. It is of primary interest to predict the energy output of a wind energy conversion system. The wind speed in a location depends on the geographic position, on the weather system of the location, and how much is the location is above the sea level.

Wind speed varies by the minute, so the wind speed data are collected and analyzed each year. It would be better to collect wind speed data and direction during 10 or more years and to calculate an annual average also a long-term average. But long-term measurements and analysis are expensive and forces projects to wait for a long time to proceed data. A solution might be to measure wind speed data of a location, to compare them to another nearby location, and later to predict the long-term annual wind speed data. This technique is known as the measure, correlate and predict technique (mcp). Wind speed, as a weather variable, changes based on seasons, and for a specific location, it will repeat itself in a period of a year. It is of high interest to relate the wind speed changes in a year with a probability distribution function. It is best related by Weibull probability distribution with two parameters [42].

The analyze of wind speed data and the power of wind speed is the topic of researches [33], [34], [35], [36], [37], [38], [39], [40]. The role of maximum likelihood method in the

Weibull parameter estimation is analyzed in [12], [13], [14], [15], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55].

3.6 The wind and Weibull

The wind speed is the most critical data needed to appraise the power potential of a candidate site. Due to the present day energy demand and growing environmental consciousness, it has become imperative to supplement our energy base with clean and renewable sources of energy [80]. A detailed knowledge of wind characteristics is required for efficient planning and implementation of any wind engineering project. Wind blows due to the warming and cooling of the earth's atmosphere and the changes in temperature. In extreme cases the wind energy can be destructive in nature.
Fortunately, most regions of the world experience moderate range of wind speeds that can allow human to extract energy from the wind [32] [80].

3.6.1 Speed and power relations

The kinetic energy in air of mass "*m*" moving with speed *V* is given by the following in SI units

$$Ek = \frac{1}{2} \cdot m \cdot V^2 J \qquad (3.16)$$

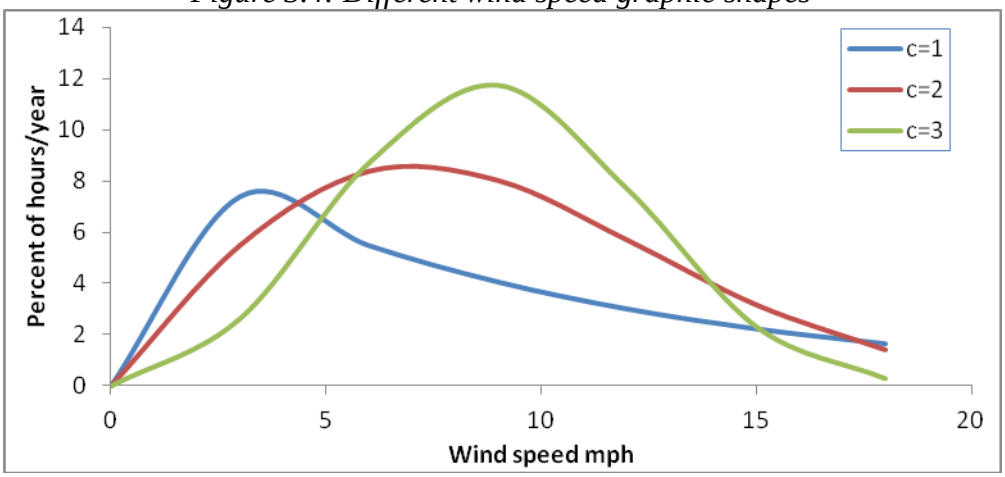
The relation, shown in (3.16), between the power and the wind speed data, highlights the importance of the wind speed as a critical data needed to evaluate the power potential of a specific location. The wind in a specific location depends by the weather system, the terrain of the location, and the height above sea level.

A distribution that describes the variation in wind speed is the Weibull distribution. Its probability distribution function *h(v)*, with two parameters, the scale parameter *b*, and the shape parameter *c*, is given in the equation :

$$h(v)=\left(\frac{c}{b}\right)\left(\frac{v}{b}\right)^{c-1} e^{-\left(\frac{v}{b}\right)^{c}} \quad \text{for} \quad 0<v<\infty \tag{3.17}$$

where v is the wind speed during any time interval.

Figure 3.4: Different wind speed graphic shapes



The curve with the shape parameter c=1 implemented in wind speed case study, is a representation where most days are windless, that means v=0. The curve with shape parameter c=3 is a bell shape distribution, represents the case when some days have high wind and an equal number of days have low wind. The curve with shape parameter c=2 is a typical wind distribution, when most days have lower than the mean speed, while only a few days have above mean wind speed.

For most sites, the shape parameter ranges from 1.5 to 2.5, and the scale parameter ranges from 10 to 20 miles per hour (about 5 to 10 m/s)

Mode speed is defined as the speed corresponding to the hump in the distribution function. This is the speed the wind blows most of the time. Mean speed over the period is defined as the total area under the h-v curve integrated from v = 0 to ∞, divided by the total number of hours in the period (which is 8,760 if the period is one year) [32].

Chapter 4

Programming models and environments

4.1 Introduction to programming models

The programming model describes a parallel computing system in terms of the semantics of the programming language or programming environment.
Meanwhile, a parallel programming model is an abstract view of how a programmer can implement a parallel algorithm. This view includes some specifications such as the architecture where it is implemented, the programming language, the interpreter or the compiler, the libraries. A specific architecture might have different parallel programming models which apply to it [81] [24].
Parallel programming models differ in some features such as:

- the way how the programmer specifies the parallelism, which might be implicit or explicit,
- the level of parallelism, which might be instruction level, statement level, procedural level or loop-level parallelism,
- the way how parallel program parts are declared, which is typical for each programming language,
- the synchronizations of parallel parts of the algorithm,
- the way of communication between parallel units, which might be message passing or shared variables,
- the synchronizations mechanisms to manage the communication between parallel units of the system.

The parallel programming models apply each of the above features, so there are different options and a large number of combinations. The role of the parallel programming models is to support the parallel programming and to offer a mechanism with which the programmer specifies the programming language. The parallel program itself will define the computations that will be executed in parallel [81].

A parallel program is executed in different parallel units, where each of them runs a parallel part of the algorithm, as parallel control flows. These control flows might be managed as parallel processes or as parallel threads. The concept of the thread similar to the concept of the process. They relate to the fact that a process consists of a single or many threads that share the same address space. Meanwhile, different processes do not share the same address space. It is the physical memory architecture the one which specifies if the parallel program will be implemented in parallel threads or in parallel processes.

On shared memory architectures, the parallel control flows are typically implemented as parallel threads, whereas on distributed memory architectures these control flows are organized as parallel processes [81].

The processes or threads executing a parallel program may be created statically at the program start. They may also be created during program execution according to the specific execution needs. Depending on the execution and synchronization mode supported by a specific programming model, there may or may not exist a hierarchical relation between the threads or processes.

4.2 Message Passing Interface

4.2.1 Introduction

The message passing interface (MPI) is a standard developed by the Message Passing Interface Forum (MPIF). More than 40 organizations participated in this forum. They started working on this standard in 1992; meanwhile the first draft is published as Version 1.0 in 1994. The second version is published as MPI-2 in 1997. It includes additional tools like dynamic process management and Fortran 90 bindings [18].

MPI is specified as a library with functions implemented in C programming language, and subroutines implemented in FORTRAN programming language. All these functions/subroutines define an interface for writing message-passing programs that are both efficient and flexible. MPI implementations can be compiled and run on every

platform that supports the MPI standard. Some details of the library are left to the individual vendors, who can upgrade the library to the optimized versions.

Implementations of the MPI-1 standard are available for a wide variety of platforms. The MPI standard includes:

- Point-to-point communication
- Collective operations
- Communication contexts
- Process topologies
- Process groups
- Bindings for FORTRAN 77 and C
- Profiling interface
- Environmental management and inquiry

MPI does not specify:

- Explicit shared memory operations
- Operations that require more operating system support than it was a standard during the adoption of MPI, for example, interrupt-driven receives, remote execution, or active messages
- Program construction tools
- Debugging facilities
- Explicit support for threads
- Support for task management,
- I/O functions [18].

MPI is built as an implementation of the message passing model used in parallel computing. In this model, different processes running at the same time are using local variables. There is no any mechanism that enables a process to access data of another process. In the case when two processes need to share the data, they can send and receive data explicitly. This model generalizes the processes, including not only the processes running in different processors.

MPI offers functions for different types of communication. It also enables the handling of defined data types and topologies.

The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. It can be implemented in a wide range of computer platforms, like shared-memory multiprocessors, networks of workstations, or even single processor machines. The message passing model enables more control over data flow and data location than the shared memory model. In this way the achieved performance is higher. This is the main reason why the message passing model will last as an option of parallel implementation paradigm [18].

MPI implementation and efficiency is analyzed in [56], [57], [58], [59], [60], [61]. An implementation of Jacobi algorithm in MPI with checker board decomposition is analyzed in [56]. Jacobi algorithm is used as an example of how parallel algorithm can be speed up the calculations by emphasizing data and estimating decomposition. This process goes through analysis, design and implementation. Analysis is the phase where the programmer finds the problem in the serial algorithm. This might be the Jacobi iterative method to solve the linear equations. Advanced mathematic calculations and linear algebra are the focus of analysis phase. This phase refers to the mathematical analysis. The design phase includes data portioning and calculation partitioning. This phase consists in specifying how to deal with sub problems of the parallel algorithm. They should be solved in parallel. It is important to specify when to start a fork and when to end it. The related study splits the data in block scope data and in row or column scope data. Another problem to solve in the design phase is the partitioning the calculations. The mathematical calculations are split in block scope calculations and row or column calculations. In the implementation phase, it is used the MPI library to introduce the parallel part of the program. A user defined library based on MPI functions enables the parallel instructions of the program. The results of speed-up of MPI version of Jacobi is an analogue test to this research, for the parallel programming part of the MPI version [56].

4.2.2 Communication modes

A key concept used in MPI is the communication domain which is a set of processes that are allowed to communicate with each other. Information about communication domains is stored in variables of type MPI_Comm, which are called communicators. They are used

as arguments to all message transfer MPI routines and they uniquely identify the processes participating in the message transfer operation. Each process can belong to many different communication domains.

The communicator defines a set of processes that can communicate with each other, by creating a communication domain. There is a default MPI communicator called MPI_COMM_WORLD which includes all the processes involved in the parallel execution. But, in some cases, a communication is performed only within a set of processes. By using a different communicator for each such group, it is ensured that no messages will ever interfere with messages sent to any other group. [5]

There are four communication modes in the MPI: synchronous, buffered, standard and ready. There is a variety of communication modes that define the procedure used to transmit the message, as well as a set of criteria for determining when the communication event (i.e., a particular send or receive) is complete. In all cases, completion of a send implies that it is safe to overwrite the memory areas where the data were originally stored.

- A synchronous send mode completes only when the message is received. When a synchronous send mode operation is completed, the sending process may assume the destination process has begun receiving the message. The destination process need not be done receiving the message, but it must have begun receiving the message.
- A buffered send mode completes regardless the message is received or not. Buffered send mode requires MPI to use buffering. The downside is that the programmer must assume responsibility for managing the buffer. If at any point, insufficient buffer is available to complete a call, the results are undefined.
- A standard send mode might be a synchronous or a buffered one. When MPI executes a standard send mode, one of two things happens. Either the message is copied into an MPI internal buffer and transferred asynchronously to the destination process or the source or destination processes synchronize on the message.
- Ready send mode is used only when matching receive has already been posted at the destination process before ready mode send is called. If a matching receive has not been posted at the destination, the result is undefined.

Blocking and non-blocking communication:

- An MPI operation is blocking, if the return of control indicates that all resources specified in the call can be reused. This is because a blocking operation returns the control only if it is fully completed.
- An MPI operation is non-blocking if the return of control occurs before all effects of the operation are completed before the resources used by the call can be reused. In this way, a call of a non-blocking operation only starts the operation. The operation itself completes only when all state transitions are completed and the resources specified can be reused.

The communications are divided in blocking and non-blocking ones based on the behavior of operations from the execution process. Considering the effect that the communication operations have on other processes, they can be synchronous or asynchronous operations

- Synchronous communication: The communication between a sending and a receiving process is performed such that the communication operation does not complete before both processes have started their communication operation. The completion of a synchronous send indicates not only that the send buffer can be reused, but also that the receiving process has started the execution of the corresponding receive operation.
- Asynchronous communication: Using asynchronous communication, the sender can execute its communication operation without any coordination with the receiving process.

Below is an example of a typical HelloWorld in MPI.

Code 1: Hello World example in MPI

```
#include <mpi.h>;
int main(int argc, char** argv) {
  // initialization
  MPI_Init(NULL, NULL);
  // total number of processors
```

```
  int nr_total;
  MPI_Comm_size(MPI_COMM_WORLD, &nr_total);
  // number of processors
  int nr_processor;
  MPI_Comm_rank(MPI_COMM_WORLD, &nr_processor);
  // Greetings from each node
  printf("Echo from node %d"
  ", in total there are %d nodes\n",nr_processor, nr_total);
  // Finalization
  MPI_Finalize();
}
```

In console is displayed:

```
Echo from node 1 , in total there are 4 nodes
Echo from node 2 , in total there are 4 nodes
Echo from node 3 , in total there are 4 nodes
Echo from node 4 , in total there are 4 nodes
```

4.2.3 Collective communicators

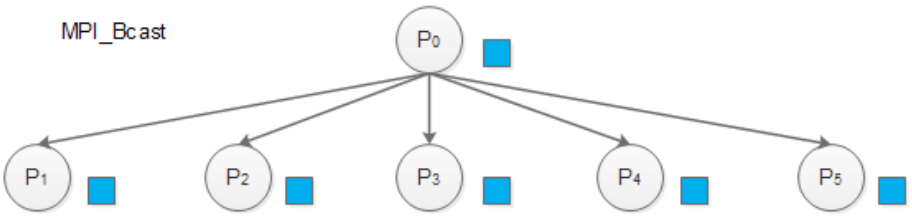
Collective communication enables the data exchange among a group of processes. The communicator argument specifies the processes involved in the communication. In this way, all the processes that members of the communicator should call the same collective subroutine with matching arguments.

All the MPI communication subroutines are blocking. The main subroutines are described below.

- `MPI_BCAST`. This subroutine broadcasts the message from a specific process called root to all the other processes in the communicator given as an argument [18].

One of the main uses of broadcasting is to send out the input to a parallel program, or send out the configuration parameters to all processes.

Figure 4.1: MPI_Bcast example



In figure 4.1, process P_0 is the root process. It has the initial copy of data. All of the other processes receive a copy of the data.

The prototype of `MPI_Bcast` subroutine is:

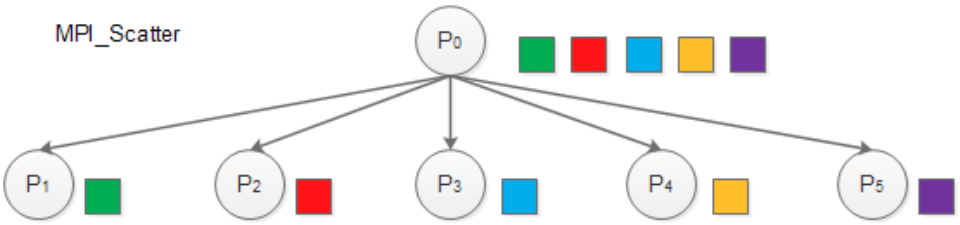
Code2: Prototype of MPI_BCast

```
MPI_Bcast(void* data_sent,
  int data_sent_counter,
  MPI_Datatype datatype_sent,
  int root,
  MPI_Comm communicator_id)
```

- `MPI_Scatter` sends chunks of an array to different processes.

The main difference between `MPI_Bcast` and `MPI_Scatter` is small but important. `MPI_Bcast` sends the same piece of data to all processes while

Figure 4.2: MPI_Scatter



While executing the `MPI_Scatter` function, process P_0 sends different data to all other processes, which will compute the same function over different data.

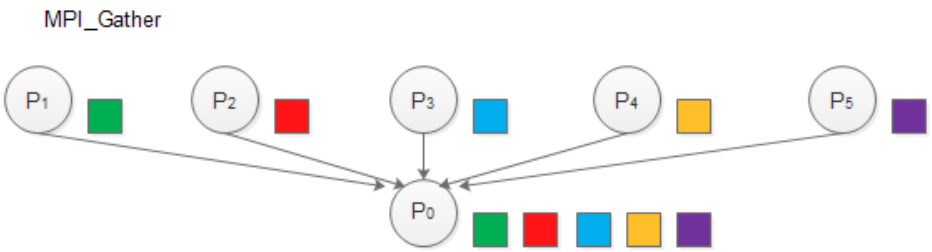
The prototype of this subroutine is:

Code3: Prototype of MPI_Scatter

```
MPI_Scatter(void* data_sent, int data_sent_counter,
  MPI_Datatype datatype_sent,
  void* data_received,
  int data_received_counter,
  MPI_Datatype datatype_received,
  int root,
  MPI_Comm communicator_counter)
```

- `MPI_Gather` is an inverted version of `MPI_Scatter`. It collects elements from many processes and gathers them to one single process.

Figure 4.3: MPI_Gather example



While executing the `MPI_Gather` function, process P_0 gets different data from all other processes, which have called the same function over different data.

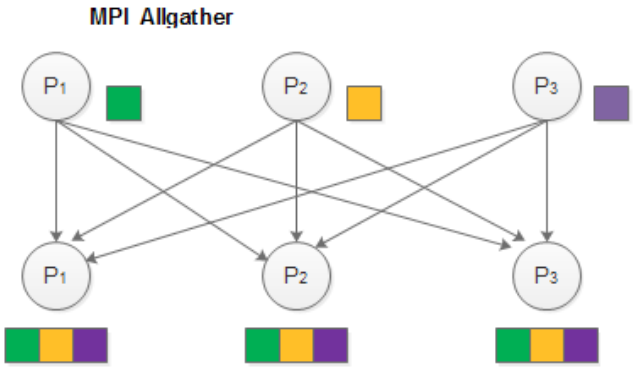
The prototype of this subroutine is:

Code 4: Prototype of MPI_Gather

```
MPI_Gather(void* data_sent,
  int data_sent_counter,
  MPI_Datatype datatype_sent,
  void* data_received,
  int data_received_counter,
  MPI_Datatype datatype_received,
  int root,
  MPI_Comm communicator_counter)
```

- `MPI_Allgather` collects all of the elements from all the processes. Practically, `MPI_Allgather` is an `MPI_Gather` followed by an `MPI_Bcast`.

Figure 4.4: MPI_Allgather example



As shown in figure 4.4, the elements from each process are gathered in the same order as their rank number, but in this case the elements are gathered to all processes. The function declaration for `MPI_Allgather` is very similar to `MPI_Gather`, but with the difference that there is no root process in `MPI_Allgather`.

The prototype of this subroutine is:

Code 5: Prototype of MPI_Allgather

```
MPI_Allgather(void* data_sent,
  int data_sent_counter,
  MPI_Datatype datatype_sent,
  void* data_received,
  int data_received_counter,
  MPI_Datatype datatype_received,
  MPI_Comm communicator_counter)
```

4.3 Parallel programming in threads

4.3.1 Introduction

A thread can be defined as a different stream of control that executes its instructions independently. In this way, a multi threaded process performs different tasks concurrently. These tasks perform different actions, for example, one runs the graphical interface, another performs the I/O operations, and a third one estimates the calculations [25].
The abstract concept of the thread includes every task that a computer does while executing the program. It represents the program state which gets scheduled to run, and which “does the work”. A thread, viewed at an instant of time, is the state of the computation.

There are many kinds of thread libraries: Win32, OS/2, and POSIX. Win32 runs only under Windows platforms. OS/2 threads run only under OS/2 platforms. POSIX is a library which works for all computing platforms. Its specifications: IEEE 1003.1c, aka Pthreads. This library works for almost all UNIX systems, including Linux.
In focus of this research is the “POSIX thread”. Technically, that means the thread “application programming interface” (API) specified by the international formal standard POSIX 1003.1c-1995. The informal term “Pthread” is used below to refer to this API [20].

In the related paper [61], it is analyzed the hybrid MPI/Pthreads implementation, applied in the parallelization of the RAXML phylogenetics code. Adding MPI instructions to the Pthread existing implementation provides the two levels parallelism: coarse-grained parallelism in the MPI instructions and fine-grained with parallel threads. This hybrid multi-grained approach is well suited for high-performance computers which might be clusters of multi-core or shared memory nodes, when applied in [61]. The related paper [61]analyzes the hybrid MPI/Pthreads MPI code is added to the Pthread existing code. The analysis of the parallel algorithm includes four stages. The research [61] concludes that in the first of three stages, the program achieves good performance when applied MPI technique. In the last stage, the program shows good performance while implementing Pthread. Compared to the Pthreads version, the hybrid version of RAXML provides significant performance benefits for all analysis performed on clusters. Of particular importance
is the ability to use multiple computer nodes to achieve greater speed-up and shorter turnaround in a single run [61].

4.3.2 Threads' libraries

4.3.2.1 POSIX Threads

The POSIX standard defines the API and behavior that all the Pthreads libraries must meet. It is part of the extended standard of POSIX. It is not a requirement for meeting XPG4, but it is required for X/Open UNIX 98, and all major UNIX vendors have committed to meeting this standard [25].

4.3.2.2 Win32 and OS/2 Threads

OS/2 implementations contain some differences from the POSIX standard to the degree that the switch from OS/2 threads to POSIX will prove moderately challenging. Microsoft has not announced any plan to adopt POSIX. There are freeware POSIX libraries for Win32 and OS/2 also has an optional POSIX library [25].

4.3.2.3. DCE Threads

Before POSIX completed work on the standard, it produced a number of drafts which are published. Draft 4 was used as the basis for the threads library in DCE. It is similar to the final specifications, but it does contain a number of significant differences. Presumably, no one is writing any new DCE code [25].

4.3.2.4 Solaris Threads

Also known as “UI threads,” this is the library which SunSoft used in developing Solaris 2 before the POSIX committee completed their work. It will be available on Solaris 2 for the foreseeable future. The vast majority of the two libraries are virtually identical [25].

4.3.3 To thread or not to thread

Threads are not always the best option for every serial program. There are some algorithms where the use of threads encounters many difficulties. In some other algorithms, threads implementation does not increase the performance of overall algorithm. Some algorithms are inherently non-concurrent. Threads functions complicate such programs. If every instruction depends on the previous one, then using threads is not a good choice. They would cause deadlocks by waiting the results from other threads [20].

Algorithms that have the following features are mostly implemented in threads:

- Applications which perform complex calculations might be computed in separated threads. They might run in multiprocessor hardware.
- Applications which perform many input or output operations simultaneously, to improve throughput. Distributed server is a typical example of this category. They must response to many clients and execute many input/output tasks over network.

Many applications have a concurrent nature of its algorithm, even when they perform simple input/output task. Thread applications are mostly faster and more responsive than their sequential versions. Also they are easier for development and maintenance that their non-threaded asynchronous versions.

The point is not to replace simple non-threaded programs, with complex threaded ones. Threads are used only when is needed to optimize complex or slow non-threaded programs. Threads are just one way to make the programming tasks easier [21].

As a response to the question “to thread or not to thread?”, threads might not be efficient or appropriate for every programming application, but threaded programming might be very useful in some cases and this is a technique that should be considered while programming in parallel [20].

4.3.4 Types and interfaces

An introduction of Pthread data types, and a description of them, is shown in table 1.
In the table below, the Pthread data types are listed. They are used in the way as described in the standard. The programmer should not assume anything related to the types’ representation. [20].

Table 1 : Pthread types

Type	Description
pthread_t	thread identifier
pthread_mutex_t	mutex
pthread_cond_t	condition variable
pthread_key_t	“access key“ for thread-specific data
pthread_attr_t	thread attributes object
pthread_mutexattr_t	mutex attributes object
pthread_condattr_t	condition variable attributes object
pthread_once_t	“one time initialization” control context

All the Pthread types listed in the table 1 are considered opaque. There is no public definition of these types’ representation, and programmers should not assume anything related of this representation. The programmer uses them only in the manner specified in the standard.

4.3.5 Life of threads

At any moment during execution, a thread is in one of the followed states: ready, running, blocked, terminated.

- In the ready state, the thread is not running because it is waiting for a processor. The thread might be just created, or just unblocked.
- In the running state, the thread is currently running. In multiprocessor system there might be more than one running threads per process.
- In the blocked state, the thread cannot run because it is waiting for some results or any variable, or waiting to complete any I/O operation.
- In the terminated state, the thread has finished its function, or has been canceled from another thread.

Before starting using the functions of Pthread, a header file called `<pthread.h>` is included in the program, to add the data types, the interfaces and the macros of Pthreads. Pthreads functions are named according this naming convention:

pthread [_<object>]_ <operation> (),

The programmer has to split the program into threads which can be executed concurrently with each other. The user threads are mapped to system threads by the library scheduler. The scheduler of the operating system brings to execution these system threads to the processors of the computing system. The programmer cannot perform the mapping of the user level threads to the processors. He cannot control the scheduler of the operating system. This makes easier the program development, but limits the role of the programmer, by not allowing him to map the user level threads according to his needs.

At the start of each program in Pthread, only a single thread is active. It executes the `main ()` function. New threads can be created by the function `pthread_create` with its prototype [24]:

Code 6: Prototype of pthread_create

```
int pthread_create (pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg).
```

The function that enables a thread to wait for the termination of another thread is the `pthread_join`. It has this prototype:

```
int pthread_join(pthread_t thread, void **valuep)
```

After a thread has finished its procedure, it might terminate by itself by calling the function:

```
void pthread_exit (void *valuep)
```

4.4 OpenMP

4.4.1 Introduction

OpenMP was defined as standard from July 1996 to October 1997 and it was an important step in the story of parallel programming models. It is supported by the programmers' community. It made possible the program switching between different platforms of shared memory systems.

OpenMP is not a programming language on its own. It works in conjunction with either standard FORTRAN or C/C++. It is comprised of a set of compiler directives that describe the parallelism in the source code, along with a supporting library of subroutines available to applications. Collectively, these directives and library routines are formally described by the application programming interface (API) now known as OpenMP. The directives are instructional notes to any compiler supporting OpenMP. They take the form of source code comments (in FORTRAN) or `#pragmas` (in C/C++) in order to enhance application portability when porting to non-OpenMP environments [22].

In the last decades, the number of shared memory system is increased significantly. In the same time, most of the programming models are designed to be used in distributed systems. The goal of OpenMP is to offer a standard and portable API which can be used to

write codes for shared systems. Programming for shared systems is easier and can be used mostly when there is a limited amount of parallel scalability. Programming for distributed systems is more difficult, but is used when there is a higher parallel scalability.
In its early stages, OpenMP was considered to be implemented as a library for shared systems. But there were two main reasons why it was not implemented in this way. First, it is easier to write directive-based code, which is automatically ignored from the compiler that does not support OpenMP. Second, directives are processed by the compilers. This makes an optimization in the compiler level possible. Therefore, it wasn't built a library, but an API defined by directives and some library calls. OpenMP API offers portability to OpenMP library for environments that don't support OpenMP.

OpenMP can be a good choice of parallel programming language. But, in some cases even a correct and executable OpenMP program is not achieving a good performance. This is because some basic programming rules are not followed. Experts have listed some rules of thumb to reach a certain basic level of performance. Such suggestions are taken in consideration during the process of building the parallel version in OpenMP of this research. Following these rules, some elementary performance problems are solved, when the program is run using a few threads. With a larger number of running threads, more work is required to achieve a good performance level [22].

The analysis of the OpenMP implementation is focused in the way the parallel threads share the memory. Its behavior has a critical role in the performance of the sequential program, but also in the OpenMP version of the program. A set of factors that influence the performance of the OpenMP program are listed below:

- The way in which the threads access the memory is very important. If each thread accesses a discreet portion of program’s data, it uses the memory hierarchy in an excellent way, especially the thread-local cache.
- The amount of the computation that can be performed only sequentially. This is the part of the code that performs each thread in the parallel version. There is such part of code in every parallel program; it is inevitable.
- The time spent executing the OpenMP constructs. Every directive in OpenMP has its cost in time. For example, during a parallel region’s creation, there is some structural set-up data, which in the sequential version do not happen. All the extra

procedures in the parallel version are part of the parallelization overhead. Different constructs have different cost in time for the parallel performance.

- The way the work is balanced in synchronization points. If threads perform different work, it is very important that this work is balanced, and more or less the threads have same amount of calculations. If they have different amount of work, the faster thread will wait. In a worst case, all the threads will wait one thread, in the synchronization point. During all the time the threads are waiting inactive in a barrier, they are not doing anything for the program. This is the overhead of work imbalance.

While analyzing the performance, it is important to evaluate the execution environment. During the execution of the parallel program, the load on the system should not be greater than the number of processors. In case it happens, the performance is degraded, and the program's behavior is difficult for the analysis.

4.4.2 Getting started with OpenMP

A parallel programming language should: specify a parallel execution, make possible the thread communication, and synchronize the tasks of different threads. Most of parallel programming languages fulfill these requirements by using a sequential language. These languages are familiar to programmers and this makes easier adding the parallel code. Different sequential languages have some constructs that form the parallel execution, or enable the communication. Some API such as MPI or thread library as Pthreads don't create new programming constructs. They form the parallel part by using some calls to runtime library routines.

OpenMP is based on directives to form the parallelism. It consists of a set of directives that can be included in any program written in a sequential language like FORTRAN, C or C++. This use of directives has some advantages. First, such code can be used in any platform, one processor or multiprocessor. In the first case, the processor interprets the directives as comments, by ignoring them, and executing the code sequentially. The second advantage is that the programmer can implement the parallel part based on the serial version. He just adds the directives that enable the parallelization.

Besides the directives, OpenMP includes a small number of runtime library routines. It includes environmental variables too. They both are used to modify the execution parameters. These three components, the directive-based extension, the runtime library routines, the environment variables all together form what is called an API (Application programming interface).

There are OpenMP compilers for many Unix and Windows versions. While switching platform, the OpenMP code should be recompiled. OpenMP implementations in C or C++ include a file named `omp.h`. It contains OpenMP type definitions and library functions prototypes. `omp.h` should be included in all C or C++ files that contain OpenMP codes.

OpenMP creates and manages the parallel synchronization by using three categories of constructs: parallel control constructs, synchronization constructs and data environment constructs.

A parallel control construct checks the flow of control in a program. OpenMP contains two types of parallel control constructs. The first type is a directive that creates multiple threads of execution. These threads can run simultaneously. The directive encloses the block that will run in parallel. It creates the threads that will run this block simultaneously. The second type is construct that distributes the work among parallel threads.

A synchronization construct coordinates the executions of parallel threads. The communication between threads is achieved by reading and writing in shared threads. In this way, it is important to synchronize some thread operations in order to avoid conflicts between threads. Such a conflict might happen when some threads try to access same shared variable in the same time, or when a thread tries to read a shared variable in same time when another thread is writing it. Synchronization is a mechanism that coordinates the parallel threads execution in a parallel program.

A data environment construct manages the communication between threads. Each program begins the execution with only one thread. This thread is called master thread. The data environment of a thread is the address space which contains all variables of the program. The thread, together with its data environment, exists all the time while the program is running. When the master thread executes a parallel construct, it creates a new

thread which runs in parallel with master thread. OpenMP decides whether to implement the program variables as shared variables, or create a copy of variables for each thread.
It is the programmer's choice if the variable will be shared or private. This is implemented by using clauses for each variable. They are used to specify the data environment of parallel threads. A variable can be either shared or private. Also, there exist intermediate data-sharing declarations, such as default, reduction and linear.

A variable declared as shared variable is allocated only in one memory address, during all the time that the parallel construct is running. All threads referring to this variable, will access the memory allocation of this variable, which is stored in only one location. Its memory address will be shared in between threads.

A variable declared as private variable is allocated in different memory addresses, one for each data environment of each thread. Each operation of read or write in this variable, will be referenced to the copy of the thread that did the operation. It will not be visible to the other threads.

Two main types of synchronization are mutual exclusion and event synchronization.

Mutual exclusion controls the access of a shared variable by granting its exclusive access to only one thread. When different threads try to modify the same shared variable, the exclusive access guaranties the variable integrity. OpenMP implements the mutual exclusion by using the critical directive.

Event synchronization is used to note how many times an event in some threads occurs. A form of event synchronization is a barrier. The barrier directive specifies a moment when a thread waits for other threads to come in that point. After all other threads will come at that moment; the waiting thread can continue its execution of the instructions beyond the barrier.

4.4.3 Loop level parallelism

Loop-level parallelism can reduce the time of execution of some applications. This means that different loops’ iterations can be executed concurrently in different processors. OpenMP includes a parallel directive which specifies the loop that will run in parallel. Many programs can switch to parallel version just by run in parallel the right loop, using

the parallel directive. This style is fine grained parallelization and can be applied incrementally. Some of the loops critically slow the performance. They can be parallelized by adding directives, by changing only a few lines in code. The programmer does not need to rewrite the entire code, but only adding such directives, and parallelizes few loops.

The programmer decides which loop to parallelize. To make this decision, he must check carefully which of the loops influence the overall performance of the program. But the program should be built carefully; the result must be the same with both versions, serial and parallel. To achieve a better performance, the execution time of the loop in the parallel version must be shorter than the execution time of the same loop in the serial version.

The program is analyzed by its execution time. It needs to add some function call at the beginning of the program and at the end, which helps to calculate the execution time of the program. Also, OpenMP implementations are analyzed in [62], [63], [64].

A hybrid programming implementation MPI + OpenMP is analyzed in [26], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79]. In [79], the message passing communication is employed by MPI, to achieve the coarse-grained parallelism between nodes. The shared memory communication is implemented by the OpenMP, within nodes, to achieve fine-grained parallelism. Experiments show the parameters of the hybrid form: the execution time and the occupancy of the memory space. There are three parallel programming models: pure MPI, pure OpenMP and hybrid version MPI/OpenMP. The algorithm tested is the Bi-conjugate gradient. It is of interest to compare the results of this related paper, because the main idea is similar to the scope of this research. In this research it is concluded that the mixed mode programming MPI/OpenMP does not obtain a better parallel efficiency than the pure MPI mode. This is due to both overheads of process creating and threads creating.

In [23], a parallel algorithm is analyzed with the OpenMP and MPI. The Gaussian elimination is processed in shared memory environment with the OpenMP, and in distribution environment with the MPI. With a static scheme and a specific chunk size, each processor has a statically allocated chunk of iterations. The allocation of iterations is done at the beginning of the loop, and each thread will only execute those iterations assigned to it.

In the related study [26], one of the major aspects of implementing the Gaussian Elimination algorithm on a distributed memory system is the communication time. It has a significant effect on the resulting performance of the algorithm but, with appropriate modification, it may achieve modest speed-ups. Instead of rewriting the algorithm to distribute the data in a different manner, in the study [26] it is chosen to test the program several times using a different number of processors and varying values of *n*. The technique used for load balancing remains the same, but the chunk sizes to be distributed differently. In the research [26], comparisons are made based on the time spent in the forward elimination phase of the problem. With MPI version implemented in [26], the time spent on communication is compared to the time spent on calculations. Results of the related work show that as the number of work increases, the MPI program displays an improvement in performance as opposed to the OpenMP program where the performance increase seems to diminish. It is possible that as *n* increases, find a point where the distributed environment will show a greater increase in performance than the shared platform [26].

Chapter 5

MPI Implementation in Weibull distribution parameter estimation

5.1 Analysis

One basic way to build a parallel program is to write it from scratch. Another option is to write first the serial version, and then to add the parallel code. In this research, the second way is applied, by parallelizing an existing serial program. To realize this, the steps below are followed:

1. *Determine the hot spots*

Sometimes, most of the CPU time is consumed by small parts in the code. These parts are hot spots, and should be processed in parallel. There are some rules of thumb, determining these hot spots. For example, mathematical functions, like logarithm or exponential functions are more complex than assignment of variables. Also, when these functions are put in loops, applied in any matrix, they will require more CPU time. It is crucial to determine these slow parts of the code, called hot spots, and enclose them in parallel code.

The ML is estimated by using the Newton-Raphson method, shown as a function below:

$$L(a_j, b_j, c_j) = n\left\{\ln\left(\frac{c_j}{b_j^{c_j}}\right) - 1\right\} + (c_j - 1)\sum_i \ln(x_i - a_j) \qquad (5.1)$$

The function in (5.1) is calculated in the Code 7. The instructions are repeated in a loop, which analyzes the entire data sample. In all cases when the data sample is big enough, the loop that processes these data will slow down the entire computation of the result, so it is most probably a potential hot spot in the program.

Code 7: Implementation in C of ML calculation

```
for( i = 0;i<tm; i++)
     {
       if (X[i]!=0.0f)
                xia = X[i];
        else
                xia=0.00001f;
        xmac = powf(xia,(*C));
        xlog = logf(xia);
        xlog2 = xlog*xlog;
        xmacl = xmac*xlog;
        suml = suml + xlog;
        sum = sum + xmac;
        suml2 = suml2 + xlog2*xmac;
        sum2p = sum2p + xmacl;
     }
```

In the implementation of the Weibull parameter estimation, shown in Code 7, there are some calculations which take more time in execution, such as :

```
xmac = powf(xia,(*C));
xlog = logf(xia);
```

These instructions form a possible hot spot in the serial implementation, which slow the execution of the overall application.

2. Preview a target for the performance

In this step, the overall performance of the serial program is checked, determining all the points that consume CPU time. Sometimes, a small part of code consumes most of the CPU time. If this small piece of code is executed in parallel, the version can have a better

performance. In some other cases, parts that require many CPU time are everywhere in the serial program. Maybe the main part of the program should be executed in parallel, in order to archive a good performance. The programmer sets a target performance for the application. During the MPI parallel implementation the target performance is the performance achieved if it's put in parallel part all the hot spots of the code, and that code runs *n* times faster, when it is executed in *n* cores.

The performance should be estimated and compared with the target performance. Some experiments are required to reach the target, and to know how much part of code should be put in parallel.

During the implementation of the parallel version of Weibull parameter estimation, several experiments test which of its functions consumes more CPU time. The decision made is: to put in parallel the Newton function, which calculates the maximum likelihood.

3. *Determine a strategy of parallelization*

There are some questions to be solved, before implementing the parallel code. Some of them are:

- Which variables, or arrays, will be exchanged in the parallel part?
- If there are arrays, will they be shrunk or not?
- If there are nested loops, which of them to parallelize?

In the Weibull parameter estimation, there are some options to implement the parallel part. One version might be to send parts of the array to processes. Each process receives its part. Another option might be to split the matrix in files, one for each processor, and each processor reads its own file. In this research, the second option is implemented.

4. *Implement a method*

In this step is chosen a parallel method. It might be message passing, or OpenMP, or any hybrid version.

In this research the parallel algorithm of Weibull parameters estimation is implemented in four different environments: MPI, Pthreads, OpenMP and in a hybrid version MPI/OpenMP. These implementations are compared based on their performance. The analysis and results are discussed in the next chapters.

5.2 Three patterns of parallelization

During the process of writing a parallel version over an existing serial one, the programmer has to decide where to add the extra code. In case of MPI, the dilemma is where to add the messages that will transmit between the processes. If these decisions are wrong, the parallel version of will not perform well. There exist some patterns; three of them show typical ways where to put the extra parallel code [9].

Pattern 1: Parallelization of only one inner loop

Sometimes, the part of code that takes too much of CPU time processing is in an inner loop. It is not necessary to execute in parallel the outer loop. This pattern suggests putting in parallel only the inner loop.

Pattern 2: parallelization of all inner loops

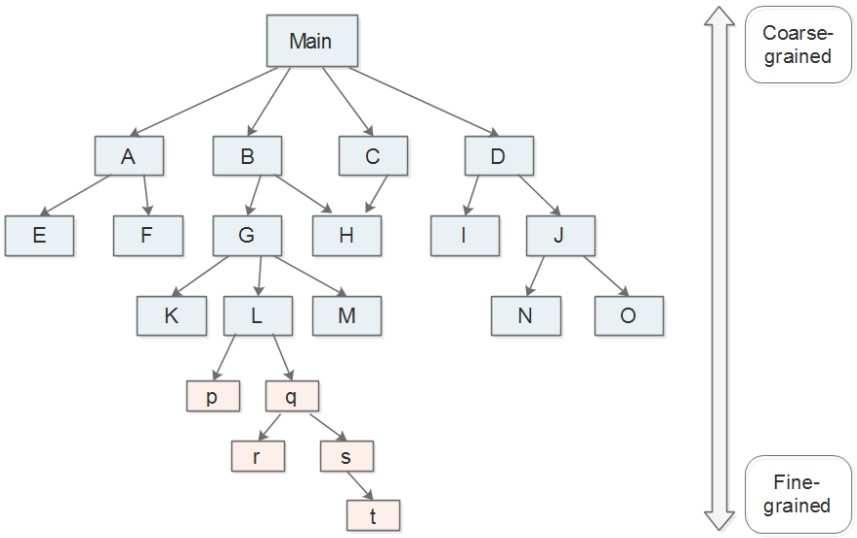
In some cases, more than one inner loop takes too much time to execute. Few inner loops in the same outer loop spend the same amount of time from the total estimation time. If some data should be synchronized after each loop, then the communication between processes will slow down the execution. In this case, all the inner loops should be put in parallel, to minimize the amount of messages exchanged, and to reach a better speed-up.

Pattern 3: Coarse-grained versus fine-grained parallelization

This pattern distinguishes two types of parallelization, based on its granularity. If there is one subroutine that runs in parallel, extra code to keep consistency of the variables in all the loops in the subroutine. In this case the parallelization is fine-grained.

If there is a loop in the main program, then a better choice is to run this loop in parallel. This is a coarse-grained parallelization, when there are fewer instructions to be added, but the work of processes might not be in balance.

Figure 5.1: Fine grained vs Coarse grained parallelism



In the algorithm of this research, there is a loop in the main function. Based on this fact, the chosen implementation is the coarse-grained parallelization. The overhead for this choice is a possible imbalance of the work among processes.

5.3 Parallel design of the algorithm

The data sample is stored in the array Data [N], which consists of integer values of random numbers or wind speed values. This array is split and distributed to *P* nodes which are running the application, and each node calculates the Weibull parameters for the slice it has. The first step in the algorithm is that each node computes the interval of rows that will calculate. At the end of this procedure, the set of rows is sent to each node to continue to the procedure of calculating the Weibull parameters. These estimations are computed for each node.

Code 8: Pseudo code of estimation of Weibull parameters

```
int main ( argc, argv )
{
  MPI Comm rank( MPI COMM WORLD, &rank );
  *The master node sends slices of data to each worker
  if (rank == MASTER)
     MPI Send(&DATA[x[p]], counts[p],MPI COMM WORLD)
  else
     MPI Recv(&DATA[0],counts[rank], MPI COMM WORLD,
&status)
  *Nodes receive their data
  *Each node receives it's section of data;
  if (rank == MASTER)
     MPI_Send(&x[p], xcount[p], MPI COMM WORLD);
  else
     MPI_Recv(&x[0],xcount[p], mtype, MPI COMM WORLD,
&status);

  *After each iteration, the object scores are centered
and normalized.
  MPI Allreduce(&b[0],&c[0],MPI COMM WORLD);
  *Weibull parameters computed;
  *Loop continues;
  MPI Finalize();
  return 0;
}
```

Each node receives its part of data, after the execution of the instruction :

```
MPI Recv(&DATA[0],counts[rank], MPI COMM WORLD, &status);
```

Also, each node executes the function:

```
NEWTON(&b,&c,&n,&m,t,&konvrg);
```

The Newton function performs the estimation of the Weibull distribution parameter estimation, shown in Code 7.

5.4 Data

Tests are performed with data samples of 10,000,000 values and 5,000,000 values. The Weibull parameters are calculated for 100 such samples, chosen randomly from a larger amount of data, from 15,000,000. For each of these samples, the mode, the mean and the variance of the distribution is also calculated. The performance is tested for different number of processors, beginning with the serial version with only one processor, up to twelve parallel processors doing the calculations.

Simulations were executed on Blade server BL685c with 2 six core processors (AMD Opteron™ processor Model 8435 (2.6 GHz, 6 MB L3 Cache)) and 56 GB RAM. The operating system was the Scientific Linux, which is a version of the Red Hat Linux.
The analysis of performance of the implementation of the Weibull distribution parameter estimation in the MPI programming environment is analyzed also when the sample contains wind speed data of specific locations.

The data are taken from three weather stations. One is implemented in Podgorica (PG-WS), another in Herceg-Novi (HZ-WS), and the third one in Shkodër (SH-WS). The PG-WS and HZ-WS measure the wind speed every 10 minutes, every day, meanwhile the SH-WS measures the wind speed every 30 minutes, every day. These data are collected over years, and the implementation for this research used the data collected in 2013 and 2014. For the SH-WS, the first data sample contains the wind speed data of the 2014, in total 17568 items. The second sample, with wind speed data of 2013 and 2014, contains 35088 items. For the PG-WS and HZ-WS, the first data sample contains wind speed data during

2014, in total 105120 items. Meanwhile the second sample contains wind speed data during 2013-14, and contains 210240 items [72], [73], [74] , [75] , [76] , [77] , [78].

During the tests with the samples containing concrete data, the time is not decreased drastically, because the samples do not contain too much data to require complex calculations. One reason of performing the tests on these samples is because of the meaning of the data, as they are concrete wind speed data of a particular location, and not the random items.

The parameters calculated for the second category of data are not related to the meteorology field of study. Nevertheless, the focus of this research is not the result of this example, but the efficiency of the program which produces the results based on real data.

Tests performed in this research used both large and small data sets. The small data set contains wind speed data. The importance of performing tests with this sample consists of the meaning of the sample items. The large data set is used to highlight the role of employing parallel computation, also to specify the relationship between the size of the data set and the speed-up of the implemented parallel algorithms. But, normally, there is dependence between the speed-up and the network speed of the cluster. The tests were made in homogeneous network and the speed-up is not decreased by any slow node of it.

5.5 Speed-up calculations

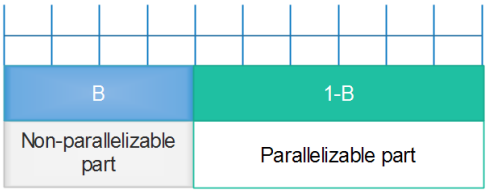
The metric used to determine the efficacy of the parallel version for each method, is the speed-up. As discussed in chapter 2.4, for a system with p processors, S_p is computed as $Sp=T_1/T_p$, where T_p is the execution time of the parallel program using p processors.

Gene Amdahl calculated the speed-up to demonstrate the limitations of parallelism [24]. Let (1-s) be the fraction of calculations performed in parallel. Let T be the total run time. The speed-up is calculated as:

$$\mathrm{Sp}=\frac{T_1}{T_P}=\frac{T_S+T(1-s)}{T_S+\frac{T}{p}(1-s)}=\frac{1}{s+\frac{1-s}{p}}$$

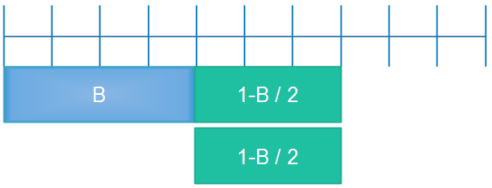
which is known as Amdahl's Law. Amdahl concluded that large scale parallelism would be limited to a speed-up of 1/s. The goal of parallel program designers is to reach the linear speed-up p when s approaches to zero.

Figure 5.2



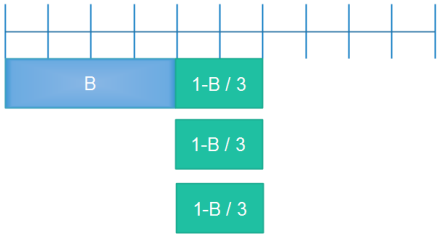
In figure 5.2, the program is divided in parts, if B is the parallelizable part, it will take 4 segments of time to run, and the 1-B parallelizable part will take 6 segments of time to run. It is not possible to minimize the 4 segments of time, but it is possible to minimize the time of the parallelizable part.

Figure 5.3



In figure 5.3, the part B is the same as in figure 5.2. The parallelizable part now is running in 2 cores. It will take half of the time spent while running in one core. The program runs in seven segments of time.

Figure 5.4



In figure 5.4, the part B is still the same, but the parallelized part is running in three cores, and each of them completes its task in two segments of time. The total time it takes is 6 segments of time.

5.6 Results

The results of performance measured in different tests are shown in table 2. In these tests, data are chosen randomly, and the samples contain a large amount of data, the first 10,000,000 random data, and the second 5,000,000 random items.

Table 2 : Results of MPI version, sample with 10000000 and 5000000 items

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 10,000,000 items			The sample contains 5,000,000 items		
1	11718	1	1	5857	1	1.00
2	6022	1.94	0.97	3050	1.92	0.96
3	4081	2.87	0,95	2122	2.76	0.92
4	3098	3.78	0,94	1566	3.74	0.94
5	2543	4.60	0,92	1325	4.42	0.88
6	2048	5.72	0.95	1103	5.31	0.89
7	1792	6.53	0.93	941	6.22	0.89
8	1567	7.47	0.93	834	7.02	0.88
9	1425	8.22	0.91	758	7.72	0.86
10	1292	9.06	0.90	685	8.55	0.86
11	1254	9.34	0.84	638	9.18	0.83
12	1220	9.60	0.80	622	9.41	0.78

In each row of the table there are the metrics of time, speed-up and efficiency of the MPI parallel program of Weibull parameter estimation, for a corresponding number of cores *p*.

The first column of the table contains the number of the cores that run the application. The second and the fifth columns of the table contain the time of execution, in seconds, when the sample contains 10,000,000 items and 5,000,000 items. The third and the sixth columns contain the value of speed-up, when the sample contains 10,000,000 items and 5,000,000 items. A super linear speed-up $S_p>p$, is achieved when the number of necessary iterations is smaller than the number of iterations in the serial program.

The forth and the last columns show the efficiency of the parallel version, as the fraction of linear speed-up of the program, $Q_p=S_p/p$. As the number of machines involved in the computation increases, so does the communication time. This communication overhead in most cases prevents us from achieving linear speed-up, though execution times have indeed improved. In general, the speed-up has a decreasing rate of return per node as the number included in the calculation increases.

Executing the serial program, with large items set, one needs 11718 seconds to estimate the parameters of Weibull distribution for this sample. The results show how parallel programming helps in reducing this time. MPI technique gets the result 1.94 times faster as soon as a second processor is added. The time is reduced up to 1220 seconds when is executed the MPI version in 12 cores.

The table also represents the time spent for estimating Weibull parameters for a sample containing 5,000,000 items. Results show that while executing the serial program, the parameters' estimation is obtained after 5857 seconds. Again, for a different sample, the parallel program shows its efficiency in time, as soon as we add a second processor. The minimum time spent, for these tests, is during the execution of the parallel versions in 12 cores. In this case the time spent is 622 seconds.

By looking at the efficiency, it is noticed that it dramatically drops after 10 nodes. This is due to the heavy communication overhead of adding extra computers to the computation. In the case when data sample contains 5,000,000, the efficiency is unstable

while the application runs in one to eight processors. It becomes poor when the application is running in nine to twelve processors.

Figure 5.5: Speed up of MPI version, sample with 10000000 items

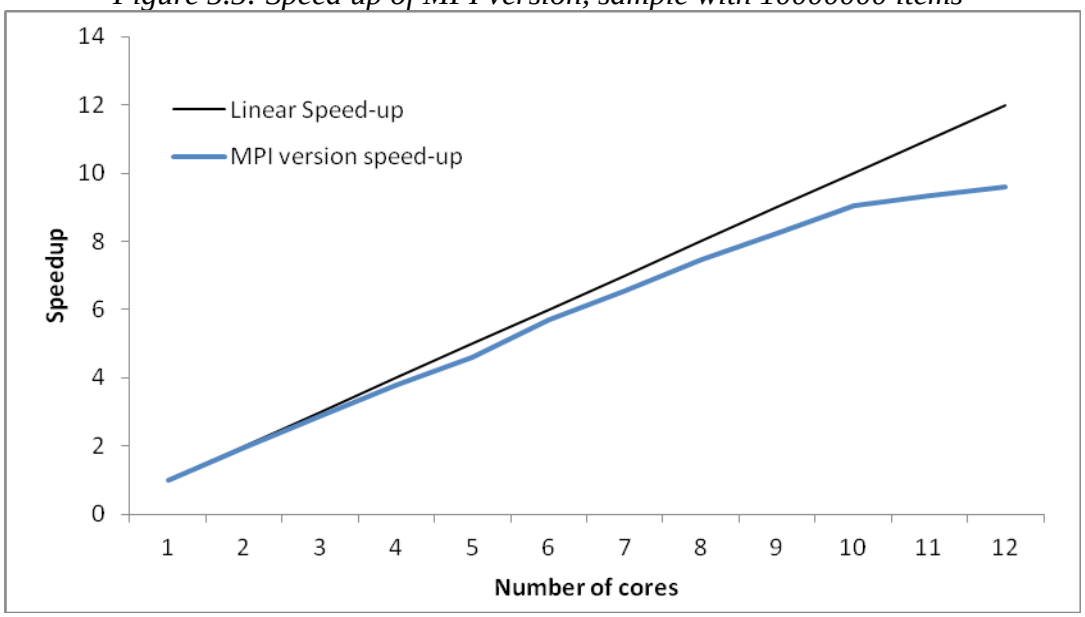


Figure 5.5 shows the deviation of curve of the speed up of MPI version from the ideal speed-up, while the sample contains 10,000,000 items. The linear curve is the ideal speed-up, represented with a linear function. Both are put in the same graph, to highlight the small difference between the speed-up of this MPI version and the ideal speed-up.

The tests show that the speed up increases in a linear form, according to the Gustafson's law [3]. The linear graphic depends also on an alpha coefficient which is the non-parallelizable fraction of any parallel process. A parallel efficiency of 1 corresponds to ideal, linear speed-up. In this case, the alpha coefficient is small, because the implementation is made with a small fraction of non parallelized part of code. As the graphic shows, for this sample, the number 10 of nodes is crucial, as the speed-up falls after adding the eleventh node. The reason might be the communication between processes.

Figure 5.6: Speed-up of MPI version, sample with 5000000 items

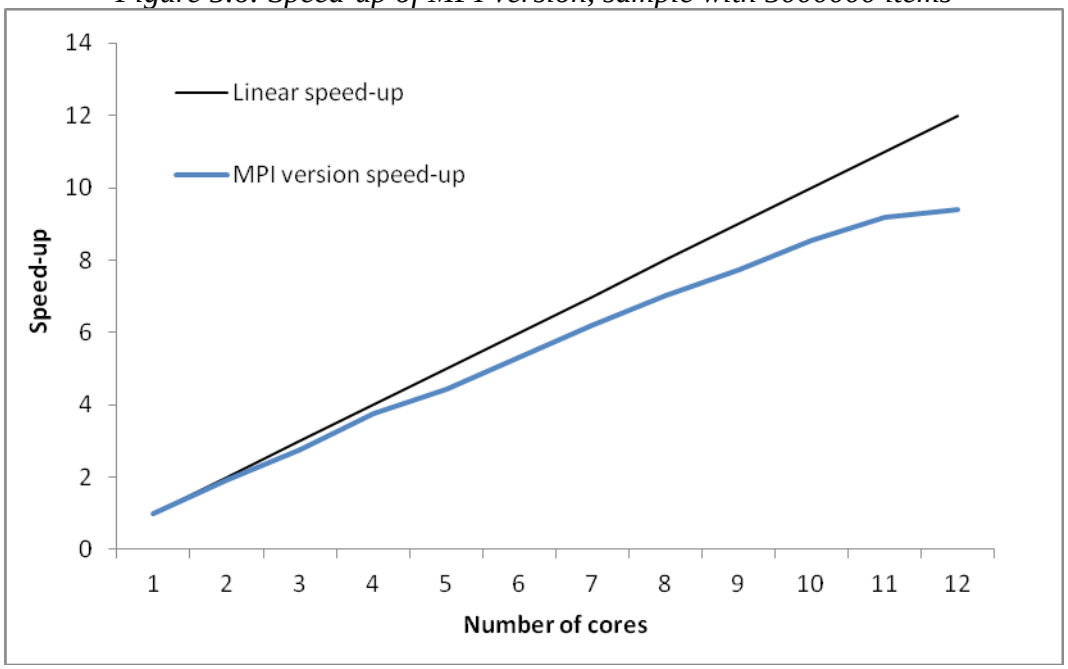


Figure 5.6 shows the curve of speed-up for MPI, when the sample contains 5,000,000 items. The speed-up tends to be linear when the application is run in two up to eight nodes. The number of eight nodes is crucial, as the speed-up tends to increase fewer when the application is running in eight, up to twelve nodes.

Table 3 contains the results of tests of performance when the sample contains wind speed items of 2014 in SH-WS and PG-WS. These samples have a small number of items, but they contain meaningful and concrete data: wind speed data in two different weather stations.

Table 3 : Results of MPI version, sample with wind data during 2014

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 17520 items of SH-WS			The sample contains 105120 items of PG-WS		
1	34.92	1.00	1.00	70.11	1.00	1.00
2	18.63	1.87	0.94	36.09	1.94	0.97
3	13.29	2.63	0.88	24.48	2.86	0.95
4	9.55	3.66	0.91	18.09	3.88	0.97
5	8.35	4.18	0.84	15.48	4.53	0.91
6	6.68	5.23	0.87	13.30	5.27	0.88
7	5.64	6.19	0.88	11.03	6.36	0.91
8	4.83	7.23	0.90	9.88	7.09	0.89
9	4.35	8.02	0.89	8.81	7.96	0.88
10	3.99	8.74	0.87	8.05	8.71	0.87
11	3.68	9.48	0.86	7.41	9.46	0.86
12	3.54	9.87	0.82	7.13	9.83	0.82

The first column of the table 3 contains the number of processors used in the tests.
The second, third and the forth columns of the table contain the results when the sample is wind speeds during 2014, in PG-WS.

The fifth, sixth and seventh columns contain the results, when the sample is wind speed data during 2014 in SH-WS.

When the sample is the wind speed data in PG-WS during 2014, the sample contains 105120 items. Executing the serial version, it takes 70.11 seconds to get the parameters of Weibull distribution for this data set. Using MPI technique, the speed-up reaches the value 1.94, when the program runs in two cores. Running the MPI version with 12 processors, it takes 7.13 seconds to calculate the results of the parameter estimation. The maximum speed-up is 9.83, and is obtained is when executing the program in 12 cores. The efficiency of the MPI version, estimating the parameters for this sample, is highly decreased when the program runs in eight or more cores.

The sample of the wind speed data in SH-WS in 2014 contains 17520 items. The serial version needs 34.92 seconds to estimate the result for this data set. Using MPI technique, the speed-up is 1.87, when the program runs in two cores. While running the MPI version with 12 processors, 3.54 seconds is needed to estimate the result of the parameter

estimation. The maximum speed-up is 9.87, and is obtained when executing the program in 12 cores. The efficiency of the MPI version, estimating the parameters for this sample, is not stable.

Figure 5.7: Speed up of MPI version, sample with wind speed data in PG-WS

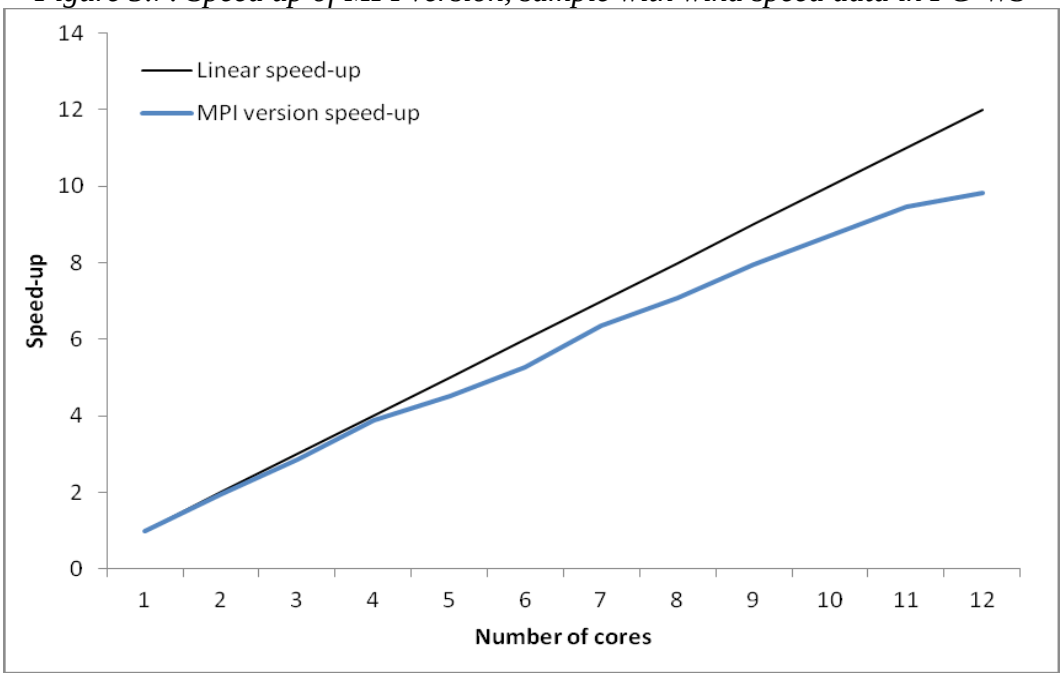


Figure 5.7 shows the curve of speed-up for MPI, when the sample contains wind speed data along 2014 in PG-WS. The speed-up tends to be linear almost during the entire curve, from two up to twelve nodes.

Figure 5.8: Speed up of MPI version, sample with wind speed data in SH-WS

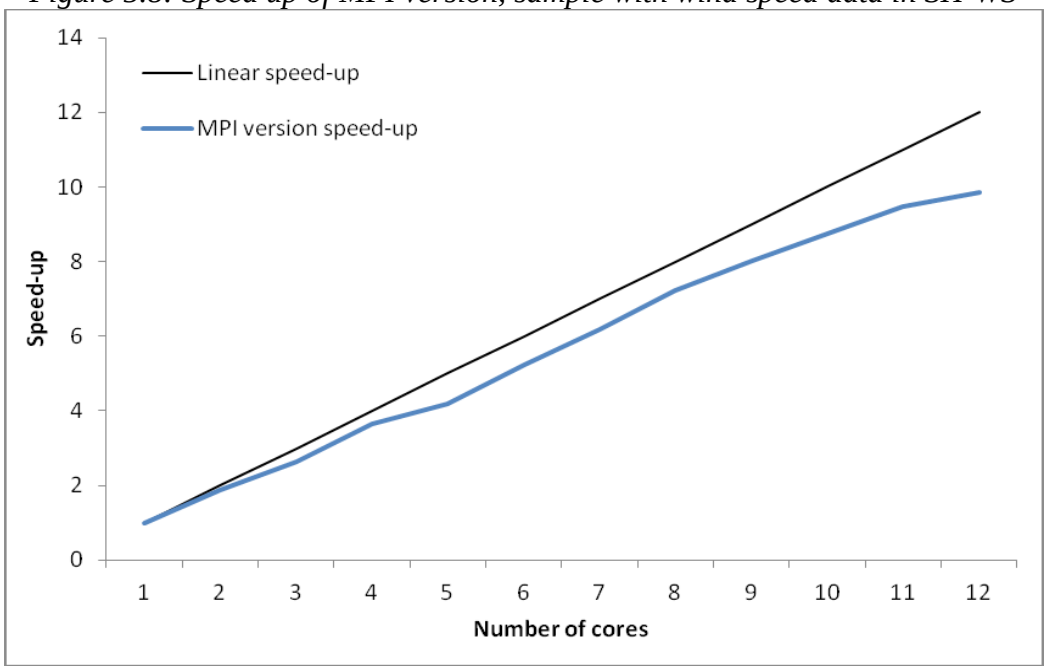


Figure 5.8 shows the curve of speed-up for MPI, when the sample contains wind speed data during 2014 in SH-WS. The speed-up tends to be linear for almost the entire curve.

Table 4 displays results of tests when the sample contains wind speed data of the 2013 and 2014, in PG-WS and in SH-WS.

Table 4 : Results of MPI version, sample with wind speed data during 2013-14

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 35040 data of SH-WS			The sample contains 210240 data of PG-WS		
1	66.71	1	1.00	136.76	1	1.00
2	34.52	1.93	0.97	71.60	1.91	0.96
3	23.29	2.86	0.95	49.19	2.78	0.93
4	17.21	3.88	0.97	36.47	3.75	0.94
5	14.72	4.53	0.91	31.01	4.41	0.88
6	12.23	5.45	0.91	25.56	5.35	0.89
7	10.51	6.35	0.91	21.99	6.22	0.89
8	9.40	7.10	0.89	19.32	7.08	0.89

9	8.38	7.96	0.88	17.67	7.74	0.86
10	7.66	8.71	0.87	15.94	8.58	0.86
11	7.05	9.46	0.86	14.85	9.21	0.84
12	6.78	9.84	0.82	14.50	9.43	0.79

The first column of the table 4 contains the numbers of processors involved in the tests. The second, third and fourth columns contain metrics of the tests performed on wind speed data in SH-WS. This set contains 35040 items, which are the wind speed data during 2013-14. The serial version takes 66.71 seconds to get the parameters of Weibull distribution for this data set. Using MPI technique, the speed-up reaches the value 1.93, while the program runs in two cores. Running the MPI version in 12 processors, it takes 6.78 seconds to calculate the results of the parameter estimation. The maximum speed-up is 9.84, and is obtained is when executing the program in 12 cores. The efficiency of the MPI version, estimating the parameters for this sample, is highly decreased when the program runs in eight or more cores.

The fifth, sixth and last columns contain metrics of the tests performed on wind speed data in PG-WS. These data are collected during 2013-14, and are in total 210240 items. The serial version takes 136.76 seconds to get the parameters of Weibull distribution for this data set. The speed-up increases to 1.91 by adding the second process. The parallel version in MPI, running in 12 processes, takes 14.50 seconds to calculate the results of the parameter estimation. The speed up is 9.43, and is obtained is when executing the program in 12 cores. The efficiency of the MPI version, estimating the parameters for this sample, is stable up to eight processors, but is dramatically decreased after adding the ninth processor.

Figure 5.9: Speed up of MPI version, sample with wind speed data in PG-WS

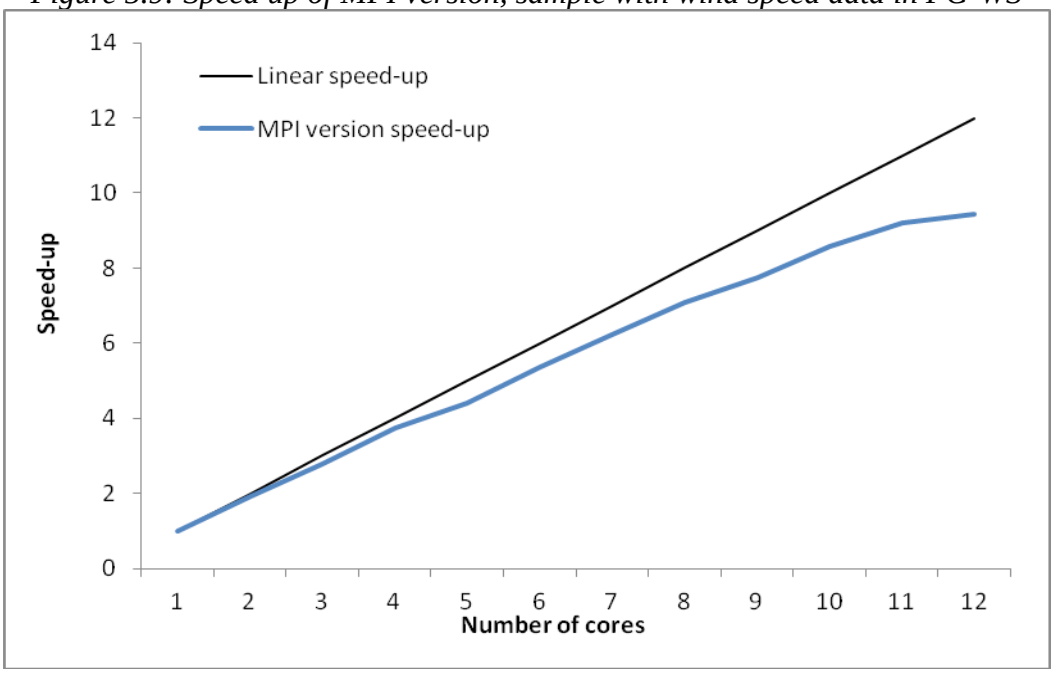


Figure 5.9 shows the curve of speed-up for MPI, when the data sample contains wind speed data during 2013-14 in PG-WS. The speed-up tends to be linear when the application runs up to eight nodes. Adding new nodes, next to eight, does improve the speed-up, but not in the same tend.

Figure 5.10: Speed up of MPI version, sample with wind speed data in SH-WS

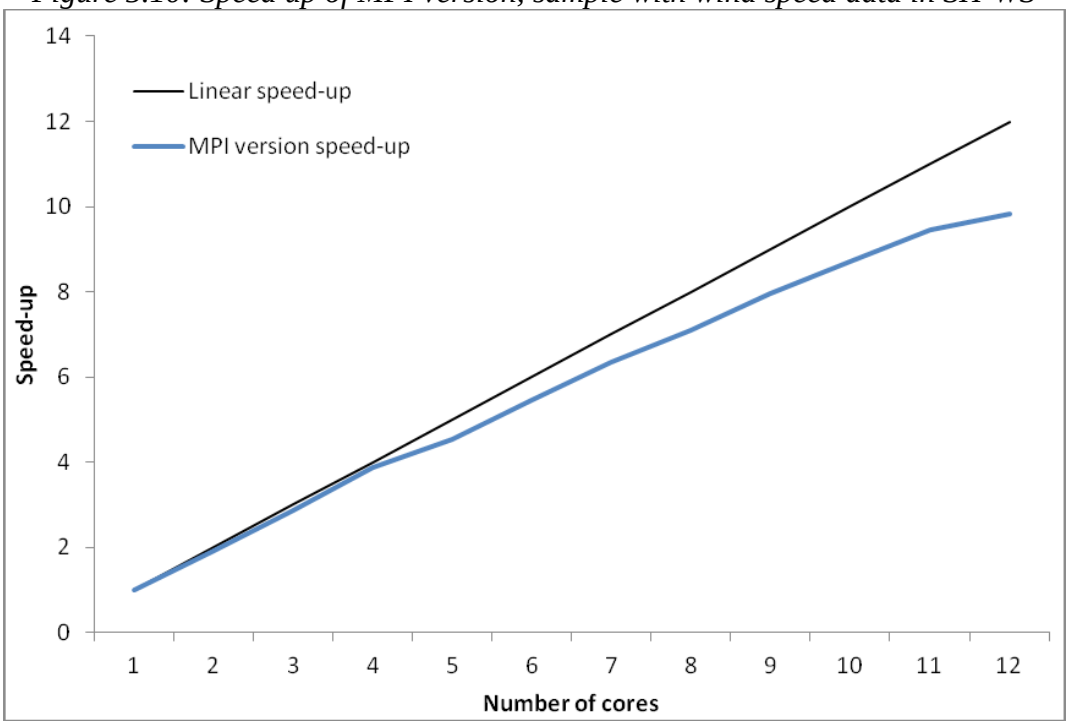


Figure 5.10 shows the curve of speed-up for MPI, when the sample contains wind speed data during 2013-14 in SH-WS. The speed-up tends to be linear almost during all the tests, while the application is running in two up to twelve nodes.

It can be concluded that the parallel version implemented in MPI shows relatively good efficiency and speed-up. When set of data is small, in case when the sample contains wind speed data in only one year, the efficiency of the MPI version is better than when the sample contains more items. The speed-up of the MPI version reaches a maximum value of 9.87 times faster than the serial version, while the program is running in 12 cores, in the case when the sample is wind speed data measured in one year in Shkodër weather station. This sample is the smallest in size, and the MPI version shows the best performance while processing this sample.

For MPI version, best performance, including speed-up and efficiency, is achieved when the sample contains real wind speed data, with a size of 17520 values.

After the estimation, the parameters of Weibull distribution for concrete wind speed data, in Podgorica and in Shkodër are as following:

- The sample containing wind speed data measured in Podgorica, during 2013-14, has the scale parameter b=2.638411 and shape parameter c=3.548239.
- The sample containing wind speed data measured in Shkodër, during 2013-14, has the scale parameter b=2.948302 and shape parameter c=3.239495.
- The sample containing wind speed data measured in Podgorica, during 2014 has the scale parameter b=2.612460 and shape parameter c=3.537527.
- The sample containing wind speed data measured in Shkodër, during 2014, has the scale parameter b=2.848374 and shape parameter c=3.309258.

Chapter 6

Posix implementation in Weibull distribution parameter estimation

6.1 Analysis

A programmer should analyze which part of the program will be included in the parallel procedure of threading. To make such a decision, there are some criteria that define if the code is suitable for threading or not. If a task in program follows at least one of these criteria, it might be run in threads [21].

1. *The task is independent of other tasks*

This is the case when the task is using separate resources from other tasks. Also the current task can be executed independently, and no other task depends on its result. If the above statements are true, then the task is independent and the need of synchronization is minimized. Most probably this task will run independently and will not block any other thread.

2. *The task will be in blocked state in long intervals of time*

If the task spends a lot of time waiting for an input/output operation, it is a good decision to put it in a separate thread, so the other tasks will accomplish more work in less time

3. *The task uses a lot of CPU cycles*

If the task includes complex calculations, like matrix crunching, hashing or encryption, then it might be better run in another thread. These time-consuming calculations that are independent are potential candidates for threading. In a multithreading environment, one thread might execute these computations, and other threads might deal with input/output operations.

4. *The task responds to asynchronous events*

If the task handles such events that occur in random intervals, interrupts from hardware, or network communications, it might be a good decision to put this task in a separate thread. In this way they can run and manage these events, apart from the rest of the application.

5. *The task has different priority than other tasks in the application*

If the task has to complete in a specific time, or its work is more time crucial than other tasks then it might be better to execute it in another thread. Scheduling considerations are a good reason for threading a task.

6.2 Overheads' calculations

6.2.1 Thread creation overhead

When a thread is created, the Pthreads library searches the database and allocates new data structures.

This synchronizes the creation of this thread with the creation of another thread that might happen in the same time. An option might be reusing threads. This technique minimizes the cost of the creation of the new thread [21].

6.2.2 Thread switches overhead

Once created, threads share some CPU limited resources. Mostly, the number of threads in the program outreaches the number of available CPU's. Scheduling a new thread requires a switch between threads. The running thread pauses, and its registers are brought in from swap space. The scheduler selects from the queue another thread, based on its priority, and its registers are brought in from swap space. In some cases the switch is a voluntary choice of the thread. For instance, this is when the thread is waiting for an I/O operation. The situation is like the thread asks the operating system to suspend itself, and to run another thread. In other cases, the switch is an involuntary choice. This happens when the thread has exceeded its quantum, and it should yield the CPU. In a perfect situation, no thread should be suspended involuntarily [21].

Some Pthreads implementation allows the programmer to control the scheduling policy, and allowing ensuring a quicker response time for high priority threads. The overall application might run a bit slower than under the default policy. The reason is the favorable treatment for the high priority threads that cause more switches.

Too many thread switches mean that the program has too many threads. To speed up the program, a solution might be to try running it with fewer threads. Analysis and tests will help to determine when the system reaches its saturation point and limit for concurrent threads [21].

6.2.3 Synchronization overhead

Each synchronization object, that is mutex, or condition variable, or key, requires Pthreads library to operate. It should create and maintain data structures and execute some code to perform its role. All these objects have its cost, which can be magnified by the way in which the programmer deploys the synchronization object.

One way to reduce synchronization costs is to take a chance: search without acquiring locks, lock the nodes found, and then confirm that the locked nodes are correct. If a synchronization conflict causes the wrong nodes to be locked, then release the locks and start over. Normally, this kind of conflict is rare, which is why we call this technique optimistic synchronization [6].

6.3 Choosing the implementation

6.3.1 User space implementation

Pthreads implementation is one of these three types:

1. thread’s implementation in user-space,
2. thread’s implementation in the kernel,
3. hybrid implementation, both in user-space and in kernel [2].

A thread implemented in user space executes the instructions in its program or in a linked library.

Otherwise, the thread implemented in kernel executes its instructions within the operating system's kernel. Usually, such instructions are a result of a system call or exceptions.

All threads in the above three types are conforming to the Pthreads standard and provide concurrency. The threads implemented in user-space don't provide global scheduling scope and don't allow that multiple threads of the same process executed in parallel in different

CPUs. Meanwhile, threads implemented in the kernel don’t offer high scalability when a process has more than 10 threads [2].

In this research, the threads are implemented through the interface called Pthreads. The platform and the operating system are crucial in the way how this implementation is provided. The Linux operating system uses kernel threads.

6.3.2 Handling deadlocks

Deadlock can be defined formally as follows: A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause [2]. Deadlocks mostly happen when there is an unlocked mutual exclusion in the code. In these cases it is solved easily by adding the locking part in the code. Another reason for the deadlocks is the wrong order in which threads are locked. To solve this, the programmer must analyze the code more in details, and put locks in the same order. Another problem that leads to deadlock is when a thread is in waiting status for a condition variable, but this variable is not signaled from the other thread. This form of deadlock is solved if the programmer checks the way that thread interprets the condition.

The Pthreads library knows everything about the mutexes and condition variables in use. The programmer lists which thread is waiting for which mutex or condition variable, and finds out what caused the deadlock. Pthread's libraries help in filling this list, if the programmer adds some procedures wrapping the thread calls in the code [72], [76].

6.3.3 Handling race conditions

A race condition happens when two or more threads share the same data and at least one of these threads accesses these data without going through synchronization mechanism. It might happen that a thread reads shared data at the same time as another thread may get an inconsistent value. This depends upon the timing between two threads [78].

It is very difficult to detect race conditions. It may not happen every time, but only one in hundred executions. When it happens, it does not show any sign, except when it causes bad memory reference and makes the program terminate. In this case the race condition is identified as a problem. Unfortunately, Pthreads library does not help in managing the race conditions [20].

A crucial issue to deal is the analysis of how much data will be shared among threads. If threads share a small amount of data, the chance to create bugs that causes these problems decreases. Moreover, there will be less synchronization overhead, and the threads will run faster. The program complexity is reduced, and there will be improvements in its performance.

6.4 Parallel design of the algorithm

The initial matrix N × 1 consists of integer values, and is block distributed to cluster nodes by the master node. The pseudo-code for the algorithm is written in Code 9. The number of rows depends on the sample chosen, and the number P is the total number of nodes used. The first step in the algorithm is that each node computes the interval of rows that will calculate. At the end of this procedure, the set of rows is sent to each node to continue to the procedure of calculating the Weibull parameters. These estimations are computed for each node.

The number of threads that will be run simultaneously is given by the user. Each thread runs the procedure of Weibull parameter estimation. When terminated, the threads run the `pthred_join` function, to wait for other threads.

The main function, where threads are created and synchronized:

Code 9: Pseudo code of threads synchronization

```
int main ( argc, argv ){
  printf ("Give the number of threads .\n" );
  scanf ("%d",&numthrds);
  *This loop creates  a  new  thread within a process.
  for(i=0;i<numthrds;i++){
     newThr=pthread_create(&callThd[i],NULL,initialize,
(void *)i);
     if (newThr)
        printf ("Error during thread creation \n");
  }
  *Each new thread executes the initialize procedure
  *in the initialize procedure it is read the matrix
with data
  *Also initialize procedure calls the serial Newton
method
  *for each thread, pthread_join() waits for it to
terminate.
  *If that thread has already terminated,
  * then pthread_join() returns immediately.
  for(i=0;i<numthrds;i++){
     newThr = pthread_join( callThd[i], (void
**)&status);
        if (newThr)
           printf ("Error in thread join \n");
  }
```

6.5 Results

The results of performance, when the sample contains random numbers, are shown in table 5.

Table 5 : Results of Pthread version, sample with 10000000 and 5000000 items

No. of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 10,000,000 items			The sample contains 5,000,000 items		
1	11718	1	1	5857	1	1
2	5960	1.97	0.9831	2938	1.99	0.9968
3	4025	2.91	0.9704	1956	2.99	0.9981
4	2947	3.98	0.9941	1471	3.98	0.9954
5	2393	4.90	0.9794	1184	4.95	0.9894
6	1998	5.86	0.9775	997	5.87	0.9791
7	1715	6.83	0.9761	843	6.95	0.9925
8	1502	7.80	0.9752	741	7.90	0.9880
9	1337	8.76	0.9738	660	8.87	0.9860
10	1213	9.66	0.9660	625	9.37	0.9371
11	1122	10.44	0.9494	592	9.89	0.8994
12	1022	11.47	0.9555	560	10.46	0.8716

Each row of table 5 contains the data for an execution of the parallel Pthread implementation, for a corresponding number of cores (in this case, threads). The execution times are compared to the serial running time when the application runs in one single core. The first column of table 5 shows the number of cores that run the application.

The first column of the table 5 contains the number of the cores that run the application. The second and the fifth columns of the table contain the time of execution, in seconds, when the sample contains 10,000,000 items and 5,000,000 items. The third and the sixth columns contain the value of speed-up, when the sample contains 10,000,000 items and 5,000,000 items. The forth and the last columns show the divergence from the linear speed-up for the program. These two columns contain the fraction of the linear speed-up the program achieves, that is the $Q_p=S_p/p$.

When the set contains 10,000,000 items, 11718 seconds is needed to get the parameters of Weibull distribution for this sample. The results in the table show how this time is

reduced when the application is run in parallel, using different number of nodes. Pthread version estimates the parameters 1.97 times faster as soon as a second processor is added. The time is reduced up to 1022 seconds when is executed in 12 parallel cores.

When the set contains 5,000,000 items 5857 seconds is needed to generate the parameters of the distribution. Again, the parallel program shows its efficiency in time, as soon as the application runs in parallel, adding a second thread. The minimum time spent, for these tests, is during the execution of the parallel versions in 12 threads. In this case the time spent is 560 seconds.

Figure 6.1: Speed up of Pthread version, sample with 10000000 items

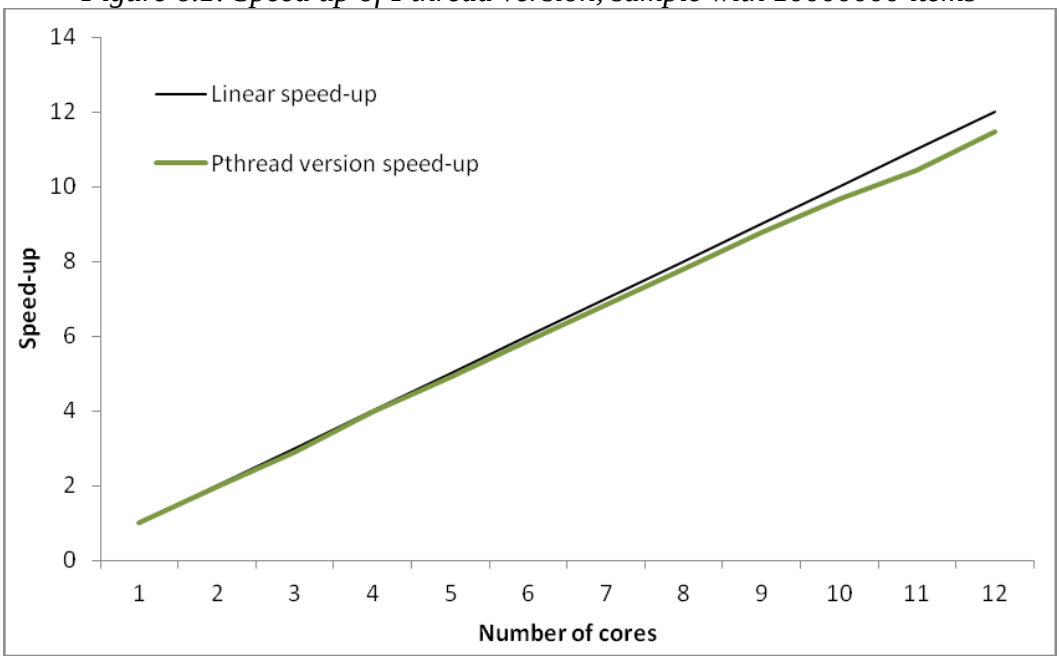


Figure 6.1 shows the curve of speed-up for the Pthread version, when the sample contains 10,000,000 items. The speed-up tends to be linear almost during all the tests, while the application is running in two up to twelve threads.

Figure 6.2: Speed up of Pthread version, sample with 5000000 items

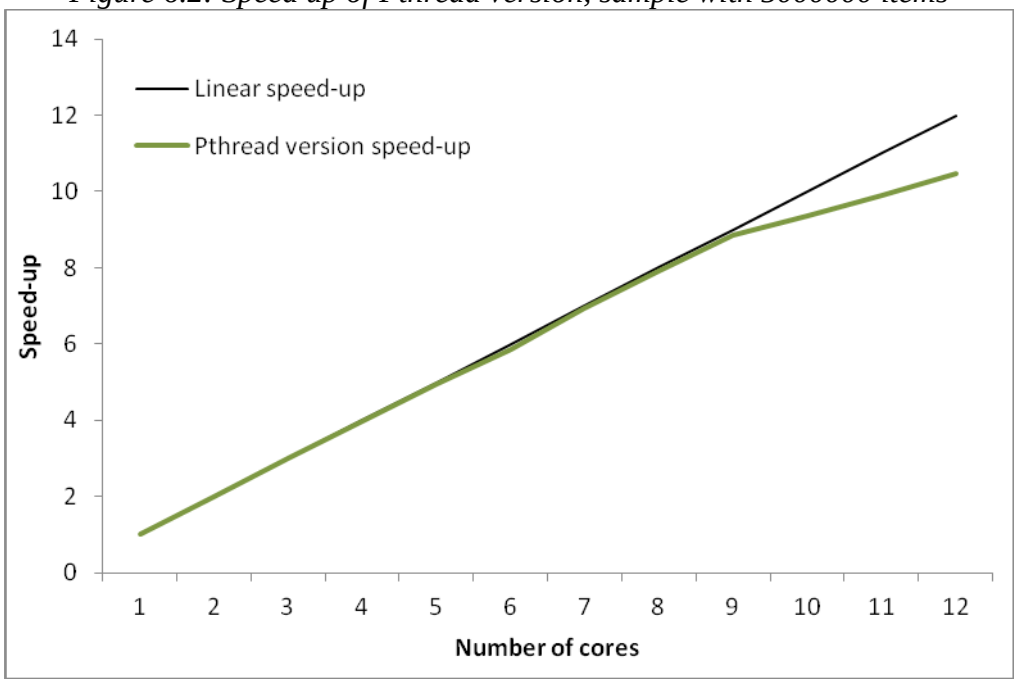


Figure 6.2 shows the curve of speed-up for Pthread version, when the sample contains 5,000,000 random items. The speed-up tends to be linear almost during all the tests, while the application is running in two up to twelve parallel threads.

The table 6 contains the result of performance when the sample contains data of PG-WS and SH-WS during 2014.

Table 6 : Results of Pthread version, sample with wind speed data during 2014

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 105120 data of PG-WS			The sample contains 17520 data of SH-WS		
1	70.11	1	1.0000	34.92	1	1
2	36.29	1.93	0.9661	17.55	1.99	0.9950
3	24.56	2.85	0.9514	11.68	2.99	0.9967
4	18.10	3.87	0.9683	8.77	3.98	0.9950
5	14.93	4.70	0.9392	7.07	4.94	0.9880
6	12.42	5.65	0.9410	5.95	5.87	0.9783
7	10.75	6.52	0.9315	5.02	6.95	0.9929
8	9.16	7.65	0.9563	4.42	7.90	0.9875
9	8.29	8.46	0.9396	3.94	8.87	0.9856

10	7.50	9.34	0.9344	3.73	9.37	0.9370
11	6.99	10.03	0.9122	3.53	9.89	0.8991
12	6.69	10.47	0.8727	3.34	10.45	0.8708

Tests are performed using concrete wind speed data. When the sample is wind speed data during 2014 in PG-WS, it takes 70.11 seconds to get the parameters of Weibull distribution for this data set. Pthread version processes them 1.94 times faster just by adding another parallel thread. Running the application in 12 threads needs 6.69 seconds to calculate the results of the parameter estimation. The maximum speed-up is 10.47, while the application is running in 12 threads. The efficiency of the Pthread version is very high during all tests, from two up to twelve threads.

When the sample is the wind speed data in SH-WS, during 2014 the sample contains 17520 items. Executing the serial version, it takes 34.92 seconds to get the parameters of Weibull distribution for this data set. The Pthread version reaches the speed-up 1.99 when it’s added a second thread. Running the Pthread version with 12 processors, it takes 3.34 seconds to calculate the results of the parameter estimation. The maximum speed-up is 10.45, and it is obtained is when executing the program in 12 threads. The efficiency of the Pthread version is relatively high, during all the tests for this sample.

Figure 6.3: Speed up of Pthread version, sample with wind speed data in 2014 in PG-WS

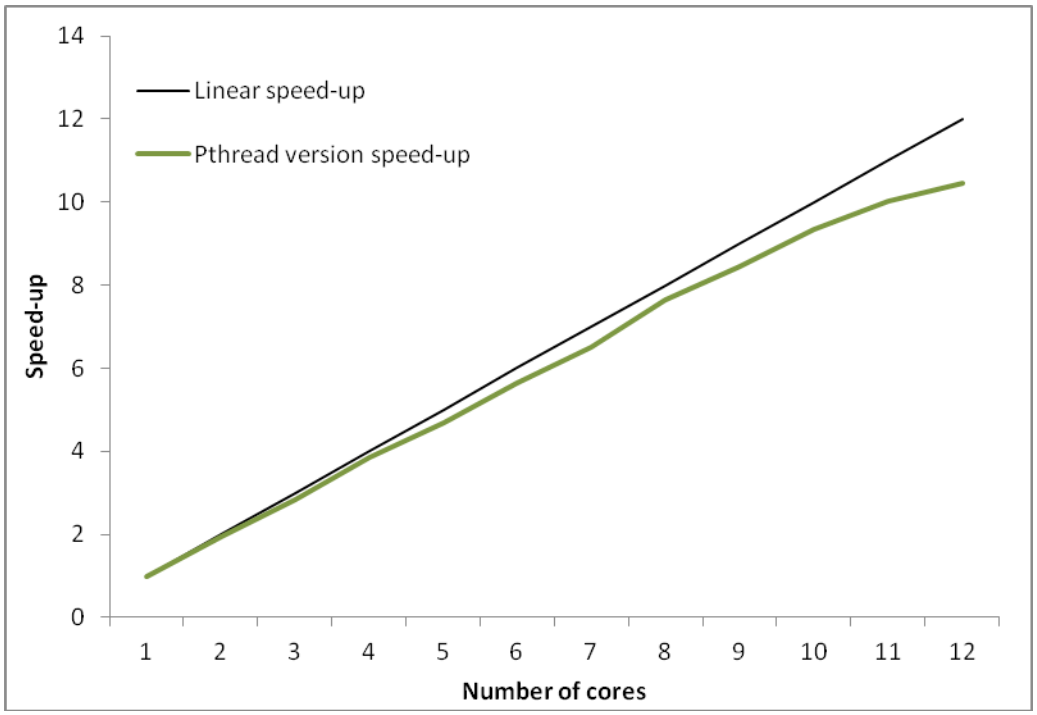
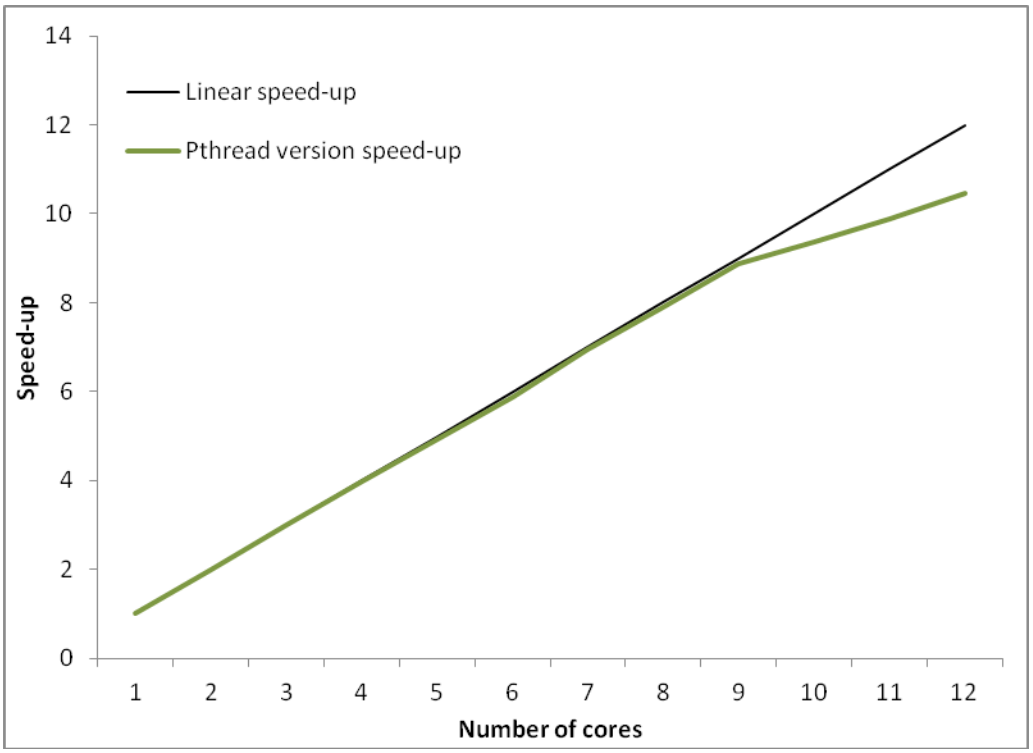


Figure 6.3 represents the graphic of the speed-up of the Pthread version, when the sample is the wind speed in PG-WS during 2014. The speed-up for this test is almost linear. When the application is run in two up to nine threads, the curve of speed-up tends to be almost super linear. The performance falls while running the application with ten threads. The efficiency has the value nearly 1, and that is the case when the parallelism is in its maximal values of efficiency.

Figure 6.4: Speed up of Pthread version, sample with wind speed data in 2014 in SH-WS



The figure 6.4 is a graphical representation of the speed-up of the Pthread version, while the data sample is the wind speed measured in SH-WS during 2014. The speed-up tends to be almost linear, for these tests too. The critical point is when running the application in ten threads. At this point the performance degrades and after adding other threads it is no longer linear.

In table 7 are shown the tests of performance when the sample contains wind speed data of 2013-14 in SH-WS and PG-WS.

Table 7 : Results of Pthread version, sample with wind speed data during 2013-14

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 210240 data of PG-WS			The sample contains 35040 data of SH-WS		
1	136.76	1.00	1	66.71	1	1
2	72.61	1.88	0.94	34.53	1.93	0.97

3	50.42	2.71	0.90	23.37	2.85	0.95
4	37.67	3.63	0.91	17.22	3.87	0.97
5	30.53	4.48	0.90	14.21	4.70	0.94
6	25.70	5.32	0.89	11.82	5.65	0.94
7	22.17	6.17	0.88	10.23	6.52	0.93
8	19.43	7.04	0.88	8.72	7.65	0.96
9	17.19	7.96	0.88	7.88	8.47	0.94
10	15.48	8.83	0.88	7.14	9.34	0.93
11	14.84	9.21	0.84	6.65	10.02	0.91
12	14.51	9.42	0.79	6.36	10.48	0.87

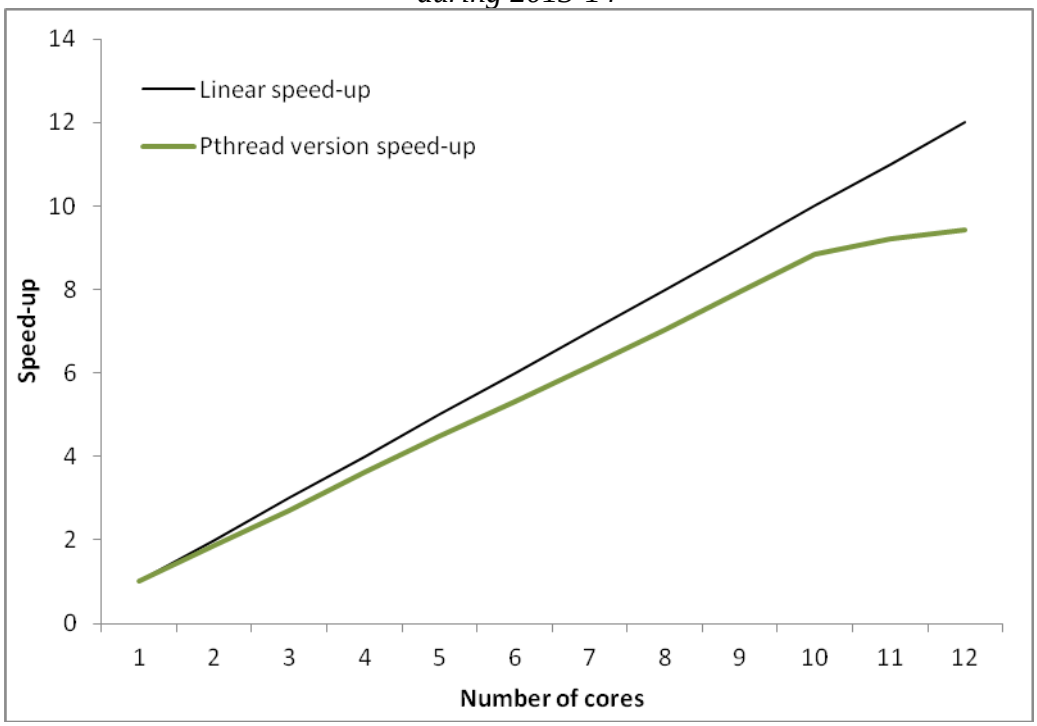
Each row of table 7 contains the data for an execution of the parallel Pthread implementation, for a corresponding number of cores (in this case, threads). The execution times are compared to the serial running time when the application runs in one single core.

The first column of the table contains the number of the threads that run the application. The second, third and forth are the results while testing the application with sample the wind speed data in PG-WS during 2013-14. The fifth, sixth and last columns are the results while testing the application with sample the wind speed data in SH-WS, during 2013-14.

The first sample contains 35040 data, and the second sample contains 210240 data. While estimating parameters for the first sample, the serial version needs 136.76 seconds to estimate the parameters. Pthread version estimates the parameters 1.88 times faster as soon as a second thread is added. The time is reduced up to 14.51 seconds when it is executed in 12 parallel threads.

While estimating parameters for the second sample, the serial program takes 66.71 seconds to generate the parameters of the distribution. Again, the parallel program shows its efficiency in time, as soon as the application runs in parallel, adding a second thread. The minimum time spent, for these tests, is during the execution of the parallel versions in 12 threads. In this case the time spent is 6.36 seconds.

Figure 6.5: Speed up of Pthread version, sample with wind speed in PG-WS during 2013-14



The figure 6.5 is a graphical representation of the speed-up of the Pthread version, while the sample is the wind speed measured in PG-WS during 2013-14. The speed-up tends to be almost linear, for these tests too, with a constant decrement of performance while incrementing the number of threads involved.

Figure 6.6: Speed up of Pthread version, sample with wind speed data in SH-WS, during 2013-14

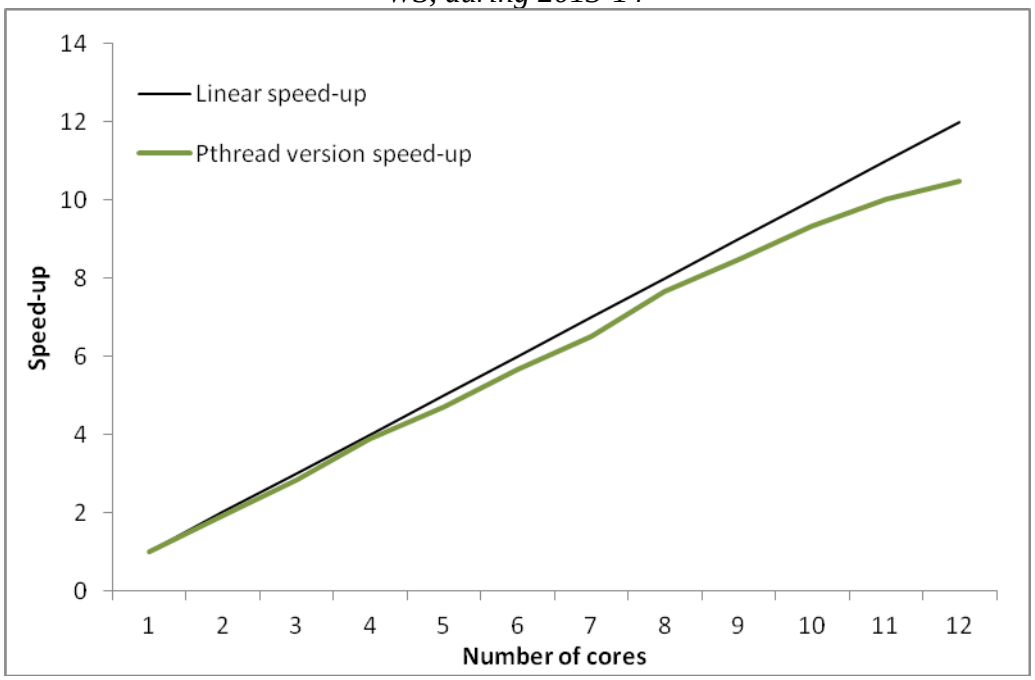


Figure 6.6 is a graphical representation of the speed-up of the Pthread version, while the sample is the wind speed measured in SH-WS during 2013-14. The speed-up tends to be almost linear, for these tests too.

In Pthread implementation, the speed-up and efficiency is relatively high, while it is running in 1 up to 12 cores. The best speed-up is achieved while the application is running in 12 cores, and it is 11.47. This is during the estimation of Weibull parameters of a sample with 10,000,000 random data. In this execution is reached the optimal efficiency 0.95. *The Pthread version shows best performance in cases when the sample is wider in size.*

After the estimation, the parameters of Weibull distribution for concrete wind speed data, in Podgorica and in Shkodër are as following:

- The sample containing wind speed data measured in Podgorica, during 2013-14, has the scale parameter b=2.638411 and shape parameter c=3.548239.

- The sample containing wind speed data measured in Shkodër, during 2013-14, has the scale parameter b=2.948302 and shape parameter c=3.239495.
- The sample containing wind speed data measured in Podgorica, during 2014 has the scale parameter b=2.612460 and shape parameter c=3.537527.
- The sample containing wind speed data measured in Shkodër, during 2014, has the scale parameter b=2.848374 and shape parameter c=3.309258

Chapter 7

OpenMP implementation in Weibull distribution parameter estimation

7.1 Analysis

While transforming a sequential code in a parallel OpenMP code, there are some costs to take into consideration. The OpenMP version needs extra time to create the parallel regions, to share the work among threads, and to manage the synchronization. This time depends on the compiler that translates the code, the characteristics of the runtime library routines and the platform where it is applied. The cost is calculated by adding the time of the threads creation and start-up. Extra time is needed for encapsulating the parallel region into a special function. Additional time is needed to block or unblock threads, to fetch work and signal threads that they are ready.

7.2. Implementation

7.2.1 Best practices

Two main characteristic of OpenMP, the flexibility and convenience, contain a risk. The programmer might build different parallelization strategies, all leading to different correct OpenMP versions, but having different performance. Sometimes, the version that is most intuitive is not the best one in performance. Unfortunately, the parallel inefficiency is not detected by checking the code. Mostly, this happens when the programmer did not follow some basic rules. Let's introduce some best practice in order to try to follow them in the implementation of the OpenMP version, used in this research.

- Optimize barrier use: Barriers are expensive operations, with lots of cost in time, so its usage should be minimized; try to avoid them completely.

- Avoid large critical regions: A critical region ensures that only one thread executes simultaneously the code it encapsulates. The more code in the critical region, the bigger a chance to have a larger time the thread waits to enter in this region. This is the reason the programmer has to minimize the code in a critical region. Also, a critical region includes the part of code that will be run one at a time; an update enforces the access to only one memory location. In the implementation in Weibull distribution parameter estimation, the critical region is not used because the threads do not apply the calculations in the same set of data, and they do not modify any shared variable.
- Maximize parallel regions**:** Large code in parallel regions enables more use of items in cache. In the case when the code contains different loops that might run in parallel, a decision is to put each loop in individual region, or to encapsulate all in one region. A good practice is to minimize the number of parallel regions and to put the loops in one parallel region. In this implementation in Weibull distribution parameter estimation, the parallel code is encapsulated only in one region, which contains all the hot spots in the code and calculates the maximum likelihood.
- Put the outer loops in parallel region**:** Another best practice is to avoid using parallel regions in inner loops, but instead to put the outer loop in parallel.
- Set the correct load balance**:** In some algorithms, defining the proper balance of the threads is a key point in reaching a good performance. In this research, data are organized in matrix and they will be processed in different threads. It is crucial the balance of tasks among threads. If this balance does not work, some threads will wait for the slowest one, in the synchronization point. In the implementation in Weibull parameter estimation, the chunk of items for each thread is calculated by dividing the total amount of items by the number of threads. Each thread performs the estimation over a balanced amount of items.

7.2.2 The choice to use private or shared data

It is a common dilemma for a programmer if a variable should be shared or private. If a variable is shared, then there exists one instance of this variable which is shared among all

threads. If a variable is private, then each thread has its own local copy of the private variable. Both options lead to a correct program, but its performance differs.

In this research this dilemma is about the two-dimensional array. Different threads need access to it. There are different ways to declare it. One option is to declare different one-dimensional array, one per each thread. Another option is that each thread allocates a two dimensional array declared as private, in the parallel region. The second choice is preferred mostly. Shared variables, in the first option, introduce an overhead. One instance of a variable is shared between multiple threads. Therefore, it is often best to minimize the number of shared variables when a good performance is desired.

7.3 Results

The rows of the table 8 represent results of tests, in particular, the time, speed-up and efficiency of the OpenMP parallel program of Weibull parameter estimation, for a corresponding number of nodes *p*. Results of the firsts tests are shown in Table 8. The two samples used contain 10,000,000 items and 5,000,000 items, both of them chosen randomly.

Table 8 : Results of OpenMP version, sample with 10000000 and 5000000 items

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 10,000,000 items			The sample contains 5,000,000 items		
1	11718	1	1	5857	1	1
2	6368	1.84	0.92	3232	1.81	0.91
3	4426	2.65	0.88	2248	2.60	0.87
4	3399	3.45	0.86	1713	3.42	0.85
5	2778	4.22	0.84	1397	4.19	0.84
6	2327	5.03	0.84	1168	5.01	0.84
7	1961	5.97	0.85	992	5.90	0.84
8	1709	6.86	0.86	902	6.49	0.81
9	1555	7.54	0.84	814	7.19	0.80
10	1458	8.04	0.80	771	7.59	0.76
11	1371	8.55	0.78	729	8.04	0.73
12	1312	8.93	0.74	690	8.49	0.71

The first column of the table 8 contains the number of the cores that run the application. Next three columns contain results for the sample 10,000,000 items. The fourth, fifth and sixth columns contain the results of tests when the sample contains 5,000,000 items.

For the sample containing 10,000,000 items, the OpenMP version gets the result 1.84 times faster when a second thread is added. The time is reduced up to 1312 seconds when it is executed the OpenMP version in 12 parallel threads.

The table also represents the time spent for estimating Weibull parameters for a sample containing 5,000,000 items. The minimum time spent, for these tests, is during the execution of the parallel versions in 12 threads. In this case time spent is 690 seconds.

Analyzing the efficiency, for the sample with 10,000,000 items, it decreases gradually while adding the number of threads. For the sample 5,000,000 the efficiency decreases gradually while adding threads, up to the eighth one. Adding threads from eight to twelve, the efficiency decreases with the value 0.1.

Figure 7.1: Speed up of OpenMP version, sample with 10000000 items

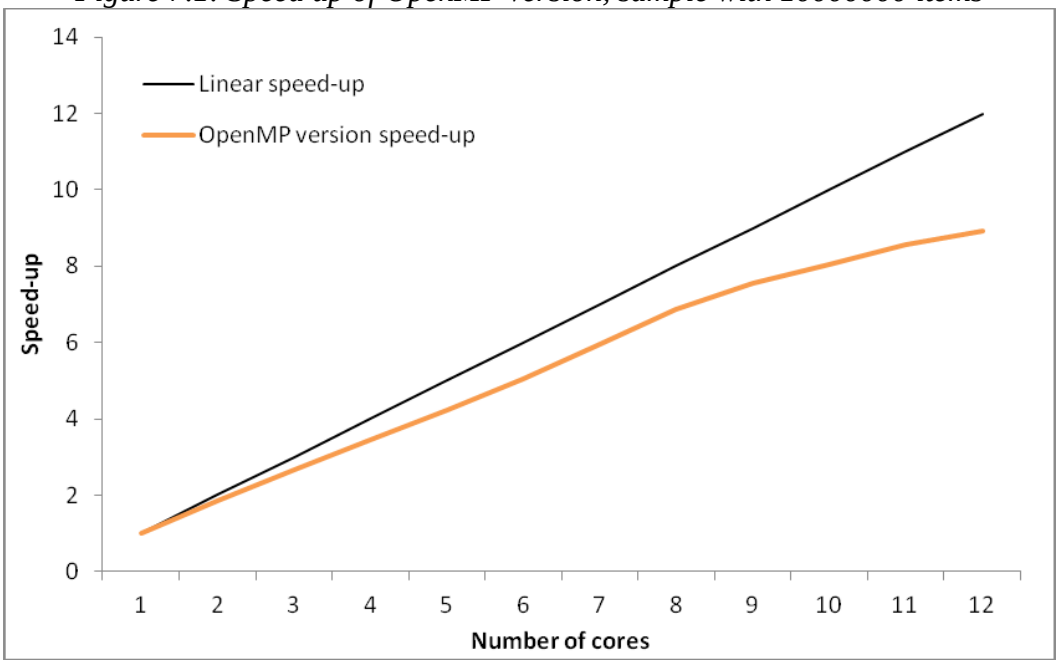


Figure 7.1 shows the graphical representation of the speed-up of the OpenMP tests running, when the sample contains 10,000,000 items and the linear speed-up. The speed-

up of the OpenMP version tends to be linear, up to eight threads, when the speed-up falls significantly.

Figure 7.2: Speed up of OpenMP version, sample with 5000000 items

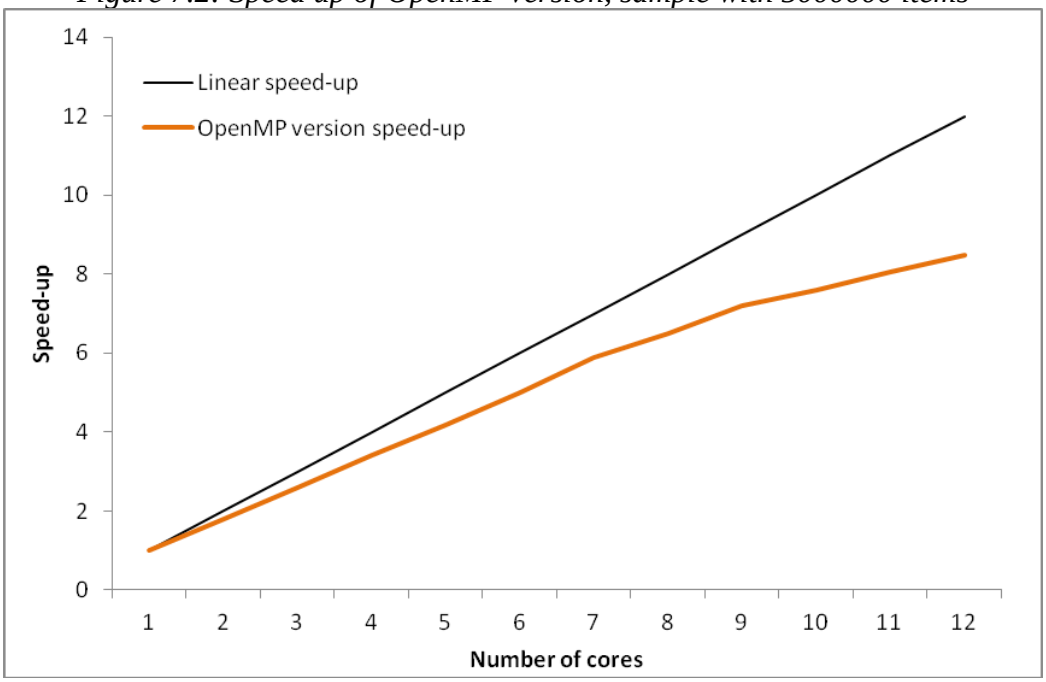


Figure 7.2 shows the curve of speed-up for OpenMP, when the sample contains 5,000,000 items. The speed-up tends to be linear when the application is run in two up to eight threads. The number of eight parallel threads is crucial, as the speed-up tends to increase fewer when the application is running in eight, up to twelve nodes.

Tests of performance, when the sample contains wind speed data of the 2013 in SH-WS and PG-WS, are shown in table 9.

Table 9 : Results of OpenMP version, sample with wind speed data during 2014

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 17520 data of SH-WS			The sample contains 105120 data of PG-WS		
1	34.92	1	1	70.11	1	1
2	19.03	1.83	0.92	37.91	1.85	0.92

3	13.07	2.67	0.89	25.77	2.72	0.91
4	10.32	3.38	0.85	20.05	3.50	0.87
5	8.24	4.24	0.85	16.20	4.33	0.87
6	6.86	5.09	0.85	13.30	5.27	0.88
7	5.80	6.02	0.86	11.36	6.17	0.88
8	5.01	6.97	0.87	9.99	7.02	0.88
9	4.52	7.72	0.86	8.95	7.84	0.87
10	4.04	8.65	0.86	8.07	8.68	0.87
11	3.74	9.34	0.85	7.49	9.35	0.85
12	3.55	9.83	0.82	7.17	9.78	0.81

The first column of the table 9 contains the number of cores used in the tests.
The second, third and the forth columns of the table contain the results when the sample is wind speeds during 2014 in SH-WS.

The fifth, sixth and seventh columns contain the results, when the sample is wind speed data during 2014 in PG-WS.

The sample of the wind speed data in SH-WS in 2014 contains 17520 items. The serial version needs 34.92 seconds to estimate the result for this data set. The OpenMP version performs a speed-up up to 1.83 by adding the second thread. While running the OpenMP version with 12 threads, 3.55 seconds is needed to estimate the result of the parameter estimation. The maximum speed-up is 9.83, and is obtained is when executing the program in 12 parallel threads. The efficiency of the OpenMP version, estimating the parameters for this sample, is gradually decreased while adding the number of threads.

When the sample is the wind speed data of PG-WS, during 2014 the sample contains 105120 data. The serial version takes 70.11 seconds to get the parameters of Weibull distribution for these data. The speed-up increases to 1.85 while running in two threads. Running the OpenMP version with 12 threads, it takes 7.17 seconds to calculate the results of the parameter estimation. The maximum speed-up is 9.78, and is obtained is when executing the program in 12 threads. The efficiency is stable up to ten threads. While running in eleven or twelve threads, the efficiency is significantly decreasing.

Figure 7.3: Speed up of OpenMP version, sample with wind speed data in SH-WS during 2014

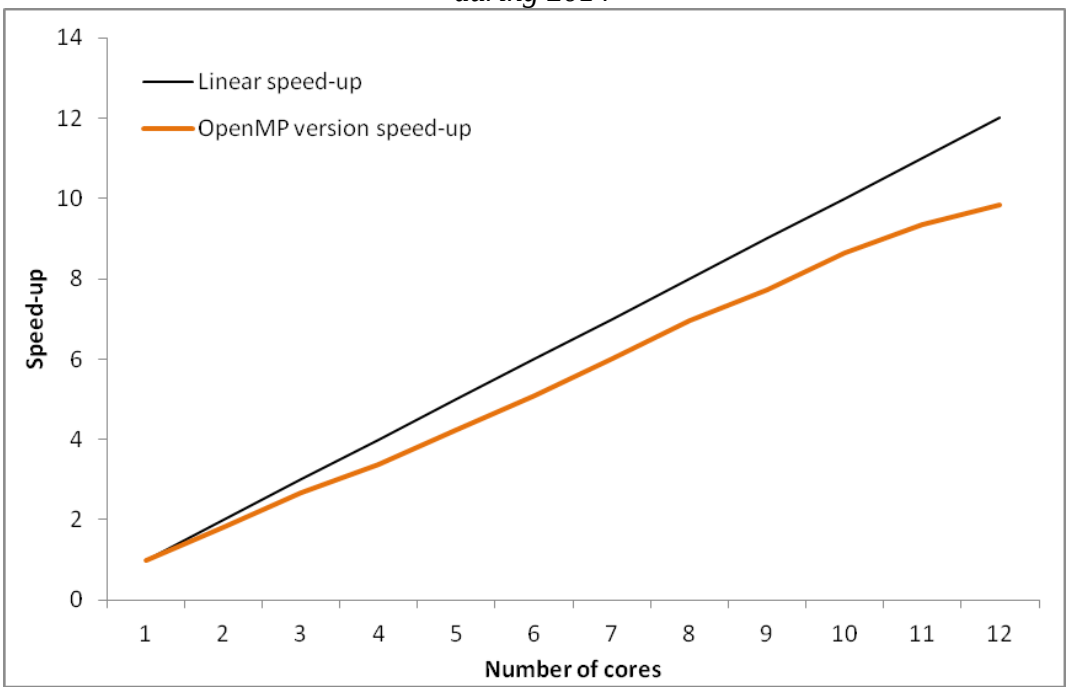


Figure 7.3 shows the curve of speed-up for OpenMP, when the sample contains wind speed data during 2014 in Shkodër. The speed-up tends to be linear almost during the entire curve, from two up to twelve nodes.

Figure 7.4: Speed up of OpenMP version, sample with wind speed data in PG-WS during 2014

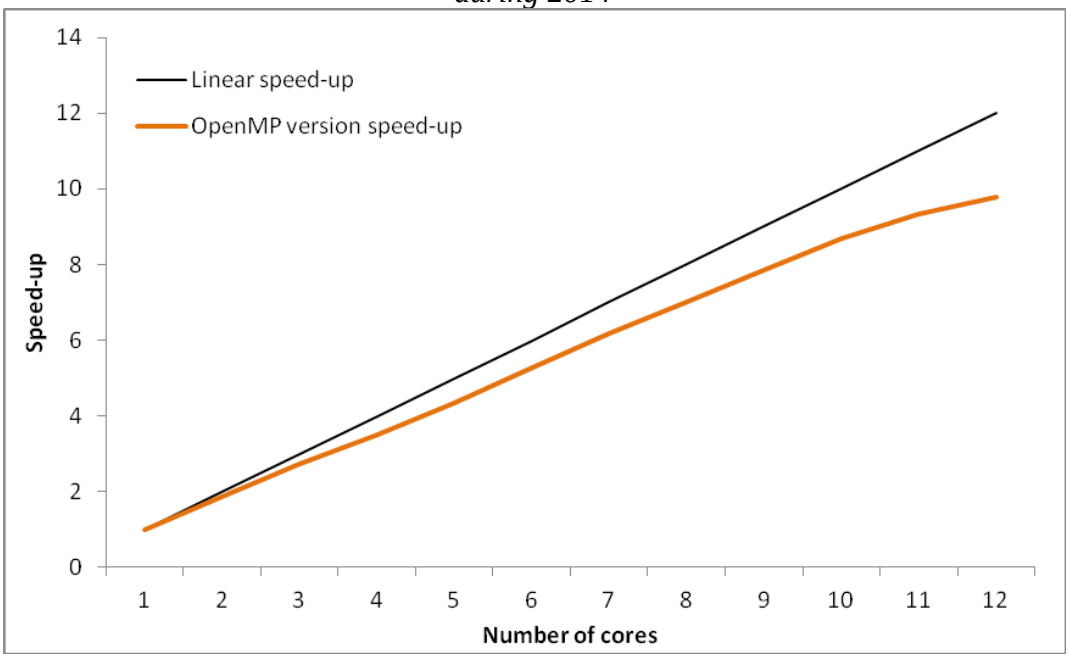


Figure 7.4 shows the curve of speed-up for OpenMP, when the sample contains wind speed data during 2014 in Podgorica. The speed-up tends to be linear almost the entire curve.

Tests of performance when the sample contains wind speed data of y2013 and 2014 of PG and SH-WS, are shown in the table 10.

Table 10 : Results of OpenMP version, sample with wind speed data during 2013-14

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 35040 data of SH-WS			The sample contains 210240 data of PG-WS		
1	66.71	1	1.00	136.76	1	1.00
2	36.26	1.84	0.92	75.39	1.81	0.91
3	24.84	2.69	0.90	53.36	2.56	0.85

4	19.65	3.39	0.85	39.91	3.43	0.86
5	15.65	4.26	0.85	31.71	4.31	0.86
6	13.10	5.09	0.85	26.03	5.25	0.88
7	11.11	6.00	0.86	22.33	6.12	0.87
8	9.76	6.84	0.85	19.50	7.01	0.88
9	8.72	7.65	0.85	17.56	7.79	0.87
10	7.89	8.45	0.85	15.86	8.62	0.86
11	7.28	9.16	0.83	14.63	9.34	0.85
12	6.93	9.62	0.80	13.97	9.79	0.82

The first column of the table 10 contains the numbers of cores involved in the tests. The second, third and forth columns contain metrics of the tests performed on wind speed data of SH-WS. The serial version takes 66.71 seconds to get the parameters of Weibull distribution for this data set. The speed-up increases to 1.84 by adding the second process. The parallel version in OpenMP, running in 12 threads, takes 6.93 seconds to calculate the results of the parameter estimation. The speed up is 9.62, and is obtained is when executing the program in 12 cores. The efficiency of the OpenMP version, estimating the parameters for this sample, is unstable up to eight processors, but is dramatically decreased after adding the ninth processor.

The fifth, sixth and last column of table 10 contains data of the tests performed on wind speed in PG-WS, during 2013-14. The serial version of the application takes 136.76 seconds to estimate the parameters for this sample. Using the OpenMP programming, the speed-up achieved is up to 9.79. This is the case when the calculations are made from 12 threads in parallel.

Figure 7.5: Speed up of OpenMP version, sample with wind speed data in SH-WS during 2013-14

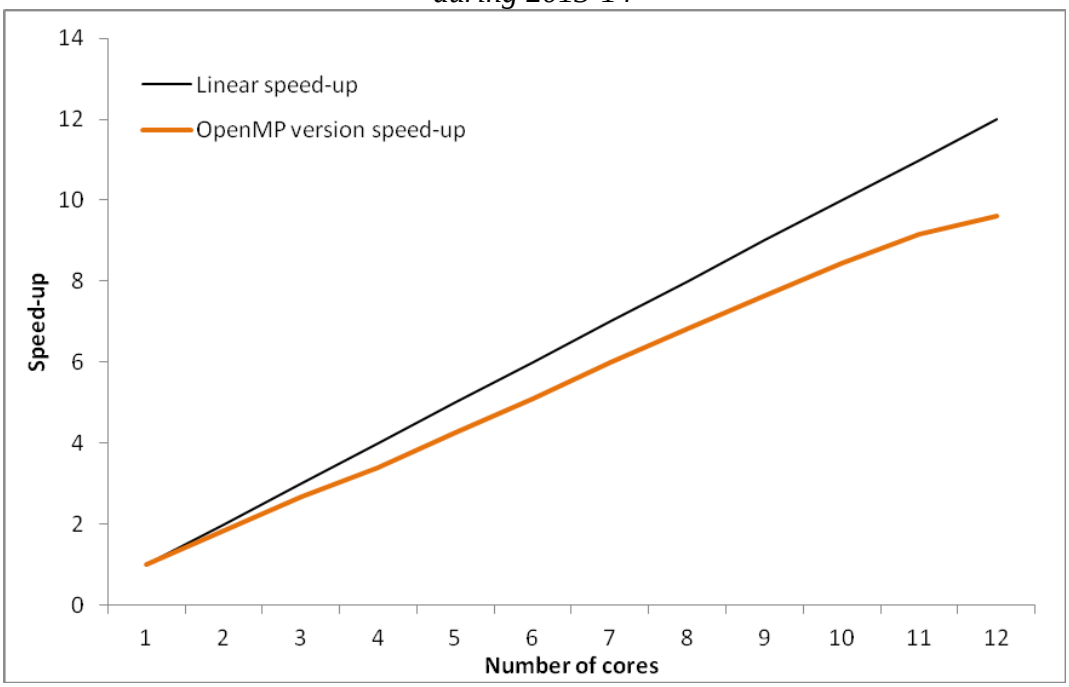


Figure 7.5 shows the curve of speed-up for OpenMP, when the sample contains wind speed data during 2013-14 in Shkodër. The speed-up tends to be linear almost during all the tests, while the application is running in two up to twelve nodes.

Figure 7.6: Speed up of OpenMP version, sample with wind speed data in PG-WS during 2013-14

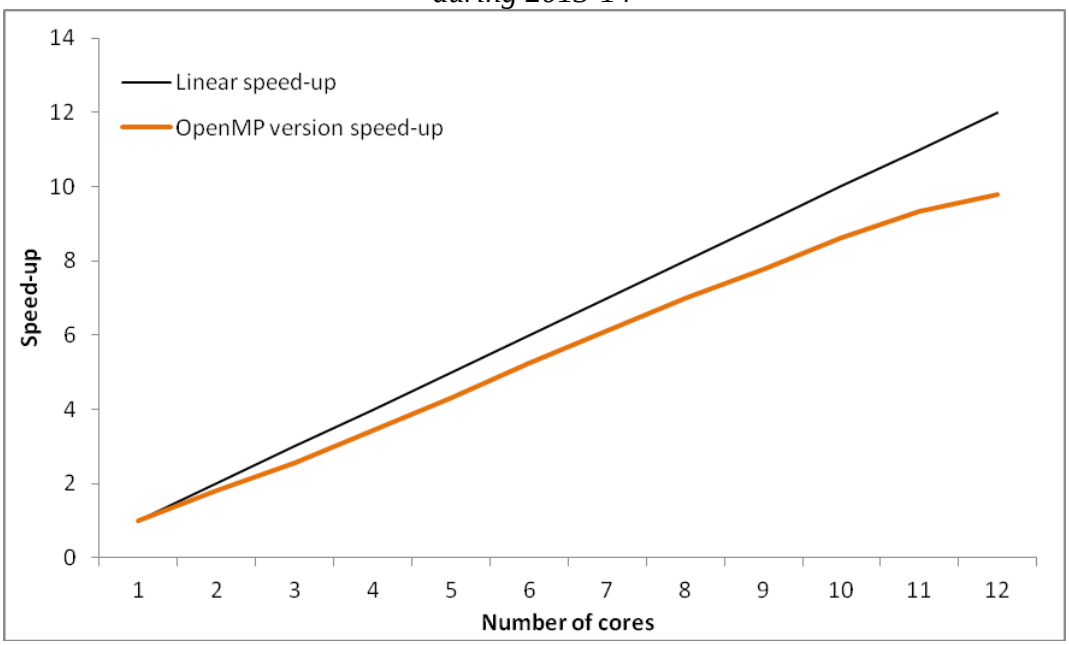


Figure 7.6 shows the curve of speed-up for OpenMP, when the sample contains wind speed data during 2013-14 in Podgorica. The speed-up tends to be linear when the application runs up to eight nodes. Adding new nodes, next to eight, does improve the speed-up, but not in the same tend.

It can be concluded for the OpenMP version, based on the results of the tests, the program exhibits a good performance while running in 2 up to 12 cores. The best speed-up achieved is 9.83, and it is reached while the OpenMP program is running in 12 cores, estimating the sample with wind speed data measured in one year, in Shkodër, which is the smallest sample. While running in 12 cores, estimating this sample, the efficiency reaches the value 0.82.

OpenMP enables the specifications of the program at a higher abstraction than Pthreads. It is not needed to write explicitly the synchronization code, which is the most difficult task in writing an efficient multi-thread program.

After the estimation, the parameters of Weibull distribution for concrete wind speed data, in Podgorica and in Shkodër are as following:

- The sample containing wind speed data measured in Podgorica, during 2013-14, has the scale parameter b=2.638411 and shape parameter c=3.548239.
- The sample containing wind speed data measured in Shkodër, during 2013-14, has the scale parameter b=2.948302 and shape parameter c=3.239495.
- The sample containing wind speed data measured in Podgorica, during 2014 has the scale parameter b=2.612460 and shape parameter c=3.537527.
- The sample containing wind speed data measured in Shkodër, during 2014, has the scale parameter b=2.848374 and shape parameter c=3.309258.

Chapter 8

Hybrid MPI/OpenMP implementation of Weibull distribution parameter estimation

8.1 Analysis

Clusters of shared nodes are becoming more and more a dominant parallel architecture. This fact imposes the need to consider the hybrid MPI/OpenMP approach. A combination of MPI and OpenMP constructs a hybrid program that achieves multiple levels of parallelism and reduces the communication overhead of MPI. Multicore clusters offer a natural programming paradigm for these hybrid programs. In this environment, OpenMP is used with the data sharing and MPI is used with the communication between nodes. The hybrid approach can help reduce the demand for resources such as memory and network [66].

For certain class of applications with easily exploitable multi-level parallelism, the hybrid model can also potentially reduce the application development effort also.

But when does hybridization make sense? There are some cases when the programmer should choose the hybrid version MPI/ OpenMP. One of such cases is when the programmer aims to scale the shared memory architecture for use on multiple SMP nodes in a cluster. Another case is when the programmer wants to reduce an MPI program's sensitivity avoiding the bound in communication. A last argument is when the programmer wants to design a parallel program from scratch to maximize the utilization of a distributed memory machine consisting of individual SMP nodes.

The analysis consists of defining the following steps in the multilevel parallelization process:

1. Identify where the MPI communication occurs
2. Identify the loops where iterations can be executed independently. In cases when there is data dependence in a loop, modify the loop to make the data independent.
3. Insert the OpenMP directives to ensure large granularity.

4. Ensure that MPI communications occur out of OpenMP parallel regions.

While implementing a hybrid version, the chances to gain from both versions are promising. MPI code helps the hybrid version to make easier the communication between nodes. Also, it facilitates the efficient scatter operations and reduction operations between nodes. MPI helps in sending messages which contains complex data structures. On the other way, OpenMP shows a very good performance in threading implementation. It offers the interface for the concurrent running threads in each node of shared memory; this is much more efficient than using MPI interface. The synchronization is made implicitly, on each SMP node. This is a solution in decreasing the overhead caused by the message passing synchronizations.

8.2. Implementation

In the hybrid MPI/OpenMP version, each MPI process executes the initialization step on each node of a multi-core cluster. Meanwhile, OpenMP is used in parallelization at the loop level. During the implementation phase, there are some issues to be considered. One of them is the care that the programmer should take,to find the right balance of MPI computation and OpenMP tasks. The shared memory does the main part of the work. A key in the analysis is the overhead of communication among SMP nodes. The implementation should be scaled and the running time in one single node should be minimized. The SMP nodes should maximize the number of threads during the computation parts. MPI implementation is to show better performance when a small number of cores communicate with each other, sending complex data structures; therefore, many small messages will lead to an unnecessary communication.

Code 10: Implementation of hybrid version MPI/OpenMP, main function

```
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define TOTAL_THREADS 4
/* Each MPI process creates a distinct
OpenMP master thread;
One MPI processes per node*/
int main (int argc, char *argv[]) {
  int pr,process_id, counter;
/* set number of threads */
  omp_set_num_threads(TOTAL_THREADS);
/* initialize MPI */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&pr);
  MPI_Comm_rank(MPI_COMM_WORLD,&process_id);
  // The parallel OpenMP routine executed by each
MPI process
  #pragma omp parallel reduction(int counter) {
      counter = omp_get_num_threads();
  }
/* Each MPI process prints a result */
  printf("%d\n",counter);
  MPI_Finalize();
  return 0;
}
```

In the implementation example in Code 10, it a common main function of a hybrid MPI/OpenMP is shown. In this example, a single MPI process is launched in each SMP node in the cluster. Number of threads is set as a constant number of four threads. This is set with the instruction `omp_set_num_threads (N)`. Every process creates N threads on each SMP node. The master thread on MPI communicates with the master thread on all other nodes. The threads that belong to each process continue until another sync point completes.

8.3. Results

The hybrid MPI/OpenMP version is tested for these numbers of cores:

- four cores, composed of two processes, each of them has two threads
- six cores, formed by three processes, each of them has two threads
- nine cores, composed of three processes each of them has three threads
- twelve cores, formed by three processes, each of them has four threads

Table 11 : Results of hybrid MPI/OpenMP version, sample with wind speed data during 2014

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 105120 data of PG-WS			The sample contains 17520 data of SH-WS		
1	70.11	1	1	34.92	1	1
4	17.86	3.93	0.98	8.96	3.90	0.97
6	12.21	5.74	0.96	6.15	5.68	0.95
9	8.39	8.35	0.93	4.21	8.30	0.92
12	6.99	10.03	0.84	3.49	10.01	0.83

Table 11 shows the results of the tests performed on the sample of wind speed data during 2014 in Podgorica and in Shkodër.

For the sample with data of PG-WS, the serial version takes 70.11 seconds to get the parameters of Weibull distribution for this data set.

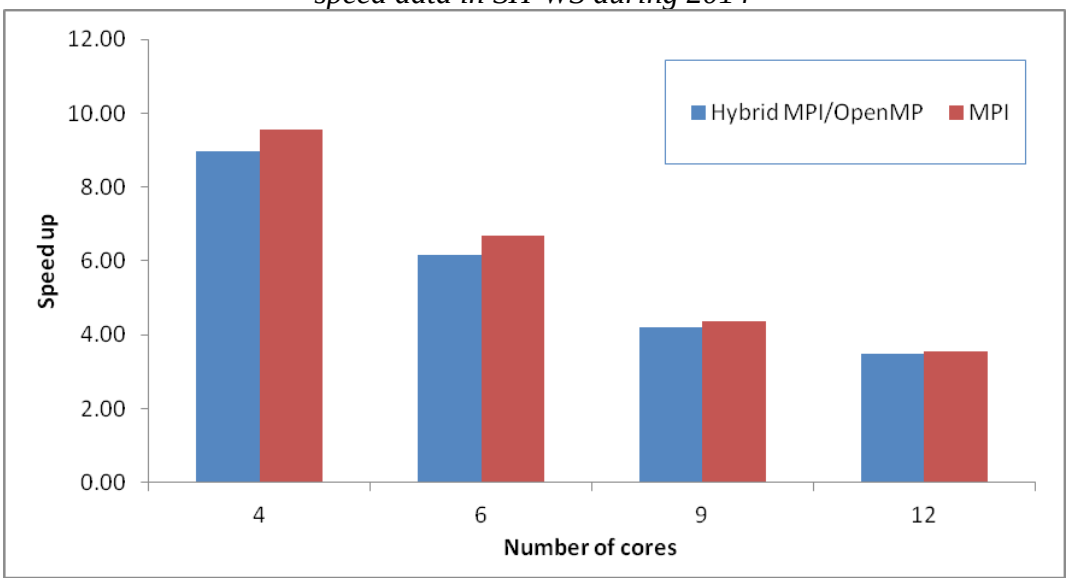
Results show that the hybrid version achieves a speed-up 3.93 while running the application in two processes each one with two threads. When the hybrid version runs in

three processes each with four threads, it needs 6.99 seconds to calculate the result. The maximum speed-up is 10.03, while the application is running in three processes and four threads. The efficiency of the hybrid version is very high during all tests.

The sample of wind speed measured in SH-WS, contains 17520 items. The serial version of the application takes 39.92 seconds to get the parameters of Weibull distribution for this data set. The hybrid version of the application takes 8.96 seconds to produce the result, in case when it is run in two processes each with two threads. Running this hybrid version in 12 cores takes 3.49 to produce the result. This test results in maximum speed 10.01.

Figure 10.1 represents a graphical comparison between the speed-up of the Hybrid MPI/OpenMp version, and the pure MPI version, when the sample is the wind speed in Shkodër, during 2014.

Figure 8.1: Comparison of hybrid version and pure MPI version, sample with wind speed data in SH-WS during 2014



The graphic shows that while running in 4 cores, the difference between two values compared is higher and it decreases while the number of cores is larger. This is interpreted, as the number of cores is twelve, three processes each with four threads, the

communication between processes is not influencing the overall performance. In this case, the communication is almost the same as the thread creation overhead.

Figure 8.2: Comparison of hybrid MPI/OpenMp and pure MPI version, sample with wind speed data in PG-WS during 2014

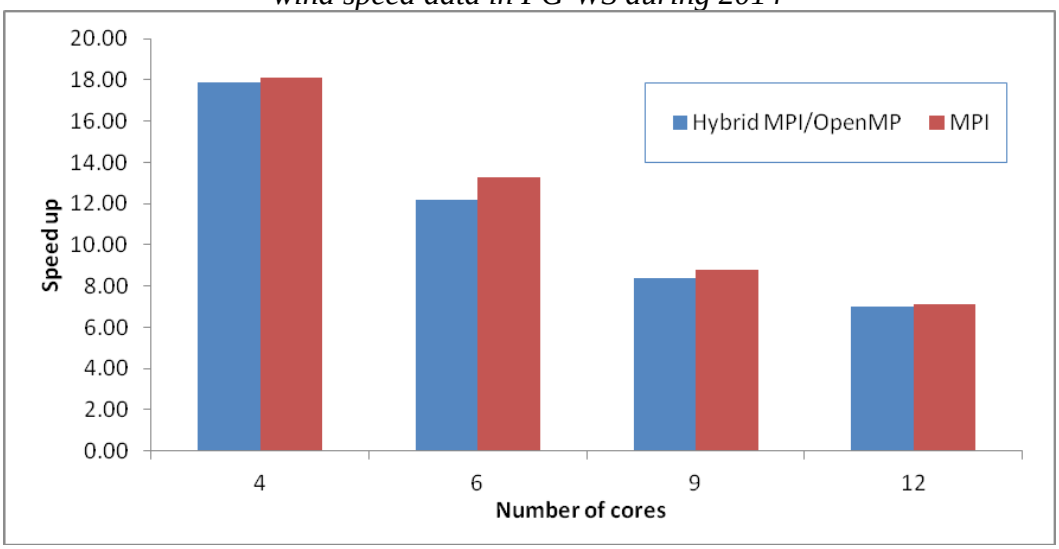


Figure 8.2 gives a comparison of the speed-up of hybrid MPI/OpenMp version and the pure MPI version. While the applications are run in 4 cores, the speed-ups of these versions are very close. When the program runs in six cores and nine cores, the hybrid version performs relatively better than the pure MPI version. Hybrid version is faster as the communication between processes is processed faster than in the pure MPI version. While the applications are run in twelve cores, the speed-up again is very close in value in both versions.

Table 12 contains the results of tests of performance of the hybrid implementation, when the sample contains wind speed data of the year 2013-14 of SH-WS and PG-WS.

Table 12 : Results of hybrid MPI/OpenMP version, sample with wind speed data during 2013-14

No of cores	Time in seconds	Speed-up	Efficiency	Time in seconds	Speed-up	Efficiency
	The sample contains 210240 data of PG-WS			The sample contains 35040 data of SH-WS		
1	136.76	1	1	66.71	1	1
4	34.64	3.95	0.99	16.92	3.94	0.99
6	23.64	5.78	0.96	11.59	5.76	0.96
9	16.23	8.42	0.94	7.96	8.38	0.93
12	14.11	9.69	0.81	6.61	10.08	0.84

Each row of the table 12 contains the results of the tests done with hybrid version, in different number of cores.

The first column of the table contains the number of the cores that run the application. The second, third and forth columns are the results while testing the application with real wind speed data in Podgorica during 2013-14. This sample contains 35040 data. The fifth, sixth and last columns are the results while testing the application with real wind speed data in Shkodër, during 2013-14. This second sample contains 210240 data.

While estimating the parameters, running in twelve cores, hybrid version takes 14.11 seconds for the wind speed data in Podgorica, and 6.61 seconds for a smaller sample with wind speed data in Shkodër.

The best speed-up is achieved while running in twelve cores, which is 9.69 in first sample test, and 10.08 in second sample tests. The performance is slightly better while estimating the parameters for a smaller sample.

Figure 8.3: Comparison of hybrid MPI/OpenMP and pure MPI version, sample with wind speed data in Shkodër during 2013-14

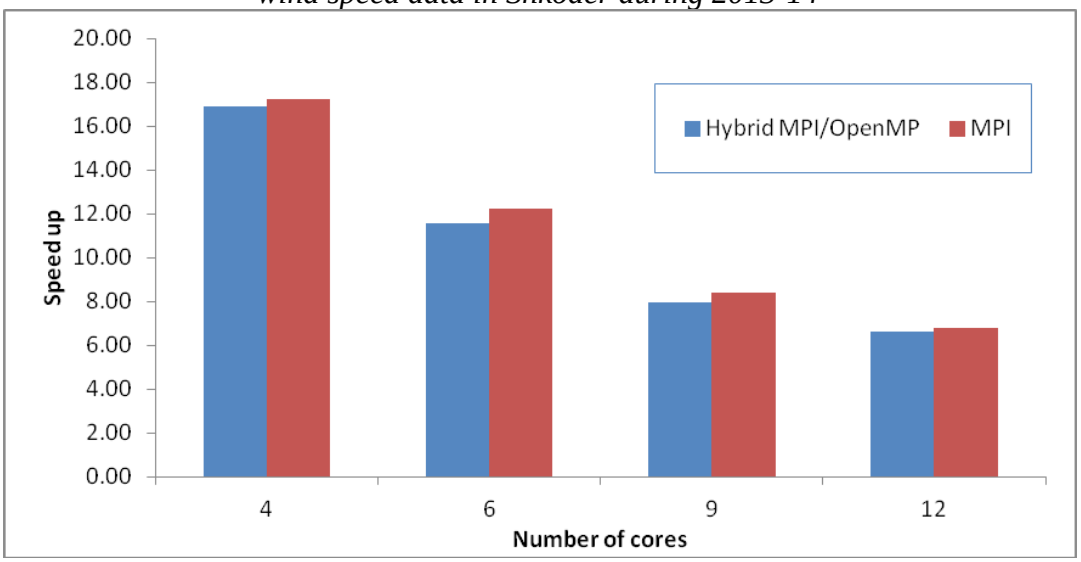


Figure 8.3 gives a comparison of the speed-up of hybrid MPI/OpenMP version and the pure MPI version. While the applications are run in four and in twelve cores, the speed-up of these versions is very close. When the program runs in six cores and nine cores, the hybrid version performs relatively better than the pure MPI version. A reason for this might be the fact that the hybrid version is faster as the communication between processes is processed faster than in the pure MPI version.

Figure 8.4: Comparison of hybrid MPI/OpenMp version, sample with wind speed data in Podgorica during 2013-14

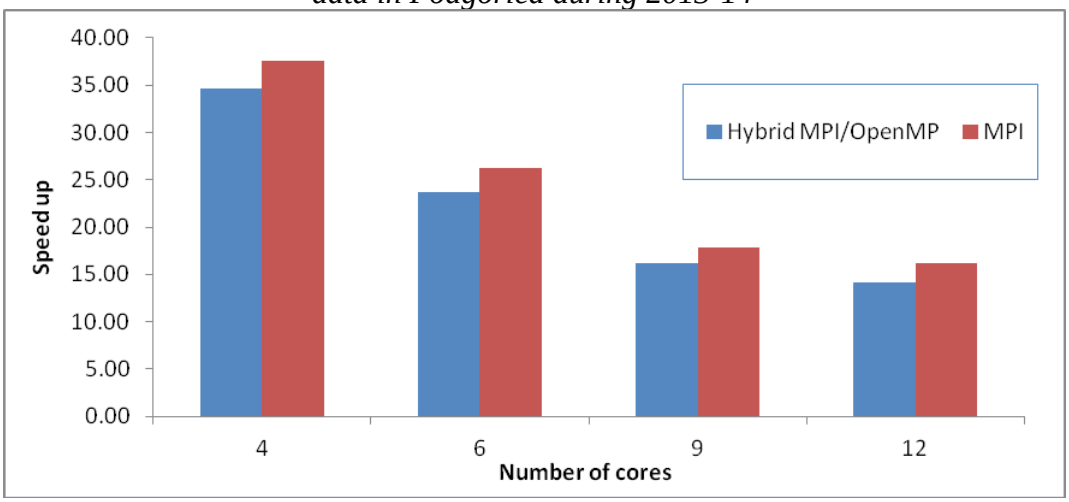


Figure 8.4 gives a graphical comparison of the speed-up of hybrid MPI/OpenMP version and the pure MPI version, when the sample is from PG-WS. When the applications are run in nine cores, the speed-ups of these versions are very close. A reason for this might be the fact that this might be a good balance between the communication overhead and the size of the sample.

It can be concluded that the hybrid version MPI/OpenMP shows a satisfactory the speed-up and efficiency, especially when the sample is smaller in size. The best achieved speed-up is 10.08, and it is reached while the hybrid program is running in 12 cores, while estimating the parameters of the sample containing wind speed data in Shkodër, in 2 years. This sample is relatively small, with 35040 items.

After the estimation, the parameters of Weibull distribution for concrete wind speed data, in Podgorica and in Shkodër are as following:

- The sample containing wind speed data measured in Podgorica, during 2013-14, has the scale parameter b=2.638411 and shape parameter c=3.548239.
- The sample containing wind speed data measured in Shkodër, during 2013-14, has the scale parameter b=2.948302 and shape parameter c=3.239495.

- The sample containing wind speed data measured in Podgorica, during 2014 has the scale parameter b=2.612460 and shape parameter c=3.537527.
- The sample containing wind speed data measured in Shkodër, during 2014, has the scale parameter b=2.848374 and shape parameter c=3.309258.

Chapter 9

Conclusions

The thesis covers the implementation of the Weibull parameter estimation, written in three parallel environments. Results of the performed tests show that the efficiency of the program based on Posix threads is better than the other implementations used in this research. During the design and implementation phase, some problems are pointed out. One of them is the effort the programmer makes to coordinate the communication between several threads. Another difficulty is the challenge to overcome latency problems of the architecture used in the implementation of these parallel programs.

In high-performance computing, the role of distributed memory is very important, but programmers still rely on the basic parallel option, named MPI. On the shared memory platform approaches it works well. Both OpenMP and Posix threads can provide a satisfactory performance. Their problem might be the requirement of manual optimization.

In the implemented MPI version of the program, there usually exists only one communicator. However, this isn't a necessary condition. MPI offers the possibility that different nodes can be organized into groups, where each group can use the same communicator. The groups can use all-to-all functions and broadcast messages, like the implementation in the cluster. But this way of organizing the nodes in the MPI implementation makes more complex the parallelism and will not have any good speed-up effect.

The MPI model has two main advantages. The most obvious to programmers is that the code written using MPI is highly portable. MPI programming model does not require special hardware for the program execution. Virtually, any collection of computers can execute a parallel program in MPI. The second main advantage of MPI is that it provides the programmer an explicit control over the memory location. Since the performance is often determined by the memory access, this ability allows the programmer to achieve higher performance of the program.

While writing the MPI version of the Weibull parameter estimation, it is important to study carefully the order of communication between processes. It considerably affects the speed-up of the program.

It can be concluded that the parallel version implemented in MPI increases the speed-up, compared with the serial version, especially when the set of data is small. While calculating the Weibull distribution of the smallest sample, the MPI version shows its best performance. It can be concluded that the MPI program shows its best performance, including speed-up and efficiency, when the sample contains real wind speed items.

Unlike MPI, Pthreads and OpenMP implement the parallelization by using threads. The advantage of using threads instead of processes is that switching between threads is much faster than switching between processes. Also, implementing the communication between two threads is much easier than the communication between two processes.

One main difficulty that arises while programming in threads' level is the memory management because threads use the same memory space. Another disadvantage of using threads is that they all have to be run on the same machine.

The Pthread implementation improves the speed-up and efficiency of the program. The Pthread version exhibits its best performance in cases when the sample is wider in size.
OpenMP differs from Pthreads in some significant ways.

While Pthreads is implemented as one single library, OpenMP is implemented as a set of libraries and a set of compiler directives. This version requires its own compiler support to interpret such directives. In Pthreads there is no structure or any a priori relationship between threads.

OpenMP enables the specifications of the program at a higher abstraction level than Pthreads. During the implementation of the parallel version in OpenMP, it is not needed to write explicitly the synchronization code. This simplifies the implementation phase. This version shows good performance in speed-up and efficiency, too.

The analysis of the hybridization using MPI and OpenMP concludes that OpenMP facilitates the shared memory programming across clustered nodes. The MPI facilitates the communication between nodes and enables sending of complex data structures. The communications computing ratio on each node should be minimized to improve the scaling features of the hybrid version. During the implementation, the greatest benefits are

observed when an application is redesigned from scratch. The programmer has to be careful of the whole program correctness and of efficient communications.

The results confirm the hypothesis suggested at the beginning of this research, that the methods which implement the parallelism using the parallel threads, will achieve a better speed-up than the methods that use the parallel processes. It proposes to implement the Weibull distribution parameter estimation in parallel Pthreads. The cost is the effort of the programmer to achieve a good communication between several threads, but it is worth the time because it (considerably, significantly) speeds up the estimation process. Especially in cases when the sample is wider in size, the Pthread version shows its best performance.

In the future perspective, communication is the key. It is important in runtime performance, in managing the software complexity and in organizing the hardware heterogeneity. A challenge to the software engineers will be the development of the communication-oriented tools and design techniques, analysis, and implementation of parallel programs.

Now, many important applications are data-intensive, but unfortunately, the growing gap in performance between memory and CPU means they do not fully utilize the current hardware. It was just over a decade ago that multicore processors appeared on the market. Meanwhile, we have seen a shift to all multicore processors, ranging from cellphone processors to high-end processors for supercomputers. This is a huge change in the landscape and a trend that will continue.

In the future, hardware will be mostly designed for parallel computing, with a wide variety of innovations to make the parallel computing more affordable and higher in performance. Software efforts will range from operating systems to tools and to language standards, which we can build upon. It is supposed that we are going to see an unleashing of computing power we have never seen before. This era will be defined by the combination of computer power originating from parallel computing, with access for everyone via the cloud, and an equipped and oriented industry to take all the advantage of it.

List of Abbreviations

API	Application Programming Interface
ASP	Application Service Provisioning
CDF	Cumulative Distribution Function
COMA	Cache only Memory Architecture
CPU	Central Processing Unit
DF	Density Function
HR	Hazard Rate
HZ-WS	HerzegNovi Weather Station
IEEE	The Institute of Electrical and Electronics Engineers
IT	Information Technology
LAN	Local Area Network
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
ML	Maximum Likelihood
MPI	Message Passing Interface
MPIF	Message Passing Interface Forum
NUMA	Non-uniform Memory Access
OpenMP	Open Multi-Processing
PC	Program Counter
PG-WS	Podgorica Weather Station
POSIX.	Portable Operating System Interface for Unix
SH-WS	Shkodra Weather Station
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMP	Symmetric Multi-processor
SPMD	Single Program Multiple Data
TLP	Thread-level Parallelism
UMA	Uniform Memory Access
WECS	Wind Energy Conversion System

Bibliography

[1] R.Trobec, M.Vajtersic and P. Zinterhof, “Grid computing” in Parallel Computing Numerics, Applications, and Trends, Springer, 2009, pp.117-143

[2] A.Tannenbaum, M. Van Steen, “Distributed Systems: Principles and Paradigms”, CreateSpace, 2016.

[3] D.Taniar, C.H.C.Leung, W.Rahayu and S.Goel, “Grid concurrency control” in High-Performance Parallel Database Processing and Grid Databases, Wiley, 2008, pp. 321-341

[4] J.L.Hennessy and D.A.Patterson, “Multiprocessor and thread level parallelism” in Computer Architecture, a quantitative approach fourth edition, Morgan Kaufmann Publication, 2007, pp.196-203

[5] A.Grama, A. Gupta, G. Karypis and V. Kumar “Analytical modelling of parallel programs” in Introduction to Parallel Computing , Second Edition, Addison Wesley, 2003, pp. 178-218

[6] M. Herlihy and N.Shavit, “Futures, scheduling and work distribution” in The Art of Multiprocessor Programming, Morgan Kaufmann Publishers, 2008, pp. 390-418

[7] T. G. Mattson, B.A. Sanders and B. L. Massingill , “The finding concurrency design space” in Patterns for Parallel Programming, Software Patterns Series, Addison Wesley, 2004, pp. 30-60

[8] H. El-Rewini and M. Abd-El-Barr, “Performance analysis of multiprocessors architecture” in Advanced computer architecture and parallel processing, John Wiley & Sons, 2005, pp. 51-77

[9] E.J. Kontoghiorghes, “Statistical applications” in Handbook of parallel computing and statistics, Taylor & Francis Group, LLC, 2006, pp. 285-505

[10] R.Nisbet, J.Elder, G.Miner, “Handbook of statistical analysis and data mining applications”, Elsevier, 2009

[11] H. Rinne , “Parameter estimation, maximum likelihood approaches” in Weibull Distribution A Handbook, CRC Press, 2009, pp.402-456

[12] H.Hirose, “Maximum likelihood parameter estimation in the extended Weibull distribution and its applications to breakdown voltage estimation” in IEEE Transactions on Dielectrics and Electrical Insulation, vol 9, no.4, 2002, pp. 524-536

[13] H.Hirose, “Maximum likelihood estimation in the 3-parameter Weibull distribution. A look through the generalized extreme-volume distribution” in IEEE Transactions on Dielectrics and Electrical Insulation, vol 3, no.1, 1996, pp. 43-55

[14] J.R.Hoobs, Moore, H.Albert, Miller and M. Robert, “ Minimum-distance estimation of the parameters of the 3-parameter Weibull distribution” in IEEE Transactions on Reliability, vol. R-34, no.5, 1985, pp.495-496

[15] H.Hirose, “Maximum likelihood estimation in the 3-parameter Weibull distribution: a look through the generalized extreme-value distribution” in IEEE Transactions on Dielectrics and Electrical Insulation, vol.14, no.1, 2007, pp.257-260

[16] O.Osarumwense, N.C. Rose, " Parameters estimation methods of the Weibull distribution: A comparative study" in Elixir International Journal, vol 69, 2014, pp.23177-23184

[17] Y.Lei, " Evaluation of three methods for estimating the Weibull distribution parameters of Chinese pine (Pinus tabulaeformis)" in Journal of Forest Science, vol 54, 2008, pp. 566–571

[18] M. Snir, S. Otto, S. Huss-Lederman, D.Walker and J.Dongarra, “Point to point communication “ in MPI, the complete reference, The MIT Press, 1996, pp.15-89

[19] W. Gropp, E.Lusk and R.Thakur,”Parallel I/O” in Using MPI-2 Advanced Features of the Message-Passing Interface, The MIT Press, 1999, pp.51-119

[20] D.Butenhof, “Synchronization” in Programming with Posix threads, Addison-Wesley, 1997, pp. 45-97

[21] B.Lewis and D. J. Berg,”Concepts” in PThreads Primer, SunSoft Press, 1996, pp. 35-57

[22] R.Chandra, L.Dagum, D.Kohr, D.Maydan, J. McDonald, R.Menon, “Parallel programming in OpenMP”, Morgan Kaufmann Publishers, 2001

[23] S.F.McGinn, R.E.Shaw, “Parallel Gaussian elimination using OpenMP and MPI” in High Performance Computing Systems and Applications, 2002, pp. 169-173

[24] T.Rauber and G. Runger,”Message passing programming” in Parallel Programming For Multicore and Cluster Systems , Springer, 2011, pp.198-257

[25] D. A. Lewine, ”Developing POSIX applications” in POSIX Programmer's Guide, O'Reilly & Associates, 1991, pp.14-39

[26] S.F.McGinn, R.E.Shaw, “Parallel Gaussian elimination using OpenMP and MPI” in High Performance Computing Systems and Applications, 2002, pp. 169-173

[27] H.Xuan, W.Tong, Y.Hu, Z.Gong, Y.Hou, “Hybrid programming implementation of BiCG method on SMP Cluster architecture” in IEEE International Conference on Computer and Information Science, 2012, pp. 470-475

[28] H.Xuan, W.Tong, Y.Hu, Z.Gong, Y.Hou, “Comparison of parallel programming models for multi-core architectures” in IEEE International Parallel & Distributed Processing Symposium 2011, pp. 1675-1682

[29] I. S. Dhillon and D. S. Modha “A Data-Clustering Algorithm on Distributed Memory Multiprocessors” in Large-Scale Parallel Data Mining Lecture Notes in Computer Science ,vol 1759, 2000, pp 245-260

[30] T.Ungerer, B.Robic and J.Silc, “Multithreaded Processors” in The computer journal, vol.45,no. 3, 2002, pp. 320-348

[31] P.E. Lopez-de-Teruel, A.Ruiz, J.M.Garcia, “A parallel algorithm for tracking of segments in noisy edge images”, in 15th International Conference on Pattern Recognition, 2000, vol.4, pp. 807-811

[32] M. R. Patel, “Wind power” in Wind and Solar Power Systems, CRC Press 1999, pp. 24-33

[33] C.G.Justus, W.R.Hargraves, A.Mikhail and D.Graber, “Methods for estimting wind speed frequency distributions” in Journal of applied meteorology, volume 17, 1978, pp. 350-354

[34] J. Hetzer, D. C. Yu and K.Bhattarai “An Economic Dispatch Model Incorporating Wind Power” in IEEE Transactions on energy conversion, vol. 23, no. 2, 2008, pp.603-611

[35] A. Zaharim, S.K. Najid, A.M.Razali and K.Sopian “The Suitability Of Statistical Distribution In Fitting Wind Speed Data” in WSEAS Transactions on Mathematics, No. 12, Vol. 7, 2008, pp. 718-727

[36] J. Nimbal, R. L. Naik and S.H. Jangamshetti, “Wind Data Analysis: A Case Study”, in Power, Signals, Controls and Computation, 2012, pp. 1-5

[37] Z. Qin, W. Li and X. Xiong, “Generation System Reliability Evaluation Incorporating Correlations of Wind Speeds With Different Distributions” in IEEE transactions on power systems, 2012, pp. 551 – 558

[38] B.G. Kumaraswamy, B.K.Keshevan, Y.T.Ravikiran, “Analysis of seasonal wind speed and wind power density distribution in Aimangala wind form at Chitradurga Karnataka using two parameter weibull distribution function” in IEEE Power and Energy Society General Meeting, 2011, pp. 1 – 4

[39] E.Chiodo, “Parameter estimation of mixed Weibull probability distributions for wind speed related to power statistics”, International Symposium on Power Electronics, Electrical Drives, Automation and Motion, 2011, pp.582-587

[40] M.A.Elshahed, M.M. El-marsafawy, H.Z.M. El-Din, “Dynamic economic dispatch constrained by wind power Weibull distribution: A here-and-now strategy”, in World Academy of Science,Engineering and Technology, 2011, vol.56, pp. 353-358

[41] D.Ling, H.Huang and Y. Liu, “A Method for Parameter Estimation of Mixed Weibull Distribution” in Reliability and Maintainability Symposium RAMS, Annual, 2009, pp. 129-133

[42] E.Chiodo, “Parameter estimation of mixed Weibull probability distributions for wind speed related to power statistics”, International Symposium on Power Electronics, Electrical Drives, Automation and Motion, 2011, pp.582-587

[43] M.Abrožič, L.Gorjan, “Reliability of a Weibull analysis using the maximum-likehood method”, in Journal of Materials Science, 2011, vol.46, pp. 1862-1869

[44] P.Bhattacharya, R.Bhattacharjee, “A study on Weibull distribution for estimating the parameters”, in Wind Engineering, 2010, vol.33, pp. 469-476

[45] M.A.Gallagher, Moore and H.Albert, “Robust minumum-distance estimation using the 3-parameter Weibull distribution” in IEEE Transactions on Reliability, vol.39, no.5, 1990, pp.575-580

[46] D.Cousineau, “Fitting the three-parameter Weibull distribution:review and evaluation of existing and new methods” in IEEE Transactions on Dielectrics and Electrical Insulation, vol.16, no.1, 2009, pp.281-288

[47] H. P. Singh, S. Saxena, J. Allen, S. Singh and F. Smarandache, “ Estimation of weibull shape parameter by shrinkage towards an interval under failure censored sampling” in Randomness and optimal estimation in data sampling, 2002, pp. 5-25

[48] S. A. Akdag and A. Dinler "A new method to estimate Weibull parameters for wind energy applications" in Energy Conversion and Management, vol 50, No. 7, 2009, pp. 1761–1766

[49] K. Hisada and I. Arizino " Reliability Tests for Weibull Distribution With Varying Shape-Parameter, Based on Complete Data" in IEEE Transactions on Reliability, vol. 51, no. 3, 2002, pp. 331-336

[50] C. D. Lai, M. Xie and D. N. P. Murthy, "A Modified Weibull Distribution" in IEEE Transactions on reliability, vol. 52, no. 1, 2003, pp. 33-37

[51] M.P. Kaminskiy and V. V. Krivtsov, "A Simple Procedure for Bayesian Estimation of the Weibull Distribution" in IEEE transactions on reliability, vol. 54, no. 4, 2005, pp. 612-616

[52] A. Tuzuner and Z.Yu, "A Theoretical Analysis on Parameter Estimation for the Weibull Wind Speed Distribution", in Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century, 2008, pp. 1-6

[53] T. Desell, N. Cole, M. Magdon-Ismail, H. Newberg, B. Szymanski and C. Varela, "Distributed and Generic Maximum Likelihood Evaluation" in Third IEEE International Conference on e-Science and Grid Computing, 2007, pp. 337-344

[54] C. R. Miller, P. Joyce and L. P. Waits, " Assessing Allelic Dropout and Genotype Reliability Using Maximum Likelihood" in Genetics, vol 160, 2002, pp. 357-366

[55] H. El-Sayed, M. Salit, J. Travis, J.Devaney and W.George, "Parallel Maximum-Likelihood Inversion for estimating Wavenumber-Ordered Spectra in Emission Spectroscopy" in Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International, 2000, pp. 71-76

[56] P.Li, Y.Cao, "Research and implementation of Jacobbi algorithm based on MPI with checkerboard decomposition" in IEEE International Conference on Computer Science and Electronics Engineering, vol 3, 2012, pp. 166-169

[57] T.Hilbrich, M.S.Müller, B.Krammer, "MPI correctness checking for OpenMP/MPI applications", in International Journal of Parallel Programming, 2009, vol.37, pp. 277-291

[58] C.T.Yang, K.C.Lai, "A directive-based MPI code generator for Linux PC clusters", in The Journal of Supercomputing, 2009, vol.50, pp. 177-207

[59] P.Balaji, D.Buntinas, D.Goodell, W.Gropp, S.Kumar, E.Lusk, R.Thakur, J.L.Träff, “MPI on a million processors”, in Parallel Processing Letters, 2011, vol.21, pp. 45-60

[60] R.Brown, I.Sharapov, “High-scalability parallelization of a molecular modeling application: Performance and productivity comparison between OpenMP and MPI implementations”, in International Journal of Parallel Programming, 2007, vol.35, pp. 441-458

[61] W. Pfeiffer and A. Stamatakis ”Hybrid MPI/Pthreads Parallelization of the RAxML Phylogenetics Code” in IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, 2010, pp.1-8

[62] R.Rufai, M.Bozyigit, J.Alghamdi, M.Ahmed, “Multithreaded parallelism with OpenMP”, in Parallel Processing Letters, 2005, vol.15, pp. 367-378

[63] W.Zhong, G.Altun, X.Tian, R.Harrison, P.C.Tai, Y.Pan, “Parallel protein secondary structure prediction schemes using Pthread and OpenMP over hyper-threading technology”, in The Journal of Supercomputing, 2007, vol.41, pp. 1-16

[64] D.a.Mey, S.Sarholz, C.Terboven, “Nested parallelization with OpenMP”, in International Journal of Parallel Programming, 2007, vol.35, pp. 459-476

[65] R.Rabenseifner, G. Hager and G. Jost, “Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes” in 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009,pp. 427 – 436

[66] J.M. Bull, J.Enright, X. Guo , C. Maynard and F. Reid, “ Performance Evaluation of Mixed-Mode OpenMP/MPI” in International Journal of Parallel Programming, vol. 38, no. 5-6, 2010, pp 396-417

[67] R.L.Graham, B.W.Barrett, G.M.Shipman, T.S.Woodall, “Open MPI: A high performance, flexible implementation of MPI point-to-point communications”, in Parallel Processing Letters, 2007, vol.17, pp. 79-88

[68] R.Brown, I.Sherapov, “High-scalability parallelization of a molecular modeling application: performance and productivity comparison between OpenMP and MPI implementations”, in International Journal of Parallel Programming, 2007, vol.35, pp. 441-458

[69] K.Woodsend, J.Gondzio, “Hybrid MPI/OpenMP parallel linear support vector machine training”, in The Journal of Machine Learning Research, 2009, vol.10, pp. 1937-1953

[70] J.M.Bull, J.Enright, X.Gou, C.Maynard, F.Reid, “Performance evaluation of mixed–mode OpenMP/MPI implementations”, in International Journal of Parallel Programming, 2010, vol.38, pp. 396-417

[71] G.Jost, B.Robins, “Experiences using hybrid MPI/OpenMP in the real world: Parallelization of a 3D CFD solver for multi-core node clusters”, in Scientific Programming, 2010, vol.18, pp. 127-138

[72] S.Duli, B.Krstajić “Parallel implementation of the Weibull distribution parameters estimator”, in Journal of Environmental Protection and Ecology (JEPE), ISSN 1311-506, March 2014

[73] S.Duli, B.Krstajić “Parallel processing of wind speed data during years 2012 – 2013 in Shkodra region” in the Scientific Bulletin of Faculty of Natural Sciences, University of Shkodra, volume 64. March 2014

[74] S.Duli, B.Krstajić “Parallel computing of Weibull distribution parameters” in the Scientific Bulletin of Faculty of Natural Sciences, University of Shkodra, volume 62. ISSN : 2221-6847, http://80.78.68.65:1000/buletin/01_Bul_62.pdf, March 2012

[75] S.Duli, B.Krstajić, “Hybrid MPI/Pthread parallelization of the Weibull distribution parameters estimator”, XIX Scientific-Professional Information Technology Conference, Žabljak, February 2014

[76] S.Duli, B.Krstajić “Methods for estimating the parameters of the Weibull distribution” in 2nd International Conference ”Research and education in natural sciences“, ISBN 978-9928-4135-5-0, November 2013

[77] S.Duli, B.Krstajić “MPI in a Weibull distribution parameters estimation” International Conference “Towards future sustainable development“, ISBN 978-9928-4135-0-5, November 2012

[78] S.Duli, B.Krstajić, “Pthread u estimaciji parametara weibull distribucije”, XVII Scientific-Professional Information Technology Conference, Žabljak, February 2012

[79] H.Xuan, W.Tong, Y.Hu, Z.Gong, Y.Hou, “Hybrid programming implementation of BiCG method on SMP Cluster architecture” in IEEE International Conference on Computer and Information Science, 2012, pp. 470-475

[80] A.Shamshad, H.Wan, S. Ahamad, M.Sanusi, “Analysis of Wind Speed Variations and Estimation of Weibull Parameters for Wind Power Generation in Malavsia”, 2018.

[81] T.Rauber,”Parallel programmimg models” in Parallel Programming, Springer, 2010, pp.93-147

Appendix

Pseudocode

```
given the array X(I),I=1,N,
from a sample of size NLocation_parameter_a=0;
perform a NEWTON-RAPHSON ITERATION :
  calculate B = THE ML ESTIMATE OF parameter b.
  calculate DB = B'
  calculate F =D(logL)/D(C), L=likelihood function of
Weibull
  calculate DF = F'
  calculate C as the root of F
output the estimations of B and C parameters
```

The pseudocode of the Newton-Raphson function is:

```
void NEWTON(float *B,float *C,long *N,long *M,float X[],int
*Konvrg)
{
  suml2 = 0;
     suml = 0;
     sum2p = 0;
     sum = 0;
     dif = *N - (*M);
     c1 = 1.0/(*C);
     c2 = c1*c1;
     cm = powf((*M),(-c1));
     for( i = 0;i<tm; i++)
     {
          if (X[i]!=0.0f)
            xia = X[i];
          else
            xia=0.00001f;
          xmac = powf(xia,(*C));
          xlog = logf(xia);
          xlog2 = xlog*xlog;
          xmacl = xmac*xlog;
          suml = suml + xlog;
          sum = sum + xmac;
          suml2 = suml2 + xlog2*xmac;
          sum2p = sum2p + xmacl;
         }
         b1 = sum + dif*xmac;
         b2 = powf(b1,c1);
         *B = b2*cm;
         if ( (*Konvrg)==1 ) return;
```

```
          db = logf((*M))*(*B)*c2 + (-logf(b1)*b2*c2+
(sum2p+xmacl*dif)*(b2/b1)*c1)*cm;
          difx = dif*xmac;
          bc = powf((*B),(-(*C)));
          bc1 = powf((*B),(-(*C)-1));
          blog = logf((*B));
          bcd = bc1*(*C)*db;
          dbb = db*bc1;
          blogc = blog*bc;
          bcdb = -bcd - blogc;
          xlogmb = (xlog-blog)*difx;
          blogm = blog*(*M);
          f1 = suml - blogm;
          f2 = -(sum2p*bc-sum*blogc);
          f3 = -xlogmb*bc + (*M)*c1;
          f = f1 + f2 + f3;
          df1 = -db*(*M)/(*B);
          df2 = -(sum2p*bcdb+sum*(blog*bcd+blog*blogc));
          df3 = sum*dbb;
          df4 = -bc*suml2 + sum2p*blogc;
          df5 = -xlogmb*bcdb;
          df6 = dbb*difx;
          df7 = -xlog*xlogmb*bc - (*M)*c2;
          df = df1 + df2 + df3 + df4 + df5 + df6 + df7
          delta = f/df;
          *C = *C - delta;
          if ( abs(delta/(*C))<=1 )
             *Konvrg = 1 ;
  }
```

Biography

Sidita Duli was born on 22 August 1983, in Tirana, Albania. She attended the primary school and high school in her hometown, Shkodra, Albania.

She graduated on 18 July 2006 in Computer Sciences, Faculty of Natural Sciences, in the University of Shkoder. Her average of grades in the bachelor degree was 8.92, and got the highest grade in the final exam.

As soon as she finished the studies, she is enrolled as assistant lecturer in the University of Shkodra, Faculty of Natural Sciences. The subjects she is teaching are : Java programming language, C++ programming language, Operating systems.

During summer of the years of 2005 and 2006, she attended summer school in Klagenfurt, Austria. The main subject learned was German Language.

During the year 2007, she was member of the project for building and maintaining the official web site for the University of Shkodra.

During years 2006-2008, she attended the master studies in the Faculty of Computer Sciences, in the University of Tirana. The program was a two years program. She defended the Master thesis “Web Services which enables data exchange between different applications” on June 2008.

In October 2010 she attended the 9-nth SimLab course in Belgrade, Serbia.

In July 2009, she enrolled in the PhD studies in Department of Electrical Engineering, University of Montenegro.

In March 2015 she is nominated Vice Dean of the Faculty of Natural Sciences, University of Shkodra. She maintained this position until April 2016.

She published one paper in International Journal (with SCI list, with impact factor 0.73) and over 15 works in international and regional journals and conferences.

IZJAVA O AUTORSTVU

Potpisani: Sidita Duli

Broj indeksa: 1/2009

Izjavljujem

da je doktorska disertacija pod naslovom

„An approach for the parallelization of the Weibull distribution parameter estimators with applications“ („Jedan pristup paralelizaciji estimacije parametara Weibull distribucije sa primjenom“

- rezultat sprovedenog naučno istraživačkog rada,
- da predložena disertacija ni u cjelini ni u djelovima nije bila perdložena za dobijanje bilo kakve diplome prema studijskim programima drugih ustanova visokog obrazovanja,
- da su rezultati korektno navedeni, i
- da nijesam povrijedio autorska i druga prava intelektualne svojine koja pripadaju trećim licima.

Potpis doktoranda

U Podgorici, jun 2018.

IZJAVA O ISTOVJETNOSTI ŠTAMPE I ELEKTRONSKE VERZIJE DOKTORSKOG RADA

Ime i prezime autora: Sidita Duli

Broj indeksa/upisa: 1/2009

Studijski program: Računari

Naslov rada: **„An approach for the parallelization of the Weibull distribution parameter estimators with applications“ („Jedan pristup paralelizaciji estimacije parametara Weibull distribucije sa primjenom“**

Mentor: Prof. dr Božo Krstajić

Potpisani: Uglješa Urošević

Izjavljujem da je štampana verzija mog doktorskog rada istovjetna elektronskoj verziji koju sam predao za objavljivanje u Digitalni arhiv Univerziteta Crne Gore.

Istovremeno izjavljujem da dozvoljavam objavljivanje mojih ličnih podataka u vezi sa dobijanjem akademskog naziva doktora nauka, odnosno zvanja doktora umjetnosti, kao što su ime i prezime, godina i mjesto rođenja, naziv disertacije i datum odbrane rada.

Potpis doktoranda

U Podgorici, jun 2018.

IZJAVA O KORIŠĆENJU

Ovlašćujem Univerzitetsku biblioteku da u Digitalni arhiv Univerziteta Crne Gore pohrani moju doktorsku disertaciju pod naslovom:

„An approach for the parallelization of the Weibull distribution parameter estimators with applications“ („Jedan pristup paralelizaciji estimacije parametara Weibull distribucije sa primjenom“

koja je moje autorsko djelo.

Disertaciju sa svim prilozima predao sam u elektronskom formatu pogodnom za trajno arhiviranje.

Moju doktorsku disertaciju pohranjenu u Digitalni arhiv Univerziteta Crne Gore mogu da koriste svi koji poštuju odredbe sadržane u odabranom tipu licence Kreativne zajednice (Creative Commons) za koju sam se odlučio.

1. Autorstvo
2. Autorstvo – nekomercijalno
3. Autorstvo – nekomercijalno – bez prerade
4. Autorstvo – nekomercijalno – dijeliti pod istim uslovima
5. Autorstvo – bez prerade
6. Autorstvo – dijeliti pod istim uslovima

Potpis doktoranda

U Podgorici, jun 2018.

1. Autorstvo – Dozvoljavate umnožavanje, distribuciju i javno saopštavanje djela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence, čak i u komercijalne svrhe. Ovo je najslobodnija od svih licenci.

2. Autorstvo – nekomercijalno. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje djela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence. Ova licenca ne dozvoljava komercijalnu upotrebu djela.

3. Autorstvo – nekomercijalno – bez prerade. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje djela, bez promjena, preoblikovanja ili upotrebe djela u svom djelu, ako se navede ime autora na način određen od strane autora ili davaoca licence. Ova licenca ne dozvoljava komercijalnu upotrebu djela. U odnosu na sve ostale licence, ovom licencom se ograničava najveći obim prava korišćenja djela.

4. Autorstvo – nekomercijalno – dijeliti pod istim uslovima. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje djela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence i ako se prerada distribuira pod istom ili sličnom licencom. Ova licenca ne dozvoljava komercijalnu upotrebu djela.

5. Autorstvo – bez prerade. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje djela, bez promjena, preoblikovanja ili upotrebe djela u svom djelu, ako se navede ime autora na način određen od strane autora ili davaoca licence. Ova licenca dozvoljava komercijalnu upotrebu djela.

6. Autorstvo – dijeliti pod istim uslovima. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje djela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence i ako se prerada distribuira pod istom ili sličnom licencom. Ova licenca dozvoljava komercijalnu upotrebu djela i prerada. Slična je softverskim licencama, odnosno licencama otvorenog koda.