



Universitat Autònoma de Barcelona

Escola d'Enginyeria

**Departament d'Arquitectura de Computadors
i Sistemes Operatius**

R/parallel

Parallel Computing for R in non-dedicated environments

Thesis submitted by Gonzalo Vera Rodríguez in partial fulfillment of the requirements for the degree of PhD per the Universitat Autònoma de Barcelona. This work was advised by Dr. Remo Suppi Boldrito.

Barcelona, May 2010

R/parallel

Parallel Computing for R in non-dedicated environments

Thesis submitted by Gonzalo Vera Rodríguez in partial fulfillment of the requirements for the degree of PhD per the Universitat Autònoma de Barcelona. This work was developed in the Computer Architecture and Operating Systems department of the Universitat Autònoma de Barcelona in option A – “Computer Architecture and Parallel Processing” of the PhD Informatics program, being advised by Remo Suppi Boldrito.

Barcelona, May 2010

Thesis advisor

Dr. Remo Suppi Boldrito

To the light, that shows me the way

Abstract

Traditionally, parallel computing has been associated with special purpose applications designed to run in complex computing clusters, specifically set up with a software stack of dedicated libraries together with advanced administration tools to manage complex IT infrastructures. These High Performance Computing (HPC) solutions, although being the most efficient solutions in terms of performance and scalability, impose technical and practical barriers for most common scientists whom, with reduced IT knowledge, time and resources, are unable to embrace classical HPC solutions without considerable efforts.

Moreover, two important technology advances are increasing the need for parallel computing. For example in the bioinformatics field, and similarly in other experimental science disciplines, new high throughput screening devices are generating huge amounts of data within very short time which requires their analysis in equally short time periods to avoid delaying experimental analysis. Another important technological change involves the design of new processor chips. To increase raw performance the current strategy is to increase the number of processing units per chip, so to make use of the new processing capacities parallel applications are required. In both cases we find users that may need to update their current sequential applications and computing resources to achieve the increased processing capacities required for their particular needs. Since parallel computing is becoming a natural option for obtaining increased performance and it is required by new computer systems, solutions adapted for the mainstream should be developed for a seamless adoption.

In order to enable the adoption of parallel computing, new methods and technologies are required to remove or mitigate the current barriers and obstacles that prevent many users from evolving their sequential running environments. A particular scenario that specially suffers from these problems and that is considered as a practical case in this work consists of bioinformaticians analyzing molecular data with methods written with the R language. In many cases, with long datasets, they have to wait for days and weeks for their data to be processed or perform the cumbersome task of manually splitting their data, look for available computers to run these subsets and collect back the previously scattered results. Most of these applications written in R are based on parallel loops. A loop is called a parallel loop if there is no data dependency among all its iterations, and therefore any iteration can be processed in any order or even simultaneously, so they are susceptible of being parallelized. Parallel loops are found in a large number of scientific applications.

Previous contributions deal with partial aspects of the problems suffered by this kind of users, such as providing access to additional computing resources or enabling the codification of parallel problems, but none takes proper care of providing complete solutions without considering advanced users with access to traditional HPC platforms. Our contribution consists in the design and evaluation of methods to enable the easy parallelization of applications based in parallel loops written in R using non-dedicated environments as a computing platform and considering users without proper experience in parallel computing or system management skills. As a proof of concept, and in order to evaluate the feasibility of our proposal, an extension of R, called R/parallel, has been developed to test our ideas in real environments with real bioinformatics problems.

The results show that even in situations with a reduced level of information about the running environment and with a high degree of uncertainty about the quantity and quality of the available resources it is possible to provide a software layer to enable users without previous knowledge and skills adapt their applications with a minimal effort and perform concurrent computations using the available computers. Additionally of proving the feasibility of our proposal, a new self-scheduling scheme, suitable for parallel loops in dynamics environments has been contributed, the results of which show that it is possible to obtain improved performance levels compared to previous contributions in best-effort environments.

The main conclusion is that, even in situations with limited information about the environment and the involved technologies, it is possible to provide the mechanisms that will allow users without proper knowledge and time restrictions to conveniently make use and take advantage of parallel computing technologies, so closing the gap between classical HPC solutions and the mainstream of users of common applications, in our case, based in parallel loops with R.

Acknowledgments

It has been several years since this long travel begins and a lot of people that have walk at my side during this time have contributed in one or another way to the conclusion of this journey, so there is a lot of friends to whom I have to be grateful.

From the early days I want to thank my work friends at La Vanguardia, Viasalus and T-Systems. I still remember how the last ones helped me to stay on track every time they were fondly teasing me with “Herr Doctor”. I also want to thank all my friends of Santa Coloma de Gramenet that, no matter the years, they kept asking for my thesis and were patiently listening for my explanations, even though sometimes, I am pretty sure, they could not understand me.

I keep great memories of my stay in The Netherlands at the Groningen Bioinformatics Centre. I want to thank Ritsert C. Jansen who opened for me the doors of his department and introduced me to world of bioinformatics and genetical genomics. I also want to thank to all my colleagues there: Rainer Breitling, for sharing, not only his room, but also his valuable experience and opinions, Morris A. Swertz, for our endless discussions about bioinformatics data modeling and bindings with R, Rudi Alberts and his coffee machine, with whom I get my first bioinformatics publication, Martijn Dijkstra, from whom I learned how to bring problems to the edge, Richard Scheltema and Bruno Tesson, with their great discussions after lunch, and Yang Li and Jingyuan Fu, an example of perseverance, for providing me with real R user needs. Een groot dank-je-wel aan alle.

I want to thank Armand Sánchez for welcome me and allow me to keep working in bioinformatics, providing new challenging problems where I was able to apply my research. These thanks are extended to all the colleagues of the Research Centre in Agrigenomics (CSIC-IRTA-UAB Consortium) with whom I have been working in the last years. My experience with them has been of great value for this work.

The members of the department of Computer Architecture and Operating Systems, where this thesis has been done, also deserves my gratitude. I want to thank all the colleagues with whom I have shared worries and experiences developing our corresponding researches and writing our papers. It would be inexcusable not to thank the technical team (famous PT), Dani and Javi, without which many experiments could not be performed. I owe you more than a couple of coffees. I also want to thank Porfidio Hernández and all the professors of the department that join me at lunch time and help me to take a break while providing good recommendations when I needed.

I want to thank my thesis director, Remo Suppi, for allowing me to begin this thesis and giving me the freedom to choose the research lines that we considered more interesting. I am also grateful for his maintained confidence, even when past professional responsibilities emerged and made things a bit more difficult.

Los agradecimientos más especiales son para mi familia. Mis padres, Asunción y Gonzalo, y mis hermanos Carolina y Abraham. Gracias por apoyarme incondicionalmente en cualquiera de mis decisiones, y darme todos los ánimos y soporte necesarios para alcanzar mis objetivos y poder terminar este trabajo. Gràcies també a la meva estimada Lorena, per caminar conjuntament amb mi tots aquests anys, en els bons moments i en els temps difícils, però sempre mirant cap endavant i avançant junts. You are my light.

A tots vosaltres, gràcies per ajudar-me a arribar al final d'aquest camí.

Contents

Abstract.....	IV
Acknowledgments	VI
Contents	VIII
List of Figures.....	XII
List of Tables	XIV
Chapter 1 Introduction.....	1
1.1. Introduction.....	1
1.2. Motivation.....	1
1.3. Goals	7
1.4. Thesis Outline	9
Chapter 2 Characterizing the Running Environment	11
2.1. Introduction.....	11
2.2. Parallel Architecture Models	12
2.3. Parallel Programming Paradigms and Models.....	16
2.3.1. Parallel Programming Paradigms	17
2.3.2. Parallel Programming Models	20
2.4. Scheduling Strategies.....	21
2.5. Classifying Parallel Loops	23
2.6. Desktop Grids as an Additional Source for Computing Resources.....	27
2.6.1. Taxonomy of Desktop Grid	28
2.7. Parallel Computing with R.....	31
2.7.1. Packages Supporting the Message Passing Model	31
2.7.2. Packages Supporting the Shared Memory Model	33
2.7.3. Packages Supporting Assisted Parallelization	34
2.7.4. Other Packages Enabling Additional Resources	35
2.8. Concluding Remarks.....	36
Chapter 3 R/parallel – A Proposal for Parallel Loops with R.....	39
3.1. Introduction.....	39
3.2. Extending the R Interpreter with R/parallel.....	41

3.2.1.	Expressing Parallelism.....	41
3.2.2.	Transforming the Original Program	46
3.2.3.	Accessing Additional Processing Units.....	47
3.3.	Providing Support for Distributed Computing.....	52
3.3.1.	Adding Support for Additional Remote Workers.....	53
3.3.2.	Allocating Additional Working Nodes.....	53
3.3.3.	Adapting the Scheduling Schemes	56
3.4.	Concluding Remarks.....	57
Chapter 4	Parallel Loop Scheduling.....	59
4.1.	Introduction.....	59
4.2.	Nonadaptive Scheduling Schemes	60
4.3.	Adaptive Scheduling Schemes.....	63
4.4.	Extended Contributions for Parallel Loop Scheduling	64
4.4.1.	Extensions of Previous Schemes	65
4.4.2.	Extensions Applied to Distributed Environments	67
4.4.3.	Other Current Loop Scheduling Schemes	69
4.5.	Improving Parallel Loop Scheduling	71
4.5.1.	Preliminary Considerations	71
4.5.2.	Proposal of a New Scheduling Scheme	72
4.6.	Concluding Remarks.....	75
Chapter 5	Evaluation of Proposals	77
5.1.	Evaluation of R/parallel with Multiprocessor Computers	77
5.1.1.	Evaluating Applicability to Real Cases	78
5.1.2.	Evaluating Scalability and Efficiency	80
5.2.	Evaluation of R/parallel with Distributed Systems.....	82
5.2.1.	Evaluating Distributed Computing in Homogeneous Environments ..	83
5.2.2.	Evaluating Parallel Loop Scheduling Schemes	85
5.2.3.	Evaluating Distributed Computing in Heterogeneous Environments ..	89
5.3.	Evaluation of the New Scheduling Scheme ATLS.....	96
5.3.1.	Evaluating ATLS in a Static and Homogeneous Environment	97
5.3.2.	Evaluating ATLS in a Static and Heterogeneous Environment	98

5.3.3. Evaluating ATLS in a Dynamic Environment.....	100
5.4. Concluding Remarks.....	102
Chapter 6 Conclusions and Future Work	105
6.1. Conclusions.....	105
6.2. Future Work.....	107
Bibliography	111

List of Figures

FIGURE 1: A RUNNING ENVIRONMENT IS DEFINED BY ITS THREE LAYERS: USERS, APPLICATIONS AND PROCESSING UNITS	3
FIGURE 2: SOURCES OF VARIABILITY AND RANGES OF CONCEPTS	5
FIGURE 3: DISTANCE TO HPC SOLUTIONS AMONG EXISTING RUNNING ENVIRONMENTS	5
FIGURE 4: CHARACTERIZING OUR RUNNING ENVIRONMENT	12
FIGURE 5: DISTRIBUTED-MEMORY ARCHITECTURE MODEL	13
FIGURE 6: SHARED-MEMORY ARCHITECTURE MODEL	14
FIGURE 7: INDIVIDUAL END SYSTEMS	14
FIGURE 8: CLUSTER OF COMPUTERS	15
FIGURE 9: INTRANET OF HETEROGENEOUS COMPUTERS BELONGING TO THE SAME ORGANIZATION	15
FIGURE 10: INTERNET CONNECTS UNRELATED COMPUTERS WIDELY DISTRIBUTED GEOGRAPHICALLY IN VERY LARGE NUMBERS.....	16
FIGURE 11: BASIC STRUCTURE OF AN SPMD PROGRAM	17
FIGURE 12: A MASTER-WORKER PARADIGM	18
FIGURE 13: DATA PIPELINE STRUCTURE.....	18
FIGURE 14: DIVIDE AND CONQUER AS A VIRTUAL TREE	19
FIGURE 15: SPECULATIVE PARALLELISM	19
FIGURE 16: SELF-SCHEDULING USING THE MASTER-WORKER PARADIGM	23
FIGURE 17: DOALL LOOP WITH INDEX VARIABLE I.....	24
FIGURE 18: pR ARCHITECTURE (SOURCE [ML+07]).....	34
FIGURE 19: INDICATING PARALLEL REGIONS	43
FIGURE 20: EXAMPLE R FUNCTION ENCODING A PARALLEL LOOP.....	44
FIGURE 21: PARALLEL LOOP INDICATED WITH AN IF-ELSE CONSTRUCT.....	44
FIGURE 22: DIVERTING THE FLOW OF EXECUTION	46
FIGURE 23: GENERAL DESIGN STRATEGY FOR PARALLELIZING A NON-THREAD-SAFE LEGACY APPLICATION	49
FIGURE 24: GENERAL MASTER-WORKER MODEL IMPLEMENTED IN R/PARALLEL	50
FIGURE 25: SEVERAL DISTRIBUTED COMPUTING RESOURCES CAN BE USED TO INCREASED THE PROCESSING CAPACITIES OF A PARALLEL SYSTEM.....	52
FIGURE 26: CODE SNIPPET TO ADD VOLUNTEER NODES	54
FIGURE 27: REQUEST FOR COLLABORATION (RFC) PROCEDURE TO RETRIEVE ADDITIONAL WORKING NODES	54
FIGURE 28: INTERNAL COMPONENTS OF R/PARALLEL INVOLVED IN A DISTRIBUTED COMPUTATION	55
FIGURE 29: CHANGES IN THE RELATIVE CAPACITY OF HETEROGENEOUS COMPUTERS CAN BE CAUSED BY SEVERAL FACTORS, LIKE VARIABLE LOADS AND NETWORK CONGESTION	56
FIGURE 30: A) SCHEDULE OBTAINED FROM EXISTING TECHNIQUES, B) OPTIMAL SCHEDULE (SOURCE [KN+05]).....	66
FIGURE 31: EXAMPLE OF CODE USED TO TEST R/PARALLEL (SOURCE [VS08])	78
FIGURE 32: INCREASING THE AMOUNT OF WORK PERFORMED BEFORE A TIME LIMIT	79
FIGURE 33: DECREASING THE REQUIRED TIME TO PROCESS A FIXED WORKLOAD.....	80
FIGURE 34: DECREASE OF THE TOTAL EXECUTION TIME PARALLELIZING QTLMAP.XPROBESET()	80
FIGURE 35: SPEEDUP DECAYS WITH INTRODUCTION OF PROCESSING UNITS	81
FIGURE 36: EFFICIENCY FIGURES OBTAINED WITH 8 CORES	82
FIGURE 37: NETWORK DIAGRAM AND CONFIGURATIONS OF HOMOGENEOUS TEST BED WITH 40 PROCESSING UNITS	83
FIGURE 38: SCRIPT USED TO PARALLELIZED LOOPS WITH R/QTL PACKAGE FUNCTIONS	84

FIGURE 39: PERFORMANCE RESULTS USING 40 WORKERS	84
FIGURE 40: CODE SNIPPET OF A PARALLELIZED CLIPPING PROCEDURE OF DNA SEQUENCE READS.....	86
FIGURE 41: NETWORK DIAGRAM AND CONFIGURATIONS OF HETEROGENEOUS TEST BED WITH 140 PROCESSING UNITS	89
FIGURE 42: TOTAL EXECUTION TIMES WITH 120 WORKERS USING HOMOGENEOUS AND HETEROGENEOUS ENVIRONMENTS	91
FIGURE 43: ORDERED SEQUENCE AND SIZE (I.E. NUMBER OF ITERATIONS) OF TASK ARRIVALS. SECOND ROW ILLUSTRATES A ZOOM OF THE FIRST 500 TASKS COMPLETED.....	92
FIGURE 44: NETWORK DIAGRAM AND CONFIGURATIONS OF HETEROGENEOUS TEST BED WITH 150 PROCESSING UNITS	94
FIGURE 45: EXPERIMENTAL RESULTS PROCESSING 434000 SEQUENCES WITH 80 AND 20 WORKERS USING A MIXTURE OF FAST AND SLOW NODES	94
FIGURE 46: EXPERIMENTAL RESULTS PROCESSING 10000 SEQUENCES VARYING THE PROPORTION OF SLOW-FAST NODES FROM 0-10 TO 10-0 EXCHANGING 2 NODES OF EACH TYPE AT A TIME	95
FIGURE 47: EXPERIMENTAL RESULTS PROCESSING 10000 SEQUENCES WITH 20, 30, 40, 60 AND 80 HOMOGENEOUS WORKERS WITH AWF AND A FIXED NUMBER OF P=40 WORKERS CONFIGURED FOR STATIC.....	96
FIGURE 48: TOTAL EXECUTION TIME AND SPEEDUP USING 4, 16 AND 32 WORKERS IN AN HOMOGENEOUS ENVIRONMENT	97
FIGURE 49: TOTAL EXECUTION TIME USING AN HETEROGENEOUS ENVIRONMENT WITH 32 WORKERS	98
FIGURE 50: NUMBER OF ITERATIONS PER WORKER DISPATCHED BY EACH SCHEDULER.....	99
FIGURE 51: LIMITATION TYPES PROGRAMMED: STEP-DOWN, STEP-UP, PULSE CONSTANT AND PULSE VARIABLE.....	100
FIGURE 52: NUMBER OF ITERATIONS PER TASK AT DISPATCHING TIME IN A DYNAMIC ENVIRONMENT	101
FIGURE 53: NUMBER OF ITERATIONS PER TASK AT FINISHING TIME IN A DYNAMIC ENVIRONMENT	102
FIGURE 54: QUEUE SUBSYSTEM TO SUPPORT FAULT TOLERANCE.....	108

List of Tables

TABLE 1: CODE GRANULARITY AND PARALLELISM.....	16
TABLE 2: COMPARISON OF R/PARALLEL WITHIN THE CONTEXT OF SOME POPULAR DESKTOP GRID SYSTEMS.....	30
TABLE 3: TOTAL EXECUTION TIME AND THROUGHPUT RESULTS OF SEVERAL PARALLEL LOOP SCHEDULING SCHEMES USING HOMOGENEOUS ENVIRONMENTS	87
TABLE 4: SCALABILITY AND EFFICIENCY RESULTS OF SEVERAL PARALLEL LOOP SCHEDULING SCHEMES USING HOMOGENEOUS ENVIRONMENTS.....	88
TABLE 5: AVERAGE μ AND PERCENTAGE OF STANDARD DEVIATION $\% \Sigma$ OF THE TOTAL EXECUTION TIMES OBTAINED DURING THE EXPERIMENTS	100

Chapter 1

Introduction

1.1. Introduction

In this chapter we describe first, from a general perspective, the problem that has motivated this work. Next we consider a particular scenario where this problem is present, and after identifying the main obstacles that currently avoid reaching a solution, the goals for this thesis are established. Finally, an outline of the contents of this dissertation is provided.

1.2. Motivation

Emerging trends in technology and new developments in science are increasing the need for parallel computing, and as a consequence a growing number of users demand new solutions that fit their specific requirements. Current developments in chip design and new measurement and sampling devices are two of the main examples of technological evolutions that are promoting an increased interest in High Performance Computing (HPC) solutions adapted for the mainstream.

During the last years we are witnessing a new age characterized by massive adoption of multiprocessor computers, even at desktop level with the popularization of multicore computers. The change on the generalized way microprocessor manufacturers are increasing the raw performance of their products started a debate

still open about the serious implications of these new micro-architectures over applications performance [SL05].

The fact is that uniprocessor performance, as a consequence of Moore's law [Moo65], does not increase their performance every 18 months any more, but, as the last trends show, every 5 years. One of the main reasons is that there are diminishing returns on finding more instruction-level parallelism (ILP) [HP07]. It is becoming more difficult to reveal more ILP via compilers and architecture innovation. Examples from the past include branch prediction, out-of-order execution, speculation, and Very Long Instruction Word systems [HP07]. The consequence, as is emphasized in [AB+06], is that increasing clock frequency it is not currently the primary method for improving processor performance and nowadays the main method selected by chip manufacturers to keep delivering more powerful microprocessors to the market is increasing parallelism at a higher level. The drawback is that in order to take advantage of multiple processing units aggregated in a single chip or multiprocessor (e.g. multicore processors), parallel applications, able to run concurrently in several processing units, are required.

A second technological evolution (revolution for some) is introducing major changes in research that are leading to the same conclusion. During the last years, with the development of new technologies in many fields like molecular biology, experimentation costs have drop to a fraction of previous ones, and finalization times have fall from months to few weeks. This situation has generated a large increase in the volume of data obtained from each experiment, as well as an increase in the rate at which data is produced. As a consequence, not only large research groups have now more data to analyze but also smaller groups, that in the past where unable to undertake such experiments due to budget limitations, now can afford the required investments and proceed likewise.

The problem appears when these new volumes of data have to be processed. In the past, researchers were able to patiently annotate their observations just by hand, and statistical analysis was possible using common spreadsheets and other tools running on top of single standard desktop computers. Nowadays, many traditional tools used by researchers are not useful anymore in terms of throughput delivery. Depending on the size of their datasets and the complexity of the analysis required for each piece of data they have to wait for long days and weeks in order to retrieve their results. Impatient users with enough technical skills in some cases are able to manually partition their data and launch their analytical methods in several additional computers that have to reach at least twice, first to set up and run their analysis and second, to collect back their results and rebuild the final figures.

Advanced users, with previous experience in parallel solutions or the time and skills to acquire such knowledge, can transform their applications and adapt them, if

feasible, with existing HPC technologies. In all cases the amount of time required is counterproductive and should be reduced, therefore new tools or adaptations of previous ones are required to harness the new demands of computing power. Moreover, if we take into consideration initiatives like the objective of obtaining a human genome for just \$1000 [Mar06] (nowadays it requires approximately 1 million dollars), it is obvious that the problem will acquire new dimensions for the coming years. Nevertheless, in order to process increased data volumes, increased processing capacities, which can be provided by parallel applications running on top of parallel systems, are required.

The obvious consequence for those users with direct relation with any of these two technological changes is that they are forced to adopt parallel computing solutions to deal with the problems imposed by the technological scenarios just described, either to reduce their processing times or to increase the size of the data being processed within the same time frame.

However, in order to adopt HPC solutions, *appropriate* parallel systems, properly linked with *suitable* parallel applications must be used. Emphasis is done in “appropriate” and “suitable” to remark that any combination of these, computing resources and programs, do not necessarily provide a working HPC solution. To define a working parallel running environment multiple types of computing resources exist as well as types of applications that can be combined. But a running environment, as it is naively depicted in Figure 1, also has to consider, on top of them, their users, the diversity of which also defines the right combination of elements which provides a working and useful parallel computing solution.

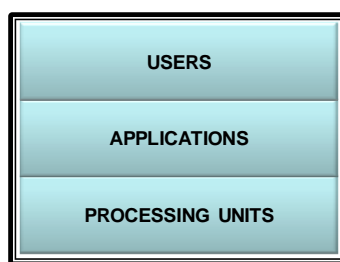


Figure 1: A running environment is defined by its three layers: Users, Applications and Processing Units

Although by users we mean indistinctly end-users and programmers, what we observe is that all users do not have the same ability (and need) to modify the running environment to which they belong, thus determining the viable options for building the parallel solutions required by their needs. Some of them are users that just know how to install their applications and use them through their particular user interfaces while others are able to program any specific functionality and deal with detailed system management configurations.

How a running environment can be adapted to provide a working parallel computing environment will depend on the specific characteristics of the elements present at each layer and the existing possibilities to adapt or replace these elements. We enumerate some differences found at each layer:

- **Processing Units (PROC).** Different types of computing resources and different access and utilization methods exist. We enumerate the most common:
 - Different number of processing units aggregated in different architectures: from SMP computers to large clusters or massive clouds.
 - Set up for general purpose utilization or for specific applications.
 - Exclusive or shared access.
 - Managed by single operating systems or large distributed platforms.
 - Optimized for HPC or non-dedicated computing platforms.

- **Applications (APP).** Different types of applications, software components and algorithms define different requirements and needs:
 - Different workloads, from few to very large data sets.
 - Different running times, from seconds to days and months.
 - Different implementation languages and running environments.
 - Different expertise and knowledge of target users and programmers.
 - Different degrees of parallelism: Independent tasks or dependent tasks.
 - Different parallel programming models: explicit or implicit.

- **Users (USER).** Different types of users define additional restrictions to possible solutions:
 - Users of static and specific applications or dynamic and generic fast prototyping tools.
 - Different knowledge levels of parallel programming.
 - Different management skills of computing resources.
 - Different available time (and will) to acquire new knowledge and skills.
 - Different available time to adapt or develop new programs.
 - Different task force: from large groups to single users and programmers.

This multiplicity of scenarios, types, design purposes, architectures, access methods and interests present among the elements of each layer show that many of them vary within a range the limits of which in many cases represent concepts completely opposite. Figure 2 helps to illustrate these ranges and their opposite ends.

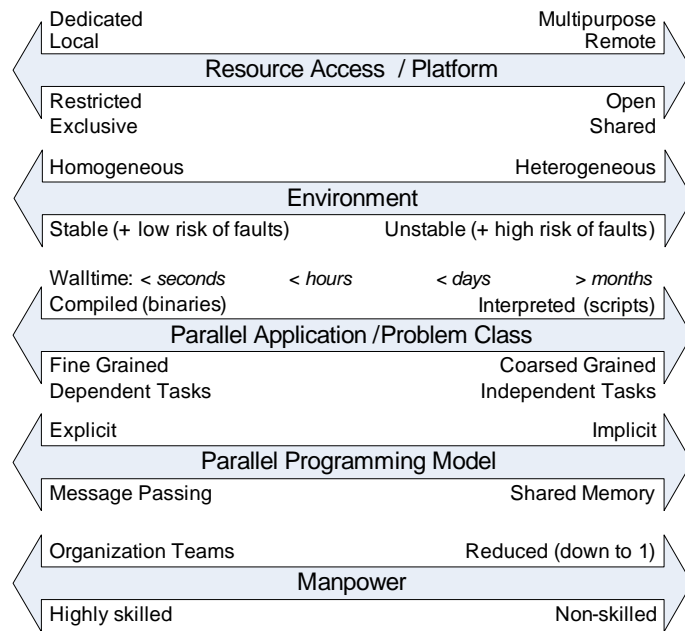


Figure 2: Sources of variability and ranges of concepts

We can observe that there are scenarios where it is more complicated than others to apply HPC techniques. The question at this point, since there are users that need and require the utilization of parallel computing, is if it is possible, considering any running environment integrated by a particular combination of processing units, applications and users, to adapt any of these combinations to provide a working HPC solution. It is clear that in some cases we are already witnessing a HPC solution while in others there is a significant gap to be filled. For example we can find users of sequential applications whom, without knowledge of parallel programming are unable to transform their programs or users who already have a parallel application but they lack the basic skills to use additional computing resources existing at their reach. In this work we are interested in the cases represented by these last examples. Figure 3 illustrates both ends of possible running environments related with their distance to HPC solutions.

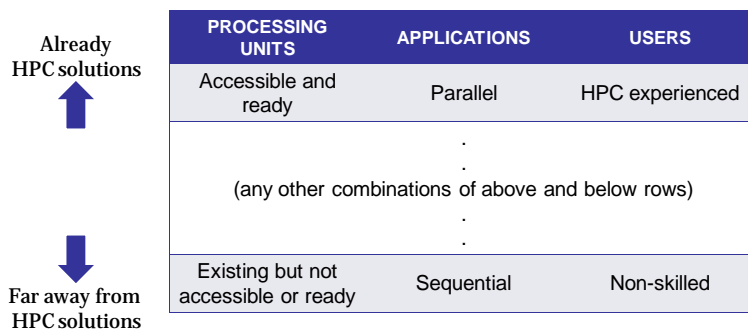


Figure 3: Distance to HPC solutions among existing running environments

In general, the running environment or scenarios of interest for our work can be summarized by those where any of the following factors appear combined:

- There are no computing resources dedicated to HPC.
- There are no parallel versions of applications.
- There is no proper experience or skills in HPC among their users.
- And there are no time and no money to learn, implement or invest.

A definitive answer to cover any scenario cannot be provided. However considering the particular details of each scenario, independently of other combinations, we think that it is possible to provide positive answers.

For our work we have selected a particular scenario where it is quite common to find the combination of factors just described, that is, within the field of molecular biology, the analysis of experimental data by bioinformaticians. In that case, for example, new sequencing devices [Met10] are able to produce millions of reads of DNA sequences within few days and at a fraction of previous costs. These sequences, like other sources of data like genotype profiles or molecular signals, must be processed to find out, besides of the structure and composition of molecules, the genomic information coded in different samples, so for their type of problems, given the large amounts of data to be processed, HPC solutions are expected and very welcome.

The relatively unforeseen sudden appearance of these computing needs has catch many groups without proper knowledge of parallel computing development or without suitable computing resources for performing their analysis. In bioinformatician teams it is not usual to find people with expertise in parallel computing. As a consequence, although they are able to produce high quality programs to analyze their data, they lack experience in parallel programming or system management skills to set up parallel systems. Besides, since the new dimension of their problems is relatively new for them they also lack of dedicated HPC computing environments. For them, with severe time and budget constrains to conduct their experiments and virtually no time to acquire new development or system administration skills, it is of high importance the availability of accessible tools that fulfils their needs while requiring a minimal effort in time and investments.

Finally, the class of programs used at several steps of their processing pipelines, and like in a large number of scientific applications, is based on *parallel loops*. A loop is called a parallel loop if there is no data dependency among all its iterations, i.e. any iteration can be processed in any order or even simultaneously. Parallel loops are a

class of embarrassingly parallel problems which provides excellent performance improvements over sequential loops and therefore methods adapted to their particular needs and running environments are of high interest.

In our case of study it is quite frequent to find programs which iteratively apply the same function to each sample of data, and which, with new increased volumes, spend longer processing times. Their options for reducing these waiting times are several. Of course they could try to run their programs, if available, in faster processors what requires additional investments, or they could recode and optimize their applications, with the required knowledge, and use more efficient platforms. For example, the difference in terms of processing speed between running a given algorithm using interpreted languages or using binaries obtained from compiled languages is in some cases several orders of magnitude faster in favor of the compiled programs. This option, if feasible for the users, is advisable for the cases where the same program is going to be used repeatedly so the time investment is recovered after several runs. Another option, not exclusive with previous ones, is the utilization of parallel computing. As already discussed the problems regarding with knowledge, time and resources apply here and solutions to ease its utilization should be provided.

As a consequence, the general objectives of this thesis are, first, identify and analyze the obstacles that prevent users from using parallel computing technologies within the context of our target running environment, second, propose methods and mechanisms to avoid or mitigate the identified problems, and finally, validate our proposals by developing a working prototype to assess the correctness of our solutions by testing them in real scenarios. Next section describes the details of this objective and explains how it is planned to be achieved.

1.3. Goals

To achieve our objective, the first issue to deal with is to remove any additional complexity introduced by current systems. By reducing the complexity of programming parallel applications and the burden of deploying and managing the underlying resources, easy-to-use tools with better degrees of accessibility can be produced. To achieve this, additional objectives have to be accomplished. For example, external dependencies from the underlying infrastructures have to be removed in order to provide a tool independent of the available resources. The same concept applies for dependencies from third party components that can hinder the future maintenance life cycle of applications.

Next, the same solution should provide methods to support the automatic parallelization of existing programs, trying to avoid the recodification of already working programs. And finally, it should also provide methods to aggregate the

already available but not used computing resources. By developing such system to adapt a running environment, while all the complexity is managed under the hood, any non skilled researcher should be able to take advantage of current parallel computing technologies.

In general, the goals of this thesis can be enumerated as follows:

- Define new methods *to transform* existing sequential applications based in parallel loops, with a minimum knowledge of parallel programming, into parallel applications.
- Define new methods to allow users *to run* their classical applications in parallel environments, with a minimum effort and a minimum knowledge of the computing resources involved.
- Define new methods *to integrate* multiple processing units of different types, mainly using non-dedicated computing resources since they exist but are not used.
- Define new methods *to improve the performance* obtained when integrating non-dedicated resources of heterogeneous types in changing environments.

In particular, this thesis will be focus on developing new methods to enable and improve the utilization of non-dedicated computing resources to process parallel loops found in programs written with R [IG96]. R is a scripting language which is becoming a *de facto* standard between bioinformaticians developing, using and exchanging statistical methods, and where several tools to support the development and execution of parallel applications have been contributed [SM+09]. However, although most of them provide powerful solutions, they have been developed with specific running platforms in mind (e.g. MPI-based clusters) and are not designed, to the best of our knowledge, for dealing out-of-the-box with parallel loops in heterogeneous environments made up of disparate non-dedicated systems and harnessed by users even with limited abilities. Our contribution is focused precisely in this scenario.

As a *proof of concept*, in order to assess the correctness of our proposals, these methods will be implemented creating and evolving the tool R/parallel [VS08], and evaluated with experiments performed reproducing real bioinformatics cases.

1.4. Thesis Outline

The content of this dissertation is organized in 5 additional chapters where we treat the objectives defined in our goals, extending the required background concepts selectively for the issues and problems raised in each chapter. Next, the titles of these chapters and a summary of their content are provided:

- **Chapter 2: Characterizing the Running Environment**

In this chapter we review the classical models and paradigms found in reference literature of parallel computing to understand the context where the running environment considered in this dissertation fits. That includes reviewing classical models regarding the development of parallel applications, common architectures found in parallel systems and general scheduling strategies used to map parallel applications with multiple processing units. Next, a detailed description of parallel loops is provided, followed by a review of major solutions of Desktop Grids, which are a feasible alternative for supplying additional sources of computing resources with reduced investments. Finally current contributions of parallel computing with R are reviewed. This information provides the required background to identify the existing options for designing a solution for the problems of our target running environment.

- **Chapter 3: R/parallel – A Proposal for Parallel Loops with R**

Here, after considering the information and issues reviewed in the previous chapter, a proposal to execute concurrently parallel loops with R is provided. The first outcome is a prototype, called R/parallel, which implements several mechanisms and methods that enable the parallel execution of loops without dependencies on symmetric multiprocessing computers like for example multicore desktop computers. Next, the prototype is extended to enable the utilization of distributed computing resources. The details of its design and implementation are provided, explaining how parallel loops can be parallelized in R, from how parallelism is expressed in original sequential programs to how remote computers are reached and independent work units are defined and managed.

- **Chapter 4: Parallel Loop Scheduling**

In this chapter scheduling schemes suitable for parallel loops are reviewed. Its main characteristics are described, discussing the benefits and drawbacks when applying them to different running environments. Next, taking into account the limitations identified as well as other restrictions and considerations introduced by our target running environment, a new scheme is proposed. This scheme, called ATLS, which stands for Adaptive Turnaround-

based Loop Scheduling, provides improved performance with respect to previous well-known schemes when applied to changing environments made up of non-dedicated computers, while still producing good results in static environments of homogeneous and heterogeneous computers.

- **Chapter 5: Evaluation of Proposals**

Here, our proposals are evaluated using experiments which reproduce real bioinformatics cases. First, the prototype is evaluated from different points of view. Initially, several parallel loop based R programs are parallelized using R/parallel, and its results analyzed using symmetric multiprocessing machines like desktop multicore computers. These experiments allow us to determine the feasibility of the proposed transformation methods. Next, the results obtained using several well-known parallel loop schedulers with different configurations of distributed computers are evaluated. This allows us to understand the strengths and weaknesses of the tested schemes in different situations. Finally ATLS is evaluated, using different scenarios and comparing its results against the previously tested schemes. The results validate our proposals, and allow us to identify new areas for further improvement.

- **Chapter 6: Conclusions and Future Work**

This chapter gathers the main conclusions of this work and reviews to which extend we have been able to accomplish the objectives defined initially. The contributions provided as results of this thesis are exposed, highlighting the specific methods and mechanisms developed to achieve the proposed goals. In the final chapter of this dissertation we also enumerate the open lines for further research that can be undertaken to continue the work started with this thesis. Special emphasis is done in support for fault tolerance, an important aspect to take into account when using non-dedicated environments that is not covered in this dissertation.

Chapter 2

Characterizing the Running Environment

2.1. Introduction

In previous chapter it has been introduced that this work is concerned with the execution of parallel loop based applications using non-dedicated computing environments with R. Before we are able to continue the research for methods that will eventually allow us to reach the goals defined, we need to understand the internal characteristics and the existing possibilities for the components of our running environment. By reviewing the available parallel computing abstract models related with our work it will possible to restrict and focus our research to those areas where further contributions are needed.

Parallel loop based applications, like any other parallel application, can be decomposed in independent work units or tasks that can be executed simultaneously in several processing units [FK99]. The relation between these tasks and the processing units where they are executed is defined by the scheduling model found in a given running environment. A scheduling model consists of a scheduling policy, a program model, an architecture model and a performance objective.

In our case, the performance objective is to minimize the completion time of our users' parallel loop based applications, so given a workload, process all the iterations in the shortest possible time. Other performance objectives include the optimization of system efficiency, system throughput or average response time. The program

model allows us to classify the parallel application being executed while the architecture model provides a classification for the computer resources being used for execution. Finally, the scheduling policy defines the set of rules used to schedule independent work units or tasks to processing units. Therefore, in order to understand our running environment, as it is depicted in Figure 4, next sections describe existing program and architecture models followed by a generic categorization of scheduling policies, identifying those aspects that apply to our work.

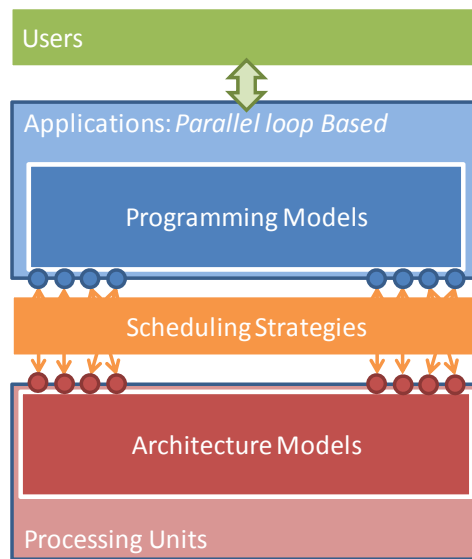


Figure 4: Characterizing our running environment

Next, after reviewing the general models and strategies followed at each layer, the background specific for our target environment, i.e. bioinformaticians with limited skills and resources, running parallel loop based applications programmed in R using non-dedicated computers, is provided. First, to understand the class of applications used by our target users, a detailed description of parallel loops, describing its main characteristics is provided. Second, as a source of additional processing units, desktop grid strategies are reviewed and an adoption of these ideas is proposed. Finally, current contributions of parallel computing with R, in the context of the models exposed in this chapter, is provided. The chapter concludes with some remarks about the concepts described and its influence in our future proposals.

2.2. Parallel Architecture Models

Parallel machines provide multiple processing units arranged in several ways. In general, the relation between these processing units with their memory and I/O subsystems, and how these components are interconnected define several architecture

models [CG+97]. There are also other particular parameters that can be used to classify these machines. For example, Flynn [Fly66] introduced the best-known taxonomy that defines four different types of computers based on whether there are multiple data streams and/or multiple instruction streams. A conventional uniprocessor computer has a single instruction stream and a single data stream and is denoted SISD. Common parallel computers have both multiple data and multiple instruction streams and are called MIMD. The single instruction but multiple data streams computers are denoted as SIMD. The fourth possibility, multiple-instruction single-data stream or MISD, is not used. A taxonomy for MIMD architectures is not clearly defined but the most accepted classification is based on whether the memory is shared or distributed. Consequently, based on memory address space organization the following classification is provided:

- **Distributed-memory model.** In this model, also known as a shared-nothing model [DF+03], several computers connected by a network are used. Each processor executes a separate set of instructions using its independent local memory, for example processes running in separate computers. The memory of the parallel system is distributed throughout the processors rather than being placed in a central location. A network connects processors and their local memories. Processors exchange data between their memories when a remote variable value is required. It is considered that modern distributed-memory parallel computers started with the work of Seitz [Sei85]. Figure 5 illustrates a simplified distributed-memory model.

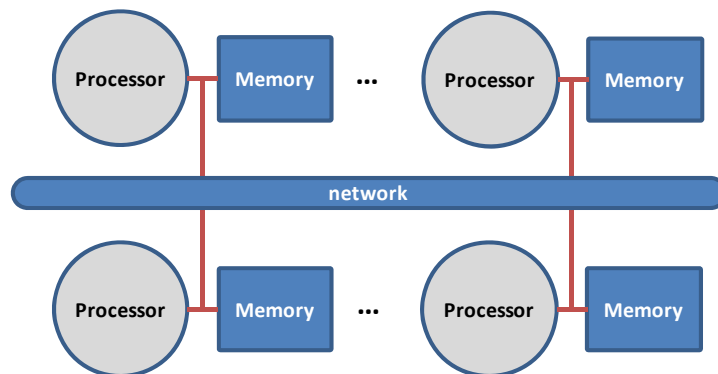


Figure 5: Distributed-Memory Architecture Model

- **Shared-memory model or multiprocessors.** In this model all the processors share access to a central memory. Users do not need to be concerned about the location where the data is stored since all the processors access the same memory space. Because access to the memory is done through load and store operations rather than network operations used in distributed-memory systems, access to shared memory has lower latency and higher bandwidth.

The problem here is that simultaneous access to data must be controlled to ensure data integrity.

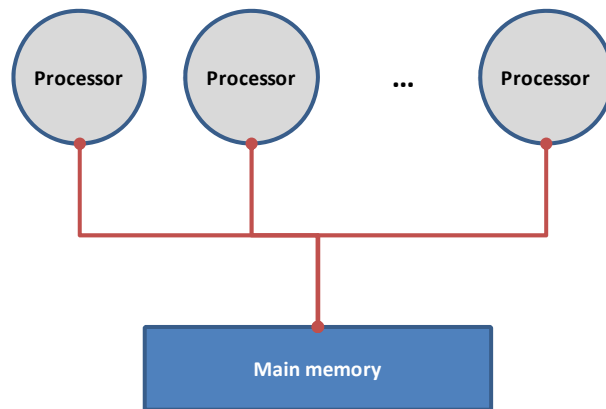


Figure 6: Shared-Memory Architecture Model

A different classification can be done from a management and organization model perspective [FK99] we can classify computer systems in four types:

- **The End System:** Individual end systems are characterized by a small scale and a high degree of homogeneity and integration (Figure 7). They represent the simplest system organization, where the operating system has absolute control over the resources of the computer, controlling process creation, system signal delivery and processor scheduling. Examples of this range from multicore desktop computers to expensive supercomputers.

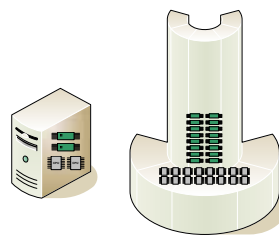


Figure 7: Individual End Systems

- **The Cluster.** This consists of a collection of computers or end systems, usually identical, connected by a local area network (LAN). It is managed by a single administrative entity, in a head node, that has complete control over each end system (Figure 8). At this level, its main functionalities concerning resources management are job submission, job scheduling and parallel process creation. Examples of these organizations include compute clusters managed

by batch queue systems like PBS [HT96] that decide where submitted jobs are being processed.

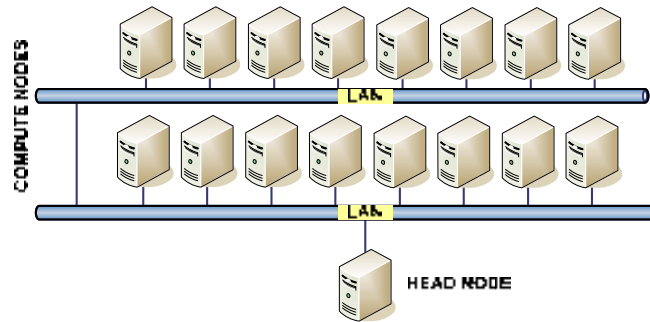


Figure 8: Cluster of Computers

- The Intranet.** It comprises a large number of computing resources that nevertheless belong to a single organization and there is no single site for coordination (Figure 9). It introduces the additional issues of heterogeneity and geographical distribution. Resource management should in addition handle resource discovery, signal distribution networks and provide high throughput to attend the demands of multiple clients of the system. Uniform access for computing resources can be provided in a similar way as it is solved with clusters but attending its increased variability and remote locations with distributed queuing systems such as Condor [LL+88].

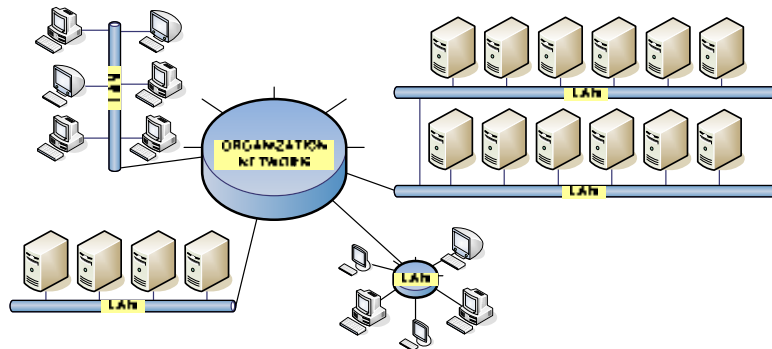


Figure 9: Intranet of heterogeneous computers belonging to the same organization

- The Internet.** These systems lack centralized control, are extremely heterogeneous and are widely distributed geographically. They use a public network that has variable behavior depending on several conditions and demands, such as variable congestion and router status. Individuals and organizations are interconnected through that network, which virtually connects millions of computers. Resource management should add brokers, negotiation and trading issues. Examples of internet based systems are Grid systems like those using Globus [FK97].

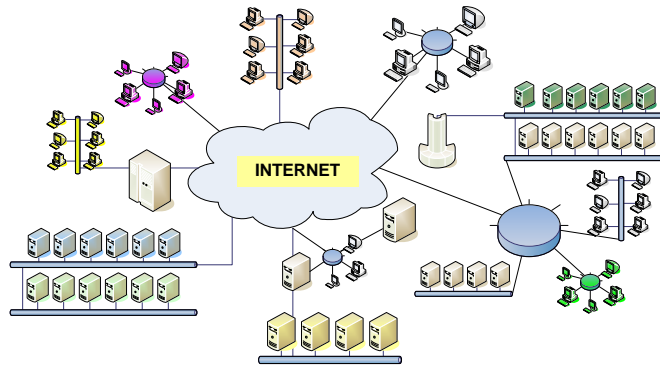


Figure 10: Internet connects unrelated computers widely distributed geographically in very large numbers

2.3. Parallel Programming Paradigms and Models

Parallel applications encode algorithms intended to be run concurrently using several processing units. Based on the internal control structure of the implemented algorithms these can be classified into several parallel programming paradigms [Han93]. Looking at the available technologies for coding a parallel application that enable the exposure of its parallelism and helps to exploit concurrency using a parallel system we can identify different common parallel programming models. Therefore, different programming models, depending on their technical capabilities allow the implementation of different parallel programming paradigms.

Levels of parallelism can be detected at several levels, and these can be based on variable lumps of code (and grain size) that can be a potential candidate for parallelism [Buy99]. Table 1 lists the categories of code granularity and its related parallelism. All approaches dealing with instruction parallelism (very fine grain) have as common goal to boost processor efficiency by hiding the latency of lengthy operations such as memory or disk access. To conceal latency, there must be another activity ready to run whenever a lengthy operation occurs. The idea is to execute concurrently two or more lumps of code having multiple tasks being processed simultaneously. The same concept can be extended to larger grains where for example the issue with local memory access latency is exchanged with other elements at a higher level like remote data replication of process synchronization.

Grain size	Code Item	Parallelism at	Parallelized by
Very Fine	Instruction	Multiple instruction issue	Processor
Fine	Loop/Instruction block	Data level	Compiler
Medium	Standard One Page function	Control level	programmer
Large	Program-separate heavyweight process	Task level	Programmer

Table 1: Code Granularity and Parallelism

The choice of paradigm is determined by the type of the available parallel computing resources and by the type of parallelism inherent in the problem. The computing resources may restrict the level of granularity that can be efficiently supported by the system. The type of parallelism reflects the structure of either the application or the data and both types may exist in different parts of the same application.

2.3.1. Parallel Programming Paradigms

Several authors propose slightly different programming paradigm classifications. Here we use a generic classification, as a superset of previous classification, proposed by Buyya [Buy99]:

- Single Program Multiple Data (SPMD).** In that model, each process executes basically the same piece of code but on a different part of the data. This means splitting the application data or workload among the available processors participating in the computation. This type of computation is also referred to as geometric parallelism, domain decomposition or data parallelism. SPMD applications can be very efficient if the data is well distributed among the processors and the system is homogeneous. Additionally, when global communication is not required, it can also provide good scalability results. If the processors support different workloads, then the paradigm requires the incorporation of some load balancing scheme able to adapt the data distribution layout at runtime execution. Recovering Flynn’s taxonomy, since the single program has branches and other control-flow constructs, SPMD is a subset of MIMD, not a subset of SIMD programs [DF+03]. Figure 11 shows the basic structure of this paradigm.

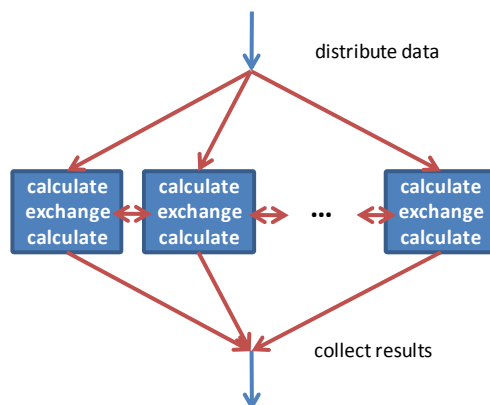


Figure 11: Basic structure of an SPMD program

- Master-Worker** (or Task farming). This model consists of two related entities: one master or host and multiple workers or nodes. The master component is responsible of decomposing the problem into small tasks that distributes among workers to let them process these tasks simultaneously, so it also controls the workers. Each task can perform a completely different algorithm, so submitted tasks do not necessarily share the same code among workers. If tasks share the same code, different sections can be run based on the workers identification. The master can optionally participate of the computation. Once workers finalize the execution of their tasks they send the results back to the master who collects them and produces the final results. Scalability is limited by the single point of communication defined at the master component. Figure 12 illustrates this paradigm.

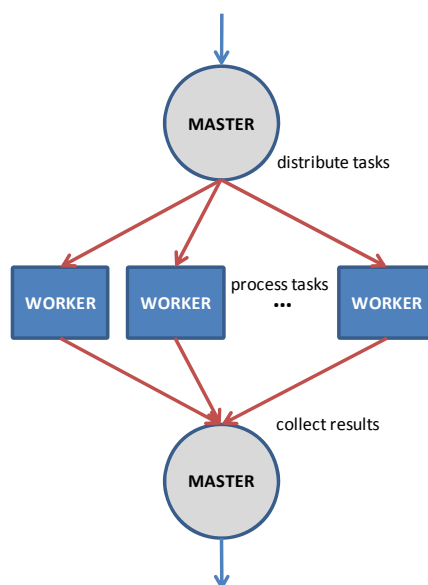


Figure 12: A Master-Worker paradigm

- Data Pipelining.** This paradigm is based in a functional decomposition of an algorithm where a different function is encoded at each stage. Data is processed through the different stages of a pipeline, the output of one providing the input for the next stage. All the stages, usually implemented as separated processes, are running simultaneously and the data flows along the stages of the pipeline. Figure 13 illustrates this paradigm.

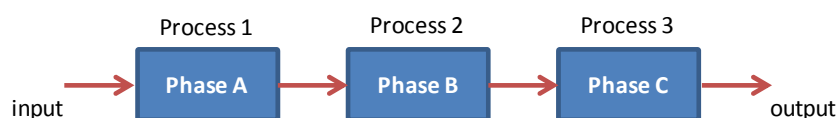


Figure 13: Data pipeline structure

- Divide and Conquer.** A problem is divided into two or more sub-problems. Each of these sub-problems is solved independently and their results are combined to produce the final results. If the sub-problems, smaller instances of the original problem, require additional decomposition, then a recursive division is produced. The tasks are actually computed by the processes at the leaf nodes of a virtual tree. Three generic operations are needed for the divide and conquer paradigm: split, compute and join. Figure 14 illustrates this structure.

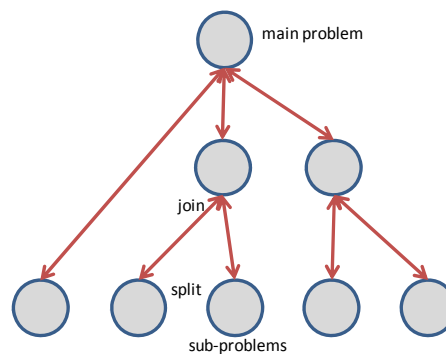


Figure 14: Divide and conquer as a virtual tree

- Speculative parallelism.** In here several processes, coding different algorithms, are processed without the certainty that they will produce valid results. This model is used in several situations and with different objectives when the others cannot be applied. For example, in some cases, some restrictions of the original problem are relaxed in an optimistic assumption, and the processing is launched, expecting that the concurrent execution will not violate the consistency of the problem. Other situations that use this model appear when it is not clear which algorithm will be the fastest to produce correct results. In that case different methods are executed concurrently and the results of the first finalizing process are kept (Figure 15).

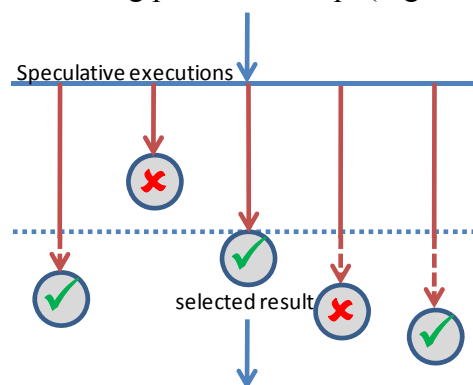


Figure 15: Speculative parallelism

2.3.2. Parallel Programming Models

A parallel programming model is a set of software technologies like compilers and frameworks that allows expressing parallel algorithms and matching applications with the underlying parallel systems.

- **Message passing.** This is a widely used programming model. In that case, programmers organize their programs as a collection of processes with private local variables and the ability to send and receive data between processes by passing messages. Examples of libraries that extend general purpose languages to support these parallel programming models are PVM [Sun90] and MPI [MPI93]. Current implementations of MPI like LAM [BD+94] and OpenMPI [GF+04] are extensively used for scientific calculations in large clusters of networked computers. This programming model is usually coupled, but not necessarily, with distributed-memory systems.
- **Shared memory.** In this model, programmers view their programs as a collection of processes accessing local variables and a central pool of shared variables. Each process accesses the shared data by asynchronously reading from or writing to shared variables. As more than one process may access the same shared variables at the same time, mechanisms to resolve mutual exclusion problems need to be provided, such as locks or semaphores. This model is adequate in multiprocessor computers with uniform access to main memory. An example of technology to support this model is multithreading, where light versions of processes called threads run simultaneously while having shared and private memory regions. Multithreading implementations (e.g. POSIX threads) are currently available in most modern operating systems.
- **Parallel languages.** Several languages have been designed to enforce straightforward development of parallel applications. Examples of current proposals are IBM's X10 [ES+04], Sun's Fortress [AC+10] and Cray's Chapel [CC+04]. While the goal of these parallel programming languages is to increase the programmers productivity for large-scale parallel computing platforms, by efficiently capturing the inherent parallelism at early stages of the software development cycle, the requirement of having to learn a new language platform prevents many researchers from adopting these kind of solutions. They really would prefer to use their traditional high-level languages like C or Fortran and try to recycle their already available software.

For these programmers, extensions to existing languages or run-times libraries are a preferred alternative.

- **Automatic compilers.** Parallelizing compilers are still limited to applications that exhibit regular parallelism, such as concurrency in loops. Since it is quite difficult to automatically and safely identify parallel regions, other semi-automatic proposals where the user provides a hint of *what* to parallelize but not *how* have been developed. Examples are HPF [Lov93] for Fortran programs and OpenMP [DM98] initially for C and Fortran programs. In general, they have been proved to be relatively successful for some applications on shared-memory multiprocessors, but still they do not succeed in the same degree when applied to distributed-memory machines. The difficulties are due to the non uniform access time of memory in the latter systems. The currently existing compiler technology is therefore limited in scope and only provides adequate speedup with specific problems and running environments.

2.4. Scheduling Strategies

Scheduling functionality can be divided into external and internal, two-level scheduling, or generally termed mapping and scheduling [FK99]. We adopt and describe the first terminology:

- **External Scheduling.** This is concerned with the assignment or mapping of applications to compute resources, i.e. resource allocation or resource partition assignment.
- **Internal Scheduling.** This is involved with the assignment of tasks belonging to an individual computation or application to the processing units assigned to that application, that is, with the use of the resources already allocated in a given partition.

Regarding *external* scheduling, as it is stated in [FR+97], each parallel application is executed in a partition that consists of a number of processors. The size of the partitions depends on the computing resources available, the application and the current load of these resources. The size of a partition may change during a computation lifetime. Different partitions are defined:

- **Fixed.** The partition size is defined by the system administrator and cannot be modified unless updating the system configuration.

- **Variable.** Size is determined at submission time, based on user request and system capacities.
- **Adaptive.** Similar to variable size, but in that case, based on the same information and including current system load, the scheduler decides the final partition size.
- **Dynamic.** The partition size may change during the execution of a computation, to reflect changing requirements and resource availability.

Once a partition has been allocated for a parallel application, *internal* scheduling must be considered. Basically it can be divided in:

- **Static Scheduling.** It consists in assigning all application tasks to compute resources before execution begins. Once a task has been assigned to a machine or processing unit, it will finish its execution on that machine, unless the machine in question fails.
- **Dynamic Scheduling.** In that case tasks are assigned to compute resources as tasks are being created or as resources become available, i.e. when either a new work unit appears or when a new processing unit joins the computation a new scheduling decision is taken. In contrast to static scheduling where the information required to define the assignments is known beforehand and so the schedules are predefined at compile time, dynamic methods must react at runtime, requiring in some cases to modify an assignment even during computation. Since initial assignments cannot be as efficient as possible, load balancing mechanisms, to redistribute tasks and hence balance processors, are introduced. To achieve this, two methods are used: preemptive, able to suspend, reassign and resume tasks, and non-preemptive, where once a task is assigned must finalize or fail. An important non-preemptive dynamic scheduling strategy is represented by self-scheduling schemes [TY86].

Self-scheduling is used to schedule a set of parallel tasks that are independent iterations of a computational loop. The loop iterations are divided into a set of tasks, and these tasks are placed in a single global work queue that is available to all the processing units involved in the computation. When a processor becomes available, it removes tasks from the work queue for processing. In distributed environments, as it is described in [CA+01], it can be implemented using a master-worker paradigm (Figure 16).

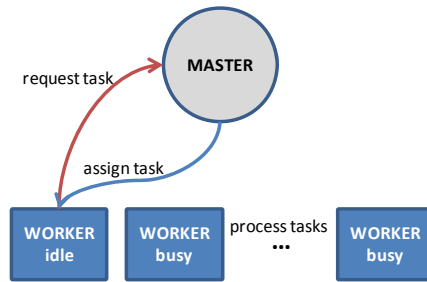


Figure 16: Self-Scheduling using the Master-Worker Paradigm

In such generic self-scheduling schemes at the i -th scheduling step, the master computes the chunk size K_i and the remaining number of tasks R_i :

$$R_0 = N, \quad K_i = f(R_{i-1}, P), \quad R_i = R_{i-1} - K_i$$

Where $f(\cdot)$ is a function possibly of more inputs than just R_{i-1} and P . Then the master assigns to a worker node K_i tasks. Imbalance depends on the time gaps t_j for $j = 1, \dots, P$ between the final total execution time of each worker. This gap may be large if the first chunk is too large or, more often, if the last chunk (called the critical chunk) is too small. The different ways to compute K_i has given raise to different scheduling schemes. The most notable examples are described in chapter Chapter 4. A variant of self-scheduling schemes include the “bag-of-tasks” policy [BS95] of the Linda system [Gel95] and eager scheduling used in Calypso [BD+95].

2.5. Classifying Parallel Loops

With the introduction of the first commercial multiprocessor computers, during the late 70’s and 80’s, it was obvious that any program using a significant amount of computer time usually spends most of the time executing one or more loops. At that time the most widely used language for scientific programming was FORTRAN, hence most of the references found in literature at that time (e.g. [Lam74, PW86]) propose new methods implemented as extensions of FORTRAN, where the main interest was loop parallelism using multiprocessor computers.

Looking at the data dependencies between iterations, which determine to which extent a loop can be executed in parallel, allows the classification of loops in three major classes, following the FORTRAN terminology [Pol88]: DOSERIAL, DOALL and DOACROSS loops.

DOSERIAL or sequential loops are found when each iteration is processed strictly one after the other, either because there is only one available processor or because the

data dependencies does not allow any other method to overlap the execution of the iterations, even when several processing units are available. In contrast, two types of parallel loops are DOALL, which describes a totally parallel loop with no dependence between iterations, and DOACROSS, which supports parallelism in loops with dependences between iterations by delaying the execution of subsequent iterations.

A DOALL [Lun85, PK+80, Pol88] loop has no data dependence relations between iterations, thus the iterations of the loop can be executed in any order, including in parallel. Scheduling iterations to execute in parallel can be done without any consideration for correctness (since any order is correct), only considering performance. Other algorithms show dependence relations preventing such straight parallel execution of the loop, so other parallel constructs are needed to support these algorithms. An example illustrating a DOALL parallel loop with a matrix multiplication follows:

```

DOALL I = 1, N
  DOALL J = 1, M
    C[I,J] = 0.0
    DO K = 1 to L
      C[I,J] = C[I,J] + A[I,K]*B[K,J]
    ENDDO
  ENDDOALL
ENDDOALL

```

It can be observed that the calculation of the result variable do not depend of previous calculations performed in DOALL loops, so they can be processed in any order. Figure 17 illustrates the basic structure of a single DOALL loop with index variable i .

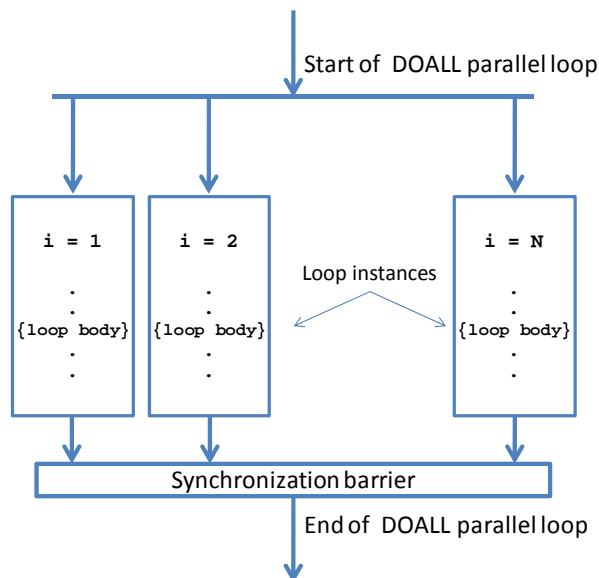


Figure 17: DOALL loop with index variable i

Each of the parallel code blocks are conceptually identical copies of the statements within the DOALL construct, i.e. they share the same loop bodies. Once an execution begins, each block of statements will be executed concurrently in independent loop instances, each with a different value of i and each potentially following a different, data-dependent path through the instance. No synchronization is assumed at launch time and the range of values assumed by the index variable is known upon entry to the loop. It can be observed the homologous structure with the SPMD model. If more instances exist than processing resources then some processors execute more than one instance. Control is not passed to the next portion of the program until all instances of the DOALL construct have been executed.

Besides of data dependency, other restrictions apply to the loop body in order to guarantee the correctness of the loop execution [Lam74]. We summarize these restrictions in the following list of assumptions:

- A1. It contains no I/O statements, except for the case of using random access methods to independent positions that guarantee independent I/O operations between iterations.
- A2. It contains no transfer of control to any statement outside of the loop body, otherwise the loop instance are not identical any more.
- A3. It contains no function calls, except for the case that it is confirmed they do not violate any of the previous restrictions.
- A4. Any occurrence in the loop body of a new value is not generated from values generated in previous iterations, i.e. it does not exist cross-iteration data dependences.

For the case of the assumption A4 an additional exception can be done for the variables used to aggregate partial results with a given operation like sums across iterations. These variables and operations are called respectively reduction variables and reduction operations, since they collect the partial results to produce a final *reduced* version of them. When these operations allow the recalculation of the final values based in subsets of partial results, then a divide and conquer approach can be applied to tolerate these cross-iteration data dependences and keep using parallel processing of loops as described.

The DOACROSS loop was proposed when parallel execution is desired in spite of dependence relations [Cyt86]. The model of execution for a DOACROSS loop is that each iteration is initiated in sequence, but the initiation of iteration i is delayed from the initiation of iteration $i-1$ by some amount of time. In the model, the delay is calculated to allow the source of any (actual or potential) dependence from iteration $i-1$ (or any prior iteration) to execute before the sink of that dependence executes in iteration i . The key element of a DOACROSS is that the dependence relations will

always be satisfied *forward in time*, corresponding exactly to the sequential model of execution. This has the benefit that every execution, sequential or parallel, is deterministic. A trivial example to better understand the DOACROSS loop follows [Cyt86]:

```
DOACROSS I = 1 to N
    A[I] = B[I-1] + 37
    B[I] = A[I] + C[I-1]
    C[I] = B[I] + D[I-1]
    D[I] = C[I] + E[I-1]
    E[I] = D[I] + 77
ENDDOACROSS
```

In this loop, there is a lexically backward loop-carried dependence from each statement to the previous statement. If each statement takes one time unit to execute and the first iteration is started at time $T=1$, the second iteration could safely start execution at time $T=3$, and each subsequent iteration i could start execution at time $T=2i-1$. Since there are only five statements (taking 5 time steps per iteration), and the delay between iterations is 2, the maximum parallelism in this loop is 3.

Parallelizing compilers can always turn sequential loops into DOACROSS loops by inserting the appropriate delays or synchronization points. However, as it is shown in the previous example, the maximum parallelism obtained for a loop can severely reduce the number of processing units simultaneously used, thus hindering the possible speedups to be achieved. Next example shows an even worse case:

```
DOACROSS I = 1 to N
    A[I] = B[I] + 37
    C[I] = C[I] + D[I]
    S = S + A[I]*C[I]
ENDDOACROSS
```

Although a DOACROSS loop always satisfies dependence relations in the same way that sequential loops do, even introducing accumulator variables (or in the general case, reduction variables) like the S variable, the expected performance improvement can be jeopardized due to the parallelizing overhead and even produce worse execution times than if run sequentially without changes.

Additionally, the DOACROSS loops require a shared memory model where modified variables are updated in a central location so that subsequent iterations are able to retrieve the new values. Although strategies have been proposed and successfully implemented for non-uniform memory architectures (NUMA), including distributed meta-computers, we do not further consider this class of loops.

Since in this work we are interested in problems where very large datasets are processed using programs required in common bioinformatics analysis, the main body of loop types found in these programs perfectly fits with DOALL loops. Additionally, given there is no cross-iteration data dependence, the maximum parallelism obtained is not limited by the loop internal statements.

Finally, looking at the time spent to process each iteration of a loop, supposing homogeneous processing units, loops can be classified in three additional types [Pla94]. A loop is *uniform* if the execution time of each of its iterations is the same. Examples of this case are matrix multiplication or, in the context of bioinformatics, given a set of id labels, the identification of the sample donors of each DNA sequence of fixed length from a large dataset. A loop is *semi-uniform* when the processing time of the iterations depends only on the index of the loop. An example is a loop processing a set of chromosomes where each one includes a different number of known genes. Finally, a loop is *unbalanced* or *non-uniform* if they depend completely on the data. Pair-wise alignment of a set of variable length DNA sequences against a common database is a bioinformatics example for this case.

2.6. Desktop Grids as an Additional Source for Computing Resources

Desktop Grids are emerging as a suitable option for providing additional computing resources for parallel computing in situations where individuals do not have direct access to dedicated computers where their parallel applications can be run.

Desktop Grid can be seen as a subclass of Grid [FK03] although important differences exist between both. Grid is a relatively new paradigm for high performance or high throughput computing which aims to aggregate dedicated, heterogeneous, large-scale, multiple-institutional, and widely distributed geographically resources, while providing transparent, secure, and coordinated access to various computing resources (e.g. supercomputers and cluster) belonging to multiple institutions joined in virtual organization [FK03]. On the other hand, Desktop Grid is mainly focused in harvesting a number of idle desktop computers owned by individuals on the edge of Internet [And04, CC+03, Sar98], being the most popular example SETI@Home [AC+02].

Scheduling is very important to understand and develop a Grid system. Scheduling in Grid systems is complicated by the heterogeneous and dynamics nature of the type of computing resources involved in their environments. A Grid scheduler consists of a meta scheduler and a local scheduler. The meta scheduler main function is to decide the site where jobs have to be dispatched while a local scheduler is responsible for

dispatching jobs within a set of computers in a single site. Examples of local schedulers are LSF [ZZ+93] and SGE [Gen01]. Scheduling in Desktop Grid is different from Grid because of the different types of resources used, their dedication and availability, results trust, frequency of failures, and kind of applications used. Desktop Grid scheduling, besides of focusing in fault tolerance, volatility, and result certification for malicious resource providers, it mainly uses pull-based scheduling, i.e., a resource provider sends a request to a server in order to get a job or work unit when it is idle, due to the non-dedication property of the participating volunteer nodes. A final characteristic of scheduling in Desktop Grid is that one volunteer node, usually a desktop computer, usually executes one task at a time, mostly independent and without preemption. Therefore, local schedulers like the ones used in Grid systems are not necessary at first instance in Desktop Grid.

One of the first available taxonomies for Desktop Grid systems is provided by Choi [CK+07]. To better understand the characteristics of Desktop Grids and where our proposal, R/parallel, can fit within this context, next we summarize the most important elements of this taxonomy.

2.6.1. Taxonomy of Desktop Grid

Desktop Grid is mainly categorized according to characteristic of their organization, platform, scale, and resource provider properties. Next we describe each of these categories:

Organization. Desktop Grid is categorized into centralized and distributed systems according to the organization of its components.

- *Centralized Desktop Grid* consists of client, resource provider or volunteer, and server components. A client is able to extend its processing capacities by submitting parallel jobs or work units to a central server. A resource provider or volunteer donates its computing resources, usually during its idle time by first informing the server about its availability. A server is a central component that accepts submitted jobs, divides them into sub-jobs or tasks, and schedules them to volunteer nodes. Once a volunteer processes the assigned task, it returns the result to the server. Additionally the server can check the correctness of the results and then returns the final result to the client. Typical examples are BOINC [And04] and Entropia [CC+03].
- *Distributed Desktop Grid* consists also of client and volunteers. In contrast to centralized Desktop Grid, there is no central server. Volunteers have partial information of other volunteers, in a peer-to-peer like relation, becoming

clients when needed, and are responsible for constructing their computational networks and schedule their jobs in a distributed way. An example with this organization is Organic Grid [CB+05].

Platform. Desktop Grid can be categorized into web-based (ActiveX and Java applet based) and middleware-based according to the platform used by the volunteer nodes.

- *Web-based* platforms are characterized by client side parallel applications for example written in Java and post as an applet on the Web. Volunteers only need to visit the web page hosting the applet with their browsers to volunteer their resources and join the parallel computation. The web component has all the information to contact back a central server to retrieve work units and perform its encoded functions. An example of volunteer computing based in Java web clients is the Bayanihan system [Sar98].
- *Middleware-based* platforms are characterized because volunteers need to install and run a specific middleware, i.e. a software layer that provides the services and functionalities to execute parallel applications, on the volunteer's computer. Typical examples, besides of the well-known SETI@Home screensaver, are BOINC [And04] and Entropia [CC+03].

Scale. Desktop Grid can also be categorized into Internet-based and LAN-based ones according to scale.

- *Internet-based* Desktop Grid is mainly based on anonymous resource providers, although identified volunteers, interested on receiving public recognition for their collaboration are also included. It should consider firewalls, network address translation, dynamic address, poor bandwidth, and unreliable connections. They provide potentially the largest number of computing resources but there is also an important competence between projects (and its parallel applications) that tries to attract to their servers the highest number of volunteers. So, for small and not very popular projects it is hard to aggregate and maintain large numbers of volunteers.
- *LAN-based* Desktop Grid is based on resource providers within a corporation, university or institution that share a private network. It has more constant connectivity than Internet-based alternatives and lower network latencies. Besides, resources are usually only shared among the users of a local network, showing less competence and shorter access time. As a consequence, LAN-based Desktop Grid systems can reach their full capacity in shorter times than Internet-based ones and are therefore better suited for execution of

applications with moderated workloads (e.g. up to several hours or few days). Finally, an additional advantage is that, given the proximity of the involved volunteers and its expected higher level of trust, measures for security and result validation can be relaxed, simplifying the development of parallel systems on these environments.

Resource Provider. This final category is based on the volunteer or enterprise properties of the resource provider.

- *Volunteer Desktop Grid* is based on voluntary participants. With Internet-based systems, it is more volatile, malicious, and prone to faults than Enterprise Desktop Grid but it provides the cheapest option for accessing additional resources. Typical examples are BOINC [And04] and Bayanihan [Sar98].
- Enterprise Desktop Grid is based on non voluntary participants usually within a corporation or university responsible for the maintenance of their own computing resources. Since institution-wide policies can be established within an organization it is possible to manage the provisioning of an homogeneous software layer to create these distributed systems. It is therefore more controllable than Volunteer Desktop Grid because its resource providers are located and controlled within the same administrative domain. A typical example is the Condor platform [LL+88].

Finally we provide a table (Table 2) where we compare the examples indicated previously, including, with the same parameters, the options chosen for our proposal, R/parallel.

System	Organization	Platform	Scale	Resource Provider
BOINC [And04]	Centralized	Middleware based	Internet	Volunteer
Entropy [CC+03]	Centralized	Middleware based	LAN or Internet	Enterprise or Volunteer
Bayanihan [Sar98]	Centralized	Web based or Middleware	Internet	Volunteer
Condor [LL+88]	Centralized	Middleware based	LAN	Enterprise
Organic Grid [CB+05]	Distributed	Middleware based	Internet	Volunteer
R/parallel [VS08]	Distributed	Middleware based	LAN	Volunteer

Table 2: Comparison of R/parallel within the context of some popular Desktop Grid Systems

The main idea, as it can be deduced from the previous table, is to provide distributed mechanisms to interconnect a client with several volunteer nodes. Since our users are already using the R interpreter for running their applications, by extending the R functionalities, a middleware approach could be used. The LAN scale is the most suitable option for our purposes given its proximity between users, its relaxed implementation requirements, and their better correlation with single users' size of problems (i.e. a extremely large number of computer resources should not be required). Finally, a best-effort environment made up of volunteered computers is considered, taking advantage of the existing resources at a given time, no matter the number and the variable capacity of the available resources.

2.7. Parallel Computing with R

Using R together with parallel computing is not a trivial task as the language does not provide mechanisms to support it natively. To compensate for this lack, several tools have been developed with different degrees of success. These tools are mainly implemented as R packages, the mechanism used by the R system to include libraries with additional functions to extend the language. R packages, besides of being coded in R itself, also allow the extension of R programs linking compiled extensions written in Fortran, C and C++ [RD08]. A review of current state of the art in parallel computing with R can be found in [SM+09]. Next we provide a classification describing representative examples of R packages for several classes taking into account the supporting parallel programming models and the enabling technologies provided.

2.7.1. Packages Supporting the Message Passing Model

Early contributions to parallel computing with R were based on available general purpose parallel computing frameworks and libraries like MPI [MPI93] and PVM [Sun90] that supported the message passing model (described in section 2.3.2). These libraries provide low-level programming interfaces that enable the codification of parallel programs without scope restrictions. However the complexity of this programming model hinders a wider use and adoption of these libraries. Examples of packages implementing these solutions are: `rpvm` [LR02], `Rmpi` [Yu02] and `snow` [TR+09]:

- **`rpvm` [LR02]** . The `rpvm` package is an interface to PVM [Sun90]. It supports low-level PVM functions from R. There are no additional R functions with high-level operations, so the user has to provide all the

communication and parallel execution directives. There is no command to launch R instances at the workers from the master. Instead, the command `.PVM.spawnR()` can be used to execute an R script file at the workers.

- **Rmpi [Yu02]**. The package Rmpi is a wrapper to MPI, providing an R interface to low-level MPI functions. In this way, the R user does not need to know the details of the MPI implementations. It requires a working MPI installation, correctly configured and linked with R, and runs under popular MPI implementations like LAM/MPI [BD+94] and OpenMPI [GF+04]. Rmpi runs on clusters with Linux, Windows or Mac OS X. The Rmpi package includes scripts to launch R instances at the workers from the master (`mpi.spawn.Rslaves()`). The spawned R instances run until closed (with `mpi.close.Rslaves()`). The package provides several R specific functions, in addition to wrapping the MPI API. For example, parallel versions of the R `apply()` like functions are provided by, e.g. `mpi.parApply()`. These functions are used to “apply” a given function to a vector or list of values, both provided as parameters. Another example is `mpi.bcast.Robj2slave()` with which R objects can efficiently be sent to workers. Rmpi also includes some error-handling to report errors from the workers to the manager.
- **snow [TR+09]**. The package snow (Simple Network of Workstations) supports simple parallel computing with R. The R interface supports several different low-level communication mechanisms, including: PVM (via the `rpvm` package), MPI (via Rmpi), NetWorkSpaces (via `nws` [NWS09]), and raw sockets (useful if PVM, MPI or NWS are not available). This means it is possible to run the same code using snow functions at a cluster with PVM, MPI or NWS, or on single multicore computer. The snow package includes scripts to launch R instances on the cluster nodes (`cl <- makeCluster()`). The instances run until they are closed explicitly with the function `stopCluster(cl)`. The package provides support for high-level parallel functions like `apply()` and simple error-handling to report errors from the nodes to the manager. It also provides basic load balancing functionality. `clusterApplyLB()` is a load balancing version of `clusterApply()`. If the length N of vector values is greater than the number of cluster nodes P , then the first P jobs are placed in order on the P nodes. When the first job completes, next job is placed on the available node; this continues until all jobs are complete. Using `clusterApplyLB()` can result in better cluster utilization than using `clusterApply()`. However, increased communications can reduce the performance. Furthermore, the node

that receives and executes a particular job is non deterministic, which can complicate reproducibility in simulation applications.

2.7.2. Packages Supporting the Shared Memory Model

Shared memory model is commonly tied with symmetric multiprocessing computers where several processors are running concurrently in multiple processors which have direct access to a shared memory space where data can quickly be exchanged (described in section 2.3.2). Packages found in this category mainly focus on one of the two topics related with this model: either using multiple processors available in a single computer or providing mechanisms to share variables among several R instance processes. Two examples of these are the packages `multicore` [Urb09] and `nws` [NWS09]. A description of both follows:

- **multicore** [Urb09]. `multicore` is an R package that provides functions for parallel execution of R code on machines with multiple cores or processing units. It is implemented using a master-worker paradigm. Unlike other parallel processing packages all jobs share the full state of the original R instance when spawned, so no data or code needs to be initialized. Spawning uses the `fork` system call or an operating system specific equivalent. Spawning establishes a pipe between the master and child process. The pipe can be used to send data from the child process to the master component. The package provides high-level parallel functions like `mclapply()` and simple error-handling to report errors from the workers to the manager. The package works on Unix systems, with experimental support for Windows available.
- **nws** [NWS09]. The package `nws` provides functions to store data in a so-called “NetWorkspace” [BC+09], and uses the ‘sleigh’ mechanism for parallel execution. It requires a running NetWorkspace server. `nws` is implemented in the server side using the python language and the Twisted framework for network programming. NetWorkSpaces currently supports the python, Matlab, and R languages, and allows limited cross-language data exchange. There are two basic functions for executing tasks in parallel (`eachElem()` and `eachWorker()`). The package uses the master-worker paradigm, and automatically load balances R independent function evaluations. To enable the manager to communicate with workers, sleigh supports several mechanisms to launch R instances at workers to run jobs: local, secure shell (SSH), remote shell (RSH), load sharing facilities (LSF [ZZ+93]) and web launch.

2.7.3. Packages Supporting Assisted Parallelization

In that category we found R packages designed for fully automated parallelization, semi-automated parallelization and program skeletons, which main objective is to relieve the programmer from dealing directly with parallelization details. This feature is very important since programmers do not need to think “in parallel” when coding their R scripts, so that anyone without proper knowledge of parallel computing can benefit from its advantages. Three examples of each case are described: pR [ML+07], ROMP [Jam08] and SPRINT [HH+08].

- **pR** [ML+07]. This solution consists not only of a simple R package but in a full framework. The key feature of pR is that it dynamically and transparently analyzes a sequential R source script and accordingly parallelizes its execution. The results of partial executions are collected to perform further analysis at run time. The framework is built on top of the R interpreter and does not require any modification to the native R environment. Internally, the MPI library is used for inter-node communication. Figure 18 illustrates the architecture. Two important design decisions must be considered. First, is the introduction of a preprocessing stage to perform dynamic dependency analysis before interpreting R statements and identify tasks and loops that can be parallelized. Parallelizing an entire program at the granularity of individual statements, however, may generate too much scheduling and data communication overhead and hurt the overall performance. This issue is addressed with a selective and asymmetric parallelization model. Instead of generating a symmetric Single Process Multiple Data (SPMD) type of parallel code using one or more “fork-join” sessions, a master-worker paradigm that only “outsources” the expensive jobs (i.e., function calls and loops) to the workers is adopted. All the light-weight operations, such as simple statements and conditional statements that do not contain any loops or function calls are executed locally by the master component.

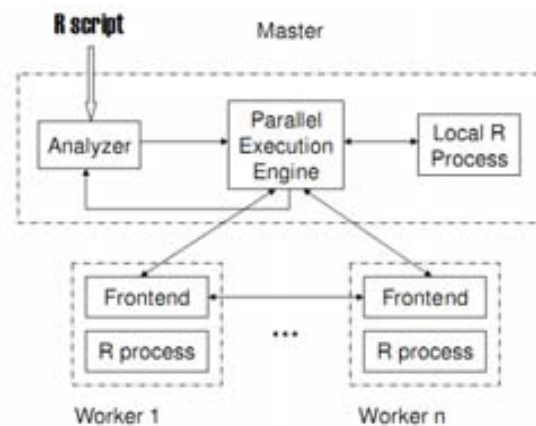


Figure 18: pR architecture (source [ML+07])

- **ROMP** [Jam08]. ROMP provides a binding of the parallel application programming interface OpenMP [DM98] with the R interpreter. Fortran code is generated and compiled on the fly by the toolkit and the OpenMP directives are inserted. The toolkit consists of a family of special apply routines together with reduction routines like sum, mean and product which generate parallel OpenMP code. The toolkit can be used for easy parallelization of parts of an R program without a steep learning curve for the user. Basically it produces compiled code run outside of the R interpreter which, combined with vectorization techniques and multiprocessor computers (with over 100 cores), can achieve for some special problems, as it is claimed by their authors, impressive speedups up to 10000.
- **SPRINT** [HH+08]. This package implements a prototype framework that allows the addition of parallelized functions to R to enable the easy exploitation of High Performance Computing systems. The Simple Parallel R INTerface (SPRINT) is a wrapper around such parallelized functions, hence it provides skeletons of commonly used algorithm structures in the form of reusable R functions. Internally, in a similar way than package snow, it uses the MPI library to communicate processes.

2.7.4. Other Packages Enabling Additional Resources

Other packages have been developed to enable access to additional sources of computing resources. These sources range from homogeneous clusters to other sources of heterogeneous resources accessed through system management software tools like batch queue systems. Next we describe four examples:

- **sfCluster/snowfall** [KP+09]. sfCluster is a small resource manager optimized for cluster solutions with R. It is a Unix tool written in Perl for managing and observing R instances running on clusters while snowfall is an R package based on snow [TR+09] that benefits from sfCluster. sfCluster automatically sets up the cluster for the user, and terminates all running R sessions after execution has finished. On top of that snowfall provides several wrappers for essential snow functions.
- **gridR** [WS+07]. In the context of grid computing the package gridR is the first main contribution. It submits R functions in a grid environment using Globus [FK97], Condor [LL+88], or single servers. The grid infrastructure uses R as interface and client. The server side implementation of gridR uses several external software components: Globus Toolkit, an installation of the R

environment on the grid computers, and a GRMS-Server installation from the Gridge toolkit [PK+06].

- **multiR** [Gro09]. It is designed to operate in highly distributed heterogeneous computing environments such as grids. The main focus is dedicated to security and authentication, simultaneous use of heterogeneous resources and interaction with grid scheduling systems. multiR is provided as a service using a Service Oriented Architecture (SOA).
- **Rsge** [Bod09] and **Rlsf** [SW+07]. These packages provide solutions for accessing a cluster of computers managed by a batch queuing system like SGE [Gen01] and LSF [ZZ+93]. For the case of Rsge, the package offers functions to get information from the queuing system and to submit R jobs to an R cluster handled by SGE. At the manager side the data object is split as specified by the number of jobs. Each data segment along with the function and argument list are saved to a shared network location. Each R instance loads the saved file, executes it, and stores the results in the shared network location. Finally the master script merges the partial results and returns them.

2.8. Concluding Remarks

One of the easiest ways to improve performance in a computer system is simply to replicate entire computers and add a way for separate computers to communicate data [DF+03]. Since one of our goals is to provide additional computing resources to build parallel systems, computers *at the reach* of users can be aggregated and made them transparently available to create a metacomputer [SC92]. A metacomputer can be defined as a dynamic environment that has some informal pool of nodes that can join or leave the environment whenever they desire. Users within an organization should be able to share their computers at their will through a common intranet. The cost is in increasing complexity of the software that has to orchestrate the utilization of these resources while keeping the user interface as easy as possible.

Using distributed-memory computers it seems a message passing paradigm should be adopted. However they require dedicated knowledge of enabling libraries and frameworks like MPI what directly confronts with the type of users we consider. The same applies for shared-memory models and since R does not provide any support for parallel computing, automatic compilers must be observed to define new methods, adapted to the characteristics of our problem, to parallelize user applications based in parallel loops.

Although the programming model has been simplified during the last years, the dependency on external frameworks and dedicated resources is still a major obstacle

for many R users (e.g. pR [ML+07] depends on a complex installation to access a cluster of MPI enabled servers).

Packages like multiR [Gro09] already provide integration capabilities and simplified methods for accessing new distributed resources. Security is also an important topic considered in some R packages, but none takes care of accurate utilization of resources or load balancing in heterogeneous environments, particularly regarding parallel loops.

These solutions are well suited for research groups with access to dedicated infrastructures (e.g. computing clusters managed by skilled technicians) and/or enough time to invest in the development of *ad hoc* parallel programs using available libraries and frameworks. However, when these requirements are not met, solutions based on self-contained tools (e.g. squid for Perl [CG+05]), capable of running in common desktop computers, are the preferred choice.

Regarding parallel loop scheduling, we can always obtain an optimal static schedule for uniform loops. If we have a parallel machine with P identical processors, we must simply partition the set of iterations of the loop into C chunks of size $K=P/N$ iterations each, where N is the number of iterations of the loop. Next, each chunk with K iteration is assigned to each of the processors. This scheduling strategy, together with other scheduling schemes is discussed in Chapter 4, where parallel loops are considered in the context of, not only homogeneous processors found in symmetric multiprocessors but also other running environments like heterogeneous processors found in distributed memory multicomputers. These running environments introduce additional issues to the problem of scheduling parallel loop iterations among several processing units where solutions related with our scope of application are reviewed.

In next chapter we present our proposal, a new R package for parallel computing with R, called R/parallel, that tries to avoid the problems identified at this point. To use it, the programmer does not need to change his algorithm nor install and maintain any additional third party software since the R/parallel package is self-contained and capable of using current multicore processor desktop computers. It easily and effectively enables the semi-automatic parallelization of loops without data dependencies [Bri96], thus bringing the benefits of parallel computing within the reach of any bioinformatician using R. Its design allows its direct use with current bioinformatics analysis tools such as for example R/qlt [BW+03], MetaNetwork [FS+07] or affyGG [AV+08] for analysis of genome-wide gene expression data. As a starting point, we consider uniform loops running on top of identical processing units. However, since one of our goals is to aggregate processors from different types of computer systems, the methods proposed next also include additional requirements to consider non-uniform loops.

Chapter 3

R/parallel – A Proposal for Parallel Loops with R

3.1. Introduction

In recent years, R [IG96] has gained a large user community in bioinformatics thanks to its simple but powerful data analysis language. Growing repositories like Bioconductor [GC+04] and CRAN [RF10] assist bioinformaticians with hundreds of free analytical methods and tools. These user-contributed methods are easily reused and adapted to each particular experiment for analysis of biological data. Examples of often reused and adapted methods are the packages `tilingArray` [HT+08] and `affyGG` [AV+08]. However, while data generated in experiments previously fitted comfortably on a CD-ROM, nowadays, using new equipments hardly fit on a single DVD-ROM. As a consequence of the post-genomic explosion of data, the demand of computational power, as already introduced in this dissertation, is increasing continuously and solutions to keep the *processing pace* of high-throughput devices are required. A common approach in many bioinformatics fields like genomics, transcriptomics and metabolomics, where large sequential data sets are analyzed, is the use of parallel computing technologies [Tre01].

However, in some situations, in order to make use of parallel computing several obstacles and barriers must be removed or mitigated to let users to take advantage of running their applications concurrently. In our case, running parallel loops in R, there are several particular issues to take into account in order *to pave the way* for

newcomers and develop the methods and mechanisms that will enable parallel computations.

In here we focus first in the utilization of single computers with symmetric multiprocessing capabilities, e.g. computers with multicore processors, to later extend the proposed solution to aggregate remote nodes and hence support distributed computing.

The R language interpreter, like many other legacy applications, is a single-thread program that is not prepared out of the box to take advantage of nowadays multicore computers. In order to do so parallel programming techniques are required to adapt the parallel regions of the original program to enable its concurrent execution. Therefore, single-thread legacy applications like the R language interpreter, running on top of current multicore processors, claim for parallel computing support.

Transforming a sequential program to make it able to run concurrently several sections of its code it is a difficult task that can be achieved by several methods, depending on our resources, requirements and limitations [BV+08]. Most of them imply either using a shared memory model, suitable for multiprocessor machines, or a message passing model, which can be used between networked machines. Shared memory based solutions, within a single process, make use of multithreading techniques to run several instances of a single process together with shared variables and inter-process communications (IPC) mechanisms like mutex sections to synchronize its parallel execution. A useful tool to automate the construction of such programs is OpenMP [DM98]. Solutions based on the message passing paradigm also use IPC mechanisms but in this case to communicate different processes, usually but not necessarily, through the network. A well-known library that provides a full set of building blocks to ease the construction of such parallel programs is MPI [MPI93]. There are more solutions that have proved their value with great success, but even in the case these useful tools are compatible with our programming language and running environment, more disadvantages still can appear.

An obstacle that dramatically increases the complexity to adapt sequential programs appears when global variables are extensively used in legacy applications. In these situations, if using multiple threads running at different instructions of the same process, it is quite complicated to ensure the correct access to these variables and avoid race conditions. When correct access order is not ensured, applications are said to be *non-thread-safe*. This is quite common in large legacy applications that have been growing for years while increasing its functionality. An example of this, with more than 10 years of evolution, is the R language interpreter.

A different practical obstacle appears when observing the maintenance lifecycle of legacy applications. After years of proven utility it is logical to expect a long life

span. Introducing an external dependency on a piece of software that later on may get discontinued can cause serious problems in the future.

Another legal obstacle is found between incompatible software licenses. For example, many open-source tools are published using the GNU General Public License GPL [GNU07]. Although partially solved by the less restrictive version Lesser GPL license, the former prevents the utilization of these GPL licensed tools together with proprietary software licenses which were commonly adopted by earlier legacy applications.

Finally, a pragmatic problem comes with the required skills to perform such transformations or adaptations. It is common for scientists to program their own applications. Although when implementing their algorithms, they produce high quality applications, without specific background on software engineering and parallel computing, this process of transformation, due to the lack of knowledge and experience, is very cumbersome and error-prone.

In this chapter we expose the methods and technologies developed to create a solution for parallelizing parallel loops using non-dedicated environments in the R language. Parallelizing loops in R has a double sided meaning. It stands for how do we have to transform the original sequential program and second how do we execute the new parallel programs. The outcome is an add-on R package called R/parallel [VS08].

3.2. Extending the R Interpreter with R/parallel

In this section we describe our proposal for parallel computing with R. In previous sections we have reviewed the characteristics of the class of algorithms we are considering, i.e. parallel loops, learning the main requirements these problems impose. After reviewing current and former proposals for parallel computing with R and comparing them with our target running environment we identify additional requirements that must be considered. Next, and also considering the technical restrictions introduced by R, we describe the design decisions and implementation details of our proposal, the R package R/parallel, initially considering single computers with multiprocessing capabilities.

3.2.1. Expressing Parallelism

The first aspect taken into account when willing to transform a sequential program into its parallel version is the desire to minimize user intervention since one of the premises defined for our running environment is to consider users with severe time

constrains. The perfect solution should not require any further modification from the programmer. This should be achieved with fully automatic parallelizers, which parse the program code, check it for data dependencies and generate a set of independent tasks that can be safely evaluated in separate processors. However, the drawback of this approach is that the parallelizer, *a priori*, does not know the execution time of each subset of statements or independent tasks. When a set of tasks are running concurrently, additional overhead and delays are introduced due to additional processing steps (e.g. code analysis or task coordination). Besides of complex implementation difficulties, that include data dependency detection and statement transformations, the additional overhead introduced by the extra preprocessing steps does not guarantee performance improvements from running the new parallel version of applications. It is quite likely that a sequence of small fast tasks is parallelized and, despite parallel execution, as a result of the transformation process and additional synchronization, the overall execution time can be increased. The conclusion is that current fully automatic parallelization does not provide better results in all situations and alternatives based on the same ideas should be evaluated.

To avoid the problem described with fully automatic parallelization, semi-automatic parallelization is considered. The design decision made is to let users indicate which sections of their programs (i.e. which loops in our case) they need to speed up. The idea is to ask the programmer *what* do they want to parallelize but not *how*. With that minimum information parallelizers are able to transform sequential programs and produce parallel versions. Methods like this have been successfully used in several frameworks, currently being OpenMP [DM98] the most representative of them.

The origins of OpenMP date back to the late 1980s when there was no parallel language standard for shared-memory multiprocessors [DF+03]. To address it an initial effort was undertaken by the Parallel Computer Forum to define a standard syntax for parallel constructs. The standardization process led to the definition of PCF Fortran [PCF91] and eventually to the ANSI abstract interface standard X3H5 [ANSI94]. In summary, PCF Fortran provided two mechanisms for loop parallelism and task parallelism. It also includes a feature called *parallel regions*, which were constructs within which tasks could be created and concurrently executed. Parallel loops were included as a special case of parallel region.

It was not until 1997 when the PCF/X3H5 standard was replaced by OpenMP [DM98], an informal standard parallel programming interface with bindings initially to Fortran and C. OpenMP drew strongly on the ideas from PCF Fortran, and it adopted its directive conventions as in HPF to specify parallelism in a program. As in HPF, OpenMP directives in a program can be ignored as comments by a uniprocessor compiler with no difference in results.

As an example of OpenMP directives, a parallel loop is bracketed by the `PARALLEL DO` and `END PARALLEL DO` directives. The `PARALLEL DO` directive can have a number of qualifying clauses that permit the specification of variables that are private to threads executing individual loop iterations and variables that are used for reduction. An example from [DF+03] follows:

```
!$OMP PARALLEL DO
DO I = 2, N
    APRIME(I) = (A(I+1) + 2*A(I) + A(I-1))*0.25
ENDDO
```

Note that the loop induction variable I is private by default and the `END PARALLEL DO` directive is optional.

The method designed for parallelizing parallel loops in R is inspired in OpenMP and its predecessors. In OpenMP parallelism is exposed by using directives, written in the form of code comments, to indicate parallel regions as depicted in Figure 19.

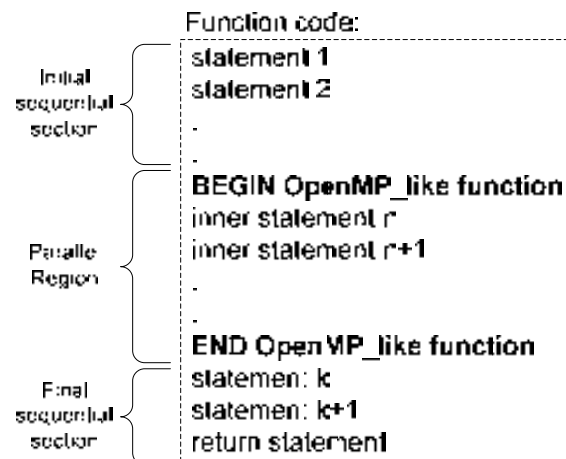


Figure 19: Indicating parallel regions

Trying to translate that to R we found the first differences. Since R is an interpreted language, in contrast to the compiled languages supported by OpenMP, a preprocessing phase should be introduced or transformation steps be performed at runtime. Due to a preprocessing phase would introduce an extra step to run R scripts this is not the preferred option. R programmers are used to run their scripts directly once coded, in many cases interactively, and this extra phase would seem too cumbersome for them. Therefore a mechanism to express parallelism at runtime should be used.

Comments introduced in source code are also not feasible since the R parser discards them when loading and creating function objects. An alternative could be the use of macros or other similar language constructs that allows the introduction of

conditional instructions, only used if some environmental conditions apply, but again R lacks of this kind of structures.

The obvious alternative, reached that point is the utilization of an IF-ELSE structure. The IF section can be used to check if the conditions to perform the parallelization are meet (and eventually start the parallelization), while the ELSE section can be used to indicate the parallel region. An example will help to illustrate our proposal. Suppose a generic R function encoding a parallel loop as the one depicted in Figure 20:

```
yourFunctionName <- function( argument1, argument2=NULL )
{
  # 1. Initializing Values
  internalVar1 <- 0
  reduceVar <- NULL

  # 2. Start of loop
  for(index in 1:nrow(argument1))
  {
    #Make some calculations
    internalVar1 <- someCalculations( argument1[ index, ] )
    tempResult <- moreOperations( internalVar1, argument2 )
    reduceVar <- reduceOperation( tempResult, reduceVar )
  }

  # 3. Finalizing the function
  return( reduceVar )
}
```

Figure 20: Example R function encoding a parallel loop

In order to parallelize the *for* loop we only need to enclose the whole loop body within the ELSE section of an IF-ELSE construct as Figure 21 illustrates:

```
yourFunctionName <- function( argument1, argument2=NULL )
{
  # 1. Initializing Values
  internalVar1 <- 0
  reduceVar <- NULL

  if( "rparallel" %in% names( getLoadedDLLs() ) )
  {
    runParallel( resultVar="reduceVar", resultOp="reduceOperation" )
  }
  else
  {
    # 2. Start of loop
    for(index in 1:nrow(argument1))
    {
      #Make some calculations
      internalVar1 <- someCalculations( argument1[ index, ] )
      tempResult <- moreOperations( internalVar1, argument2 )
      reduceVar <- reduceOperation( tempResult, reduceVar )
    }
  }

  # 3. Finalizing the function
  return( reduceVar )
}
```

Figure 21: Parallel loop indicated with an IF-ELSE construct

Besides of the IF-ELSE structure, a single helping function is used, `runParallel()`, to collect extra information required for the transformation process. This method has the following characteristics:

- Environmental conditions can be checked in the IF section (condition statement and body) and decide if the parallelization can take place.
- Extra information about the exposed parallel region can be collected to indicate what to parallelize (e.g a parallel loop with a reduction variable called `reduceVar`), and eventually if desired, adjust the parallelization (e.g. which scheduling method should be used at runtime).
- This method can be used together with different parallel structures. Although our current implementation is restricted to parallel loops coded with R, it can be extended to other parallel structures, for example it could be used to indicate several independent expressions that should be evaluated simultaneously by several processors (i.e. task parallelism).
- From the `runParallel()` function call the parallelization process can be undertaken.

Another aspect to consider when developing parallel programs is the difficult task of debugging when coding errors arise. When multiple processing units are running concurrently at different steps of a program, the identification of the conditions that triggers a bug and the retrieval of the state of each execution thread is a cumbersome task that should be avoided. To minimize this risk, an objective of the design of this package, and enabled by the proposed method, is the ability to run the sequential (and parallel) version of the R programs without changing any further line of code (besides of the IF-ELSE added). By running a program sequentially it is possible to test the correctness of the implemented algorithm and debug it using traditional tools. The user can activate the parallel execution just by loading the R/parallel package before performing a calculation (condition tested with `"rparallel" %in% names(getLoadedDLLs())`). This design decision is also supported by the fact that, as the user program is not functionally dependent on R/parallel, it can always be shared with other bioinformaticians without requiring them to install the package or modify a single line of code.

Finally, we must observe that using a high level of abstraction restricted to a few control structures, e.g. parallel loops, is less flexible and probably the produced programs are less efficient than using low-level languages and restructuring our programs with a specific target machine. However, given our running environment considers users with reduced knowledge and available time, the chosen option, besides of being more portable, enables the quick transformation of programs with minimal user contribution providing good performance improvements that otherwise would be probably never achieved.

3.2.2. Transforming the Original Program

R is an interpreted language that does not include mechanisms for preprocessing their programs or scripts. As it was exposed previously this prevents the utilization of techniques used by other solutions available for compiled languages like OpenMP. However it also provides other mechanisms, not present as straightforward in non-interpreted languages, from which we can take advantage. In this section we describe the method proposed to create a parallel version of sequential programs with parallel loops coded in R.

The basic idea is to let the R program run as usual until we reach a function from where we want to invoke the parallel version of a section of code, a parallel loop in our case, that the user has indicated to be parallelized. From there we can prepare and check any precondition to launch the parallel version and once everything is ready, proceed with the concurrent execution. Once finished, any postcondition or finalization task can be performed and the flow of execution is returned to the same point from where it was diverted. Figure 22 illustrates this process.

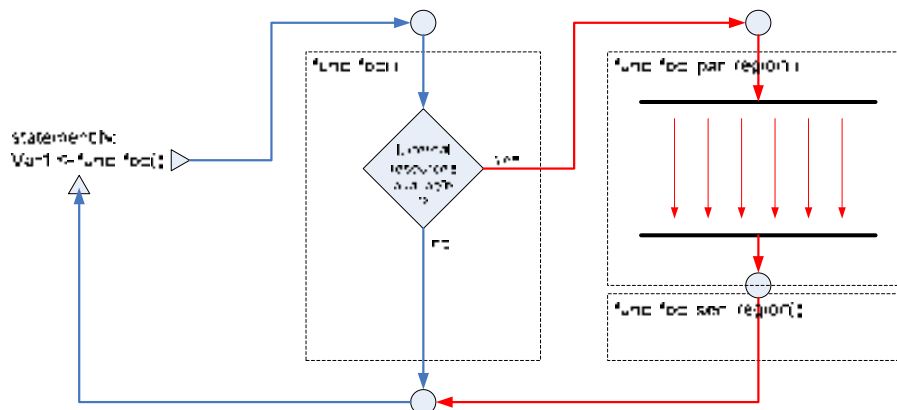


Figure 22: Diverting the flow of execution

The main requirement before launching the parallel version is to retrieve all the available information at the point from where the execution flow was diverted. This includes the state of the current execution, i.e. all the accessible variables and their values, as well as the code or statements of the loop body that must be parallelized. Here is where R, being an interpreted language, provides different mechanisms that we can benefit for our proposal.

Interpreters normally do not use the complete view of the program until it is needed. They parse one statement after the other and they retrieve the variables only when they are required to evaluate an expression. That implies that certain flexibility retrieving variables must exist.

R represents any manipulable entity with an object. There exist several types of objects which include regular variables but also functions and sets of bindings called frames. It is of special interest the scoping rules used in R which define how objects are located along the search path [RD08]. Taking advantage of this characteristic, it is possible to retrieve the value of any accessible object, manipulate them and create new objects at runtime. Working with R we have two important advantages for our goals. First, at any time, we can retrieve all the accessible objects and take a snapshot with the state of the ongoing computation. With this snapshot, any object can be serialized and transferred to a different computer with a different R session, where it can be restored and later continue with a different computation restarting from the same point. And second, it is possible to retrieve the source code of any function, modify it or create new ones at runtime. This characteristic will allow us to extract the source code of the iterations, i.e. the statements that contain the body of the loop, and create *templates* which can later be used as a base for the adapted scripts generated to run the parallel regions on (remote) worker nodes.

At this point we have all the data and the code templates and next step is to define working units that will be assigned to different processors to be executed concurrently. Independently of the type of processing units available and the method used to access them, we can use a master-worker model. The idea is to let the master component to perform all the preparation steps, including the set up of working units that will be dispatched to the workers for their execution. Once they have finalized, it will collect back the results and it will perform any additional operation before returning the flow of execution. Next section describes how the master-worker model is implemented to access additional processing units.

3.2.3. Accessing Additional Processing Units

In symmetric multiprocessing systems the most commonly used technology is multithreading. Programming with threads can be quite complex for inexperienced programmers although since we are considering a semi-automatic parallelization method this complexity will be hidden from the user and the technical details will be handled by the parallelizing solution. However, the design of a solution to provide parallel computing capabilities in single thread legacy applications like the R language interpreter is constrained by several problems [VS10a], some of them already introduced in this dissertation. Next we review the problems identified and the proposed solutions.

Language interpreters are a good example of programs that have evolve considerably over time. Their implementations can be grouped into two different approaches observing how they handle the global variables shared between different concurrent threads: share all or share nothing. The share all approach has the advantage that any variable is directly accessible at any time. However, since the access has to be

controlled continuously with global locks, its performance falls down with an increasing number of parallel threads. The second approach, in contrast, shares nothing unless explicitly defined. This imposes more work for the programmer but results in better scalability. This approach has been used by many language interpreters like python, erlang or perl. In fact, the perl implementation of threads switched from the share all to the share nothing approach in version 5.8.0 [Pr102] to overcome the poor performance of its earlier implementations. With this second implementation the scalability is dramatically increased although the sequential access to shared variables is still a bottleneck.

The interpreter implementations, besides of choosing a share nothing approach, can be classified into two additional groups. One group implements dedicated user level threads to manage the shared resources, also known as green threads, while the other delegates its control to native system calls at kernel level, known as native threads.

The first option has the advantage, on single processor computers, that since the controlling thread has specific knowledge about their own family threads, its expected performance should be greater than if managed by general purpose kernels, which have no knowledge about the future requirements of the threads being scheduled. However, the common disadvantage, since all user-level threads belong to the same process and they share the processor quantum of scheduled time (i.e. cooperative timeslicing scheduling) is that only one thread is scheduled at a time to a processing unit. With a high number of running threads, the controlling thread turns into the busiest one, blocking the others to get access not only to the shared variables but also to their share of processor time.

This scalability problem appeared for example in early versions of the java virtual machine or in the ruby interpreter language. Using native threads have been adopted by other programs like the python interpreter or later versions of the java virtual machine to solve this limitation.

Taking into account the evolution and experience of those general purpose interpreters seems logical that when performance on multiprocessor systems matters, and restricted to the situations depicted in the introduction, using a share nothing approach based on internal operating system mechanisms is the recommended option.

Being the R language interpreter a considerably large non-thread-safe application it is not advised to initiate a restructuration that will require an extensive revision of all the global variables used all over its source code and a later validation of the changes introduced to ensure its initial quality levels. Moreover, the R language interpreter, like many modern languages, is evolving continuously, and every year a few updates are released. That will require a continuous tracking of the changes introduced in new versions that clearly discourages any direct modification.

At the other hand, choosing a third party tool, if available and compatible, technically and legally, for our application, has to be carefully done if we expect this introduced dependency to coexist safely for the coming years. Therefore, as long as multithreading within the same working process is not directly a feasible option, a classical alternative, multiprocessing, seems a right choice.

The requirement for that option, multiprocessing, is to find a way to create multiple processes with selected code and manage its execution. As we exposed earlier, libraries like MPI provide helpful functions than can assist with the task of spawning and communicating processes but they also have two major inconvenients for our specific needs: first, they require the installation and configuration of additional system software, what turns to be too difficult and scary for non-technical users, and second, the available wrappers existing for the case of R does not provide (yet) an standard and stable programming interface over MPI versions (e.g. Rmpi [Yu02] does not have a seamless integration of different MPI implementations like LAM [BD+94] and OpenMPI [GF+04]).

Finally, taking all the arguments into consideration, the design chosen is depicted in Figure 23. The basic idea is to identify, within the legacy application the sections of code that can independently run in parallel. These sections can be replicated in several independent processes so we are sure we will avoid race conditions when accessing its local copy of the global variables. In order to prepare these processes with different input data, coordinate its execution and collect back the partial results we need a central piece of software. This additional module, as long as it is completely new and shares nothing with the original application can be implemented using multithreading.

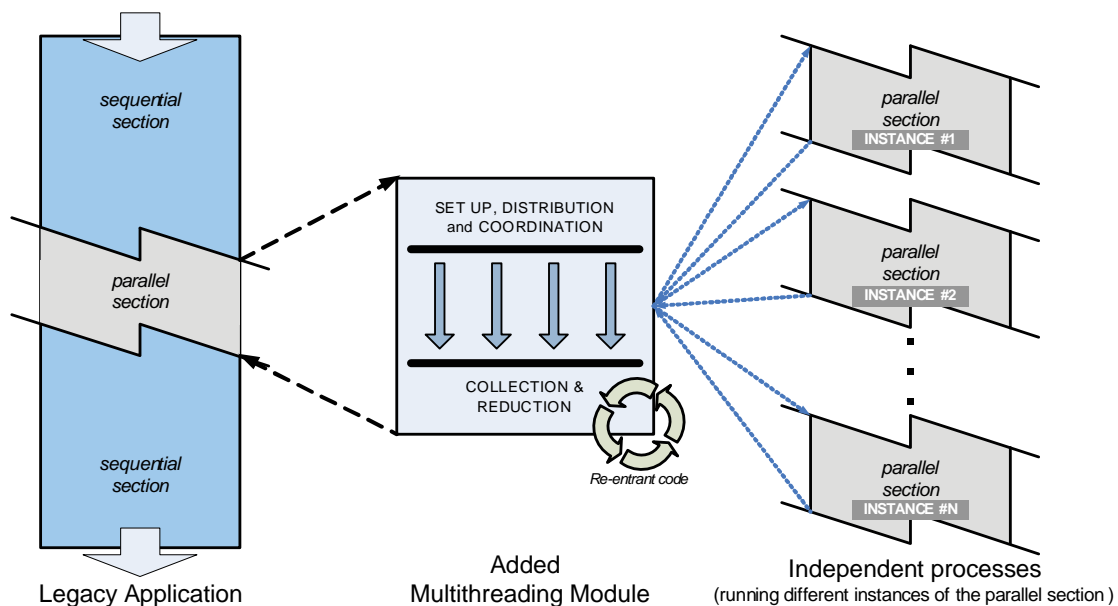


Figure 23: General design strategy for parallelizing a non-thread-safe legacy application

These threads will be used to manage independently the creation and communication of the processes with the module. The result is a master-worker model, suitable for embarrassingly parallel problems, where the central module acts as a master coordinator and a set of working processes, running concurrently over different processing units or cores, perform the calculations that previously were done sequentially within the legacy application.

The master component runs within the main R instance and it is implemented using R scripts and C++ objects, taking advantage of both programming worlds. Workers also make use of R and C++, but they run in separate R instances in independent processes. Combining low-level operating system calls in C++ to manage processes, threads and inter-process communications (IPC) with the intrinsic features of R, like the capability of retrieving or creating functions at runtime (a feature known as “computing on the language” [RD08]), it has been possible to build a generic solution able to automatically transform a sequential loop and parallelize its execution. The general mechanism is depicted in Figure 24.

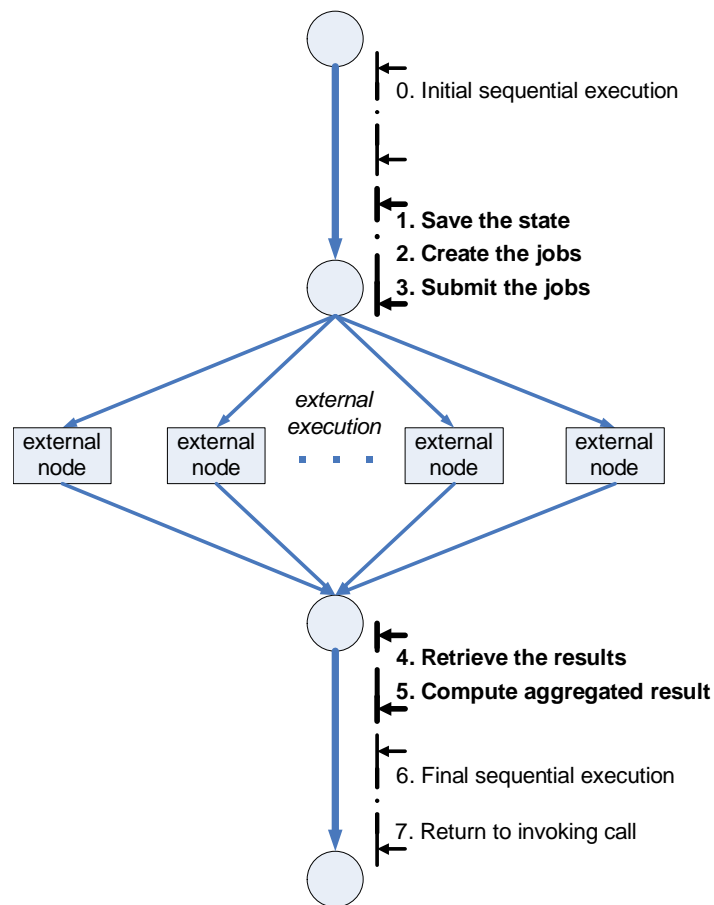


Figure 24: General master-worker model implemented in R/parallel

Once a parallel section is reached and the parallel execution of a loop is invoked from the function `runParallel()`, a new object with a controlling master module is created. There the first step performed is to retrieve the current value of all the accessible variables of the ongoing execution, including the source code of the body of the parallel *for* loop, and the state of the ongoing computation (i.e. all the accessible variables) is stored in a file. With this information, and knowing the parallel section of code to be run, the independent processes are set up based on script skeletons and spawned using bootstrap files (with the generated scripts) and system calls (i.e. *fork_like* functions). Using standard system calls, although less straightforward than using already done wrapper libraries ensures the autonomy, and therefore the long term maintainability of the application.

Next, the workers, once launched and after loading the files that contains the state of the computation, have almost all the information and only have to contact back the master to request their portion of work, in our case, the indexes of the iterations to be performed. Since we do not know beforehand which data or subsets of data will be required by each worker, all the variables are loaded by the workers. This will ensure the minimization of interprocess communication and synchronization.

The master, accordingly to the number of available workers will define and assign a task with a variable number of iterations. The number of iterations (also called the chunk size) is determined by the task scheduler implemented. In the first version the policy defined is just to dispatch chunks of size $K = N/P$, being N the number of iterations and P the number of workers. Next, each worker will perform its assigned task and once finished it will contact back to the master. The communication is carried out and coordinated using standard IPC system calls and objects (i.e. mutex variables and pipes). Once the workers have processed all the iterations and have returned back their partials results, the master has to calculate the final result. To do it, knowing the jobs assigned to each worker and the reduction operations indicated by the programmer the master module is able to compute its aggregated results (i.e. reduces the partial values) to obtain the single final values. Finally, from the same `runParallel()` function, the R environment of the invoking function is updated with the new values of the reduced variables, and the execution continues from the next sequential section without further changes. With this strategy, we can effectively run concurrently any section of R scripts by raising several instances of R conveniently prepared to communicate with the central module.

3.3. Providing Support for Distributed Computing

In this section the extension of our prototype to enable the utilization of distributed computing resources is described. In the previous section we describe our prototype design and implementation details for multiprocessor computers, which at this point supports the parallelization of loops without data dependencies in multicore computers. The next logical step to increase the performance of a parallel system is to extend its capacities by aggregating the computational power of networked computers and adapt the software to support distributed execution of parallel loops using non-dedicated computers volunteered by their owners or users.

There are several types of computing resources that can be used to perform distributed computations and that may not be used because it does not exist the convenient mechanisms to access them. Figure 25 illustrates several options that can be considered for aggregating additional computer resources. Particularly for our running environment we consider these additional sources of computation within the context of Desktop Grid solutions, which general characteristics have been reviewed in section 2.6.

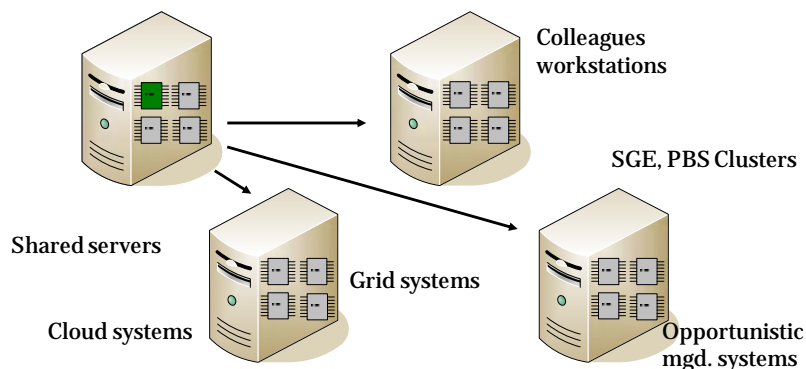


Figure 25: Several distributed computing resources can be used to increased the processing capacities of a parallel system

Given the heterogeneity of these environments, well-known scheduling schemes, suitable for parallel loop scheduling should also be considered. This will allow us to identify the right scheduling methods required for efficiently make use of the additional resources and consequently increase the performance of our parallel applications. They are reviewed in Chapter 4 Parallel Loop Scheduling.

In order to achieve such extension additional developments must be done. First me must develop suitable methods in R/parallel to cross the boundaries of a single

computer and operating system and reach additional processing units. Besides of using internetworking methods, that objective implies adapting the templates and launch scripts used to reach each different type of working node. Next, resource allocation methods are required to find the available computers that will be used for a calculation, at runtime.

3.3.1. Adding Support for Additional Remote Workers

The utilization of remote workers requires several additions to the original structure of R/parallel. The idea is to enable the access to remote nodes, *volunteered or enabled* by their owners and users, in such a way that they can be contacted at any time to request their collaboration with an ongoing calculation.

Besides of requiring the utilization of a communication protocol, standard TCP/IP sockets in our case, a listener component is required in the volunteer side to enable such contact. Internally, since we are using C++ objects, the methods for sending messages between components use either sockets or pipes, depending on whether the communication is established between the remote or the local components. The internal developed interfaces in general classes encapsulate the new functionalities, allowing the internal software architecture of R/parallel to remain without changes.

Another change implies adapting the templates and script skeletons used to raise the workers. Since we intend to use different types of computing resources (e.g. standalone computers or clusters), there are several available methods to launch processes or execute jobs for each case [VS10b, VS10c]. Besides, the local storage cannot be used to exchange information between the master and the workers. Due to that the templates internally used must be extended to: first, configure the right method to launch the workers, for example, the `qsub` sentence with the right parameters in a batch scheduler, and second, provide the required information for the worker to contact back the master in order to download the state of the ongoing calculation, the tasks including their assigned iterations and return the results obtained after processing their assigned tasks.

3.3.2. Allocating Additional Working Nodes

The new version, besides of providing new scheduling capacities, takes care of the coordination and synchronization of remote workers incorporating distributed nodes. The only requirement left for the user (and a third colleague or collaborating peer) is to perform the action of volunteering his or her computer to another user (or himself in a different computer), and in turn, the other colleague, besides of acting in a *quid*

pro quo basis and also volunteering his or her computer, only has to add this colleague's machine to the list of known collaborating remote nodes. Both actions are performed from R with simple function calls provided with R/parallel. An example is provided in Figure 26.

```
# to install the volunteer listener and grant access to a peer
> install.volunteer()
> grant.access( node="192.168.0.4" )

# to add the new collaborating peer with
#   IP 192.168.0.5 to the list
> add.volunteer("192.168.0.5")
```

Figure 26: Code snippet to add volunteer nodes

Internally, the function `install.volunteer()` will create a `cron_like` process that will monitor that an R session with a listener is running continuously. Next, the `grant.access()` function modifies the internal access policy of a volunteer to grant access to a computer. Finally, the function `add.volunteer()` adds a new entry to the list of known volunteers. With this information, the next time a computation is performed, R/parallel, before splitting the loop and creating the tasks, will request to all the nodes included in this list to collaborate in a calculation. The process is illustrated in Figure 27.

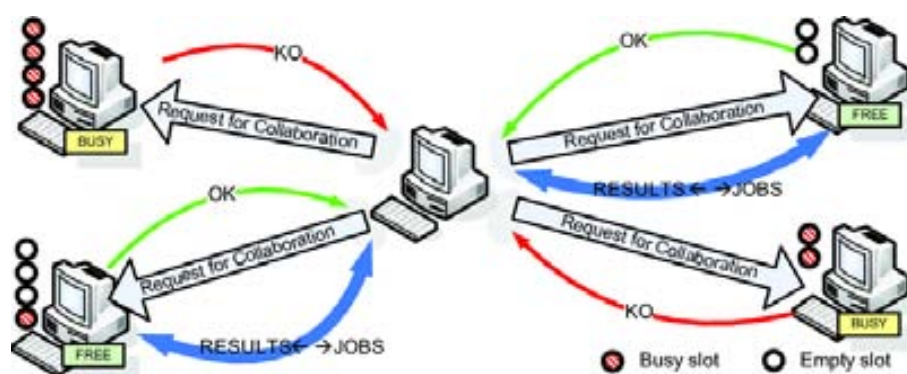


Figure 27: Request For Collaboration (RFC) procedure to retrieve additional working nodes

If the requirements sent by the master (e.g. additional loaded R libraries) are fulfilled by the remote node and there is a free slot, the remote node spawns the worker processes with *ad hoc* bootstrap initialization files that provide the workers with the

information required to join the calculation. The internal components of R/parallel involved in this procedure and the subsequent distributed computation are illustrated in Figure 28.

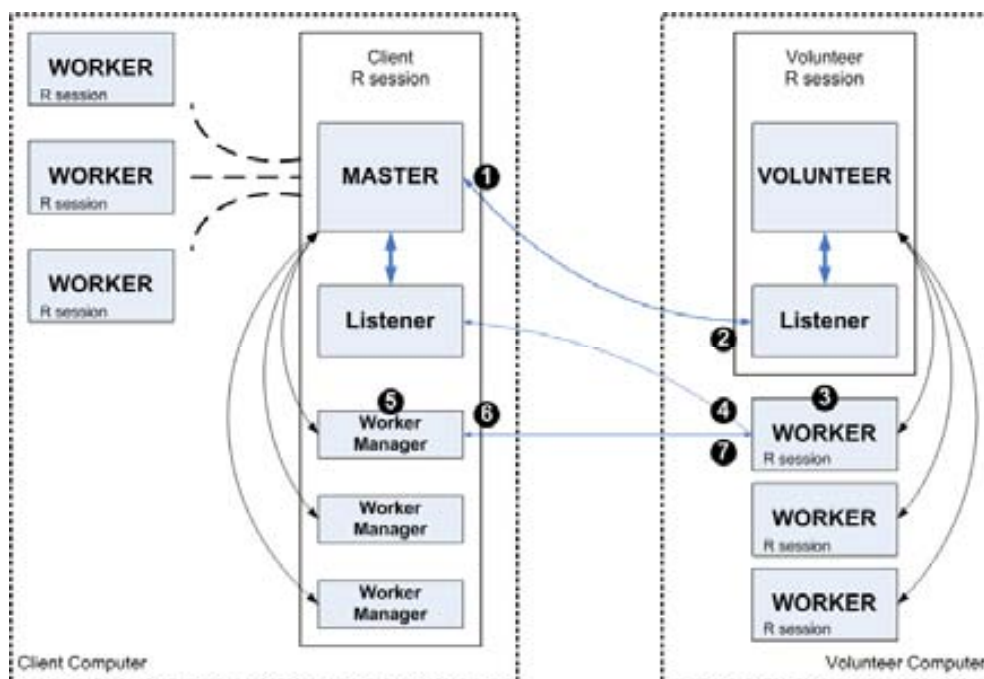


Figure 28: Internal components of R/parallel involved in a distributed computation

Once a new calculation begins, the master component tries to reach each remote node in its list and sends a request for collaboration (RFC) to the listeners previously set up with `install.volunteer()` (step-1). Within this step, information regarding the current computation is exchanged (e.g file space required for the serialized environment variables). If the requirements are fulfilled by the remote node, a positive answer is returned to the master including the number of workers going to be provided (step-2). In the next step (3), the remote node spawns as many worker processes as indicated to the master with ad-hoc bootstrap initialization files that provide to the workers with the information required to perform the calculation, including the network address of the requesting client. As it can be seen in the Figure 28, the master component is also able to spawn local workers. Once the remote workers are initialized they will connect back to the master listener to get involved in the computation (step-4). At this time the master component will create a dedicated thread for each worker (step-5) to handle the exchange of tasks and results between the master and the workers (steps 6 and 7). Besides of this mechanism and the new scheduling schemes explained next, there are no significant design differences between the local only version of R/parallel and this extension.

3.3.3. Adapting the Scheduling Schemes

Since the quantification of the computing power with a positive constant is still an open problem [DL06], performance indicators based on intrinsic characteristics of computers, such as processor frequency, can not completely be trusted. Performance results based in predefined metrics which are true for a given application and computer can change if any of these elements is also changed. Moreover, when running on non-dedicated environments, there are multiple factors that have a negative effect over the raw computing power. These factors range from network congestion to computer overloading. Due to that, it is quite feasible that at some times, relatively small nodes perform faster than more capable ones. For this reason, an given we are focused in dynamic environments like the one depicted in Figure 29, we have chosen the task turnaround completion time as the only performance indicator we can really trust.

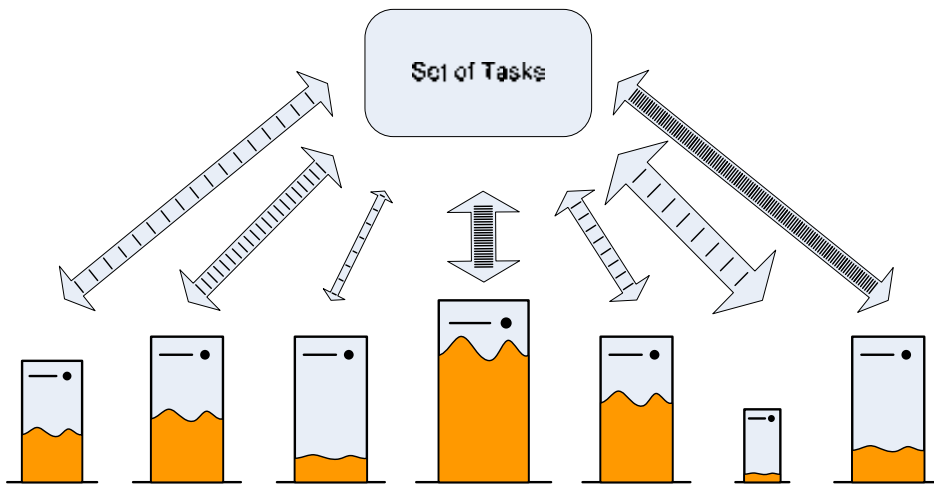


Figure 29: Changes in the relative capacity of heterogeneous computers can be caused by several factors, like variable loads and network congestion

The implementation of the scheduling schemes described in Chapter 4 follows the original definitions provided by their authors. The introduced modifications are limited to the utilization of information only available in the client/master side, specifically as described, the elapsed time since each task is scheduled to a worker with a chunk of K iterations until its partial results are returned back, i.e. the task turnaround completion time. The utilization of the task turnaround time is therefore included as a performance indicator within the scheduling schemes that require such kind of information, like for example the AWF scheduler that requires a performance

indicator to calculate the workers relative weights. The schemes implemented, and later on tested in the experimental section are: STATIC, PSS, FSS, WF, AWF and AF (for details see Chapter 4 Parallel Loop Scheduling).

3.4. Concluding Remarks

R is the preferred tool for statistical analysis of many bioinformaticians due in part to the increasing number of freely available analytical methods. Such methods can be quickly reused and adapted to each particular experiment. However, in experiments where large amounts of data are generated, for example using high-throughput screening devices, the processing time required to analyze data is often quite long. A solution to reduce the processing time is the utilization of parallel computing technologies. Because R does not support natively parallel computations, several tools have been developed to enable such technologies. However, these tools require multiple modifications to the way R programs are usually written or run. Although these tools can finally speed up the calculations, the time, skills and additional resources required to use them are an obstacle for most bioinformaticians.

Our proposal to overtake these problems is materialized in an R add-on package called R/parallel which can be loaded dynamically into the R language interpreter and allows the parallel execution of for loops without data dependencies using the strategy explained previously. The design principles have been proved correct regarding the supporting technologies chosen. The R package has remained completely independent and functional across several version updates of R since R/parallel was released for the first time [VS08].

We have show that it is feasible, even for an interpreted language like R, to integrate different sources of non-dedicated computing resources in a transparent way to the end user. In our case we have integrated the resources from two different types of sources: non-dedicated volunteered computers and compute nodes from a compute cluster managed by the SGE job scheduler. However, additional types of resources which are accessed with different types of frameworks can also be integrated within the same platform by adapting the required templates and access methods.

R/parallel, as shown, can save time to bioinformaticians in their daily tasks of analyzing experimental data. It effectively removes the most common obstacles encountered by bioinformaticians approaching parallel computing in R, like complex programming models or external dependencies on hard-to-maintain software frameworks. R/parallel is an easy-to-use R package which allows any programmer to parallelize their loops in a matter of minutes. The results in Chapter 5 will demonstrate that R/parallel efficiently increases the performance of R when running

parallel computations in current desktop multicore processor computers as well as using non-dedicated distributed environments. As a consequence, bioinformaticians are able to approach reducing the processing time of a growing number of analytical methods based in parallel loops in some cases even by N-fold, N being the number of aggregated processing units.

Chapter 4

Parallel Loop Scheduling

4.1. Introduction

Loops exhibit most of the parallelism present in numerical programs. Therefore, distributing the workload of a loop evenly among the processors is a critical factor in the efficient execution of this kind of parallel programs. A loop scheduling strategy assigns iterations to the processors of a machine in such a way that all of them finish their workload at more or less the same time. A simple strategy is static scheduling, which determines the assignment of iterations to processors at compile-time. But in many situations, the workload of the iterations is unpredictable at compile-time. The main source of inefficiency, specially when using heterogeneous systems like volunteer-based ones comes from load imbalance.

Besides of the static differences among distributed nodes, e.g. different processor clocks, other dynamic parameters vary during a computation, e.g. other programs running in shared computers, changing the state and performance of nodes and consequently modifying the expected results for the scheduled workload. To prevent this and reduce the negative effects of changes dynamic scheduling schemes are used. In this chapter we review both, static and dynamic scheduling strategies for parallel loops. First, well-known schemes are described, next several examples of current contributions are provided, and finally a new scheduling scheme is proposed.

Scheduling parallel loops have been extensively studied in the literature [BV02, BV+03, HS+92, KW85, PK87]. The common approach is to set up tasks with subsets of N iterations in chunks of different sizes K which are dispatched to P processing units or nodes. The objective is to distribute conveniently the workload, i.e. all the iterations of a given loop, among all the processors to minimize the overall execution time. The method used to calculate the chunk size assigned to each processing unit as well as other parameters is defined by the scheduling policies implemented in a scheduling scheme.

In order to obtain a balanced distribution of the workload, several scheduling methods can be selected. Parallel loop scheduling schemes found in the literature can be classified into two classes accordingly to the time when the required information to define the distribution is obtained and used: nonadaptive scheduling schemes and adaptive scheduling schemes.

4.2. Nonadaptive Scheduling Schemes

Nonadaptive scheduling schemes define chunk sizes based on the available information before running the loop, usually at compile time. These chunk sizes can be fixed for the whole computation or updated at runtime based on predefined rules. Next we describe the most representative scheduling schemes that define fixed chunk sizes:

- **Static scheduling (STATIC).** This scheme is suitable for homogeneous environments with fixed time iterations, i.e. uniform loops. It generates, being N and P respectively the number of iterations of the loop and the number of processing units, chunks of size $K = \frac{N}{P}$. It has low overhead since all the iterations are dispatched in equal single chunks to each processor. This scheduling method belongs to the subclass of *chunk scheduling schemes* since it assign subsets of consecutive iterations to each different processor, and is the scheme most frequently implemented in static environments given its simplicity and effective results.
- **Pure self-scheduling (PSS).** In this scheme chunks of just $K = 1$ are generated. PSS, in contrast to STATIC, belongs to the subclass of *cyclic scheduling schemes* since it distributes all the iterations cyclically among the processors, one at a time. This method achieves load balancing but also a likely high overhead depending on the communication required among processors.

- **Chunk self-scheduling (CSS)**. This variant allocates a constant set of K iterations, but in this case the constant must be specified by the programmer. A high value of K is likely to cause load imbalance and a small value can eventually produce too much overhead. It can be observed that CSS can be set with the same values to match PSS and STATIC schemes.
- **Fixed size chunking [KW85] (FSC)**, proposed by Kruskal and Weiss is based in the following formula:

$$K_{FSC} = \left(\frac{\sqrt{2}Nh}{\sigma P \sqrt{\log P}} \right)^{\frac{2}{3}}$$

It provides an optimal constant value for the chunk size that minimizes the execution time on homogeneous and equally loaded processors once all its parameters have been defined, which includes the standard deviation σ of the iteration time and the scheduling overhead h . Defining these parameters can be complicated for some programmers.

These schemes provide fixed chunk sizes suitable for homogeneous environments but in many cases its results are not satisfactory for heterogeneous environments. Next schemes provide solutions for heterogeneous processing units.

- **Guided self-scheduling [PK87] (GSS)**. This scheduling scheme and several others based on the same idea can dynamically change the value of K by generating decreasing size chunks. Their approach is to dispatch large chunks at the beginning of the calculation to reduce the overhead, as it is achieved with STATIC, and gradually reduce the chunk size to align the finalization time of the involved processors, as happens with PSS. Its equation for calculating K , based on the remaining iterations to be scheduled, is as follows:

$$K_{GSS} = \left\lceil \frac{\text{remaining}}{P} \right\rceil$$

Therefore, the first processor at the first assignment obtains the largest chunk, the second processor, a smaller one, and so on, until there are no more remaining iterations. For example, for $N = 100$ and $P = 4$, it will generate the following chunk size sequence: $K_0 = 25$, $K_1 = 19$, $K_2 = 14$, $K_3 = 14$, $K_4 = 11$, $K_5 = 8$, $K_6 = 6$, $K_7 = 5$, $K_8 = 3$, $K_9 = 3$, $K_{10} = 2$, $K_{11} = 1$, $K_{12} = 1$, $K_{13} = 1$, $K_{14} = 1$.

It is very important to align the finalization of the processor at the end of the

calculation to avoid inefficient utilization of the resources due to possible delays of the whole processing in the presence of too slow or overloaded processors, as this scheme provides.

- **Trapezoid self-scheduling [TN93] (TSS).** This scheme is a variation where chunk sizes decrease linearly, in contrast to the geometric decrease of chunk sizes in GSS. Based on the parameters f and l specified by the programmer, $TSS(f, l)$ distributes all the iterations, starting with a chunk of size f , and decreasing its size linearly until the last chunk of size l . The authors proposed $TSS(N/2p, 1)$ as a general selection. In this case, chunk sizes are decreasing in steps of $N/8p^2$, starting from $N/2p$ and finalizing in 1. The strategy of this approach is double and defined by the values of f and l . With the f value it is possible to modify the initial value of the chunks defined in GSS. This is important in scenarios where heterogeneous processors are used. It is possible to find slower nodes that just with the initial chunk assignment spend more time than rest of the nodes processing all the other iterations. Adjusting the value of f this problem can be avoided and let the slower nodes contribute in the right way. Regarding the parameter l it is possible to adjust the size of the last chunks. This is desirable when the overhead of processing one or a few iterations (communication or synchronization overhead) is higher than the time required to process a small chunk, even if that involves a reduced misalignment of the processors at the end. Therefore, adjusting l it is possible to control the granularity of the last tasks and avoid a potential inefficiency due to an unfavorable processor-communication ratio.
- **Factoring self-scheduling [HS+92] (FSS).** This scheme proceeds in phases. During each phase, only a subset of the remaining iterations, i.e. a batch B , is divided equally among the available processors. The batch size is a fixed ratio of the unscheduled iterations. The ratio depends on the mean and standard deviation of the iteration execution times. When these statistics are unknown, the ratio 0.5 has been experimentally proved to provide reasonably good results. In this case, the batch and chunk size are calculated as follows:

$$B_{FSS} = 0.5 * remaining$$

$$K_{FSS} = \left\lceil \frac{B}{P} \right\rceil$$

Next phase starts once all the chunks in the current batch have been scheduled.

- **Weighted factoring** [HS+96] (WF). This scheme incorporates, when available before the execution of the loop, information about the processing speed of the processing units or working nodes. In this method, the chunk size assigned to each processor i is readjusted accordingly to its relative speed, defined as a weight w_i which is used as follows:

$$K_{WF_i} = w_i * K_{FSS}$$

Using this method, faster processors obtain bigger chunks than slower ones. This scheduling scheme is suitable for heterogeneous computers in static environments since it adjusts the assigned portion of the workload to each working node based on its processing capacity. However, it assumes that the relative speeds are constant throughout the execution of the loop, what cannot always be assured in some cases due to several factors, for example operating systems interference.

The scheduling schemes presented, nonadaptive, expect uniform loops with fixed iteration bounds and static processing units in order to provide correct schedules. However, when using non-dedicated environments, like happens in other changing environments, the conditions that led to a given schedule may change and hence the assignments done may not be well balanced any more. Changes may be originated within the worker nodes, e.g. parallelization overhead or unbalanced loops, and its external environment, e.g. network congestion. In order to react against these changes that degrade the performance obtained with nonadaptive methods, adaptive scheduling schemes have been proposed.

4.3. Adaptive Scheduling Schemes

Adaptive scheduling schemes described in literature mainly evolve from the factoring and the weighted factoring schemes. Next, two representative schemes are described.

- **Adaptive weighted factoring** [BV+03] (AWF) was originally developed to tackle with applications where changes during loop execution not only occur due to the processing speeds of the involved working nodes but also to the parallel loops themselves, which present different loads between iterations. This can be the case of semi-uniform or non-uniform parallel loops. AWF, when determining the chunk sizes, attempted to incorporate both sources of variability, external due to the running environment changes and internal due to the characteristics of the workload. Initially, chunk sizes are determined as in WF, but at the end of each time step, the processor weights w_i are adjusted

based on the information collected during the current and previous steps. For the first time step, $w_i = 1$ is defined. These timing data are used to update w_i for the next chunk calculation. This method provides very good results in heterogeneous environments where the running conditions evolve during a calculation. However depending on the implementation and the running environment, for example when the number of available processors varies during the calculation (i.e. not the quality of the processors but its quantity), the results obtained can be improved.

- **Adaptive factoring (AF)** [BV02] modifies the FSS method by using an estimation of the mean and standard deviation of the iteration execution times. They are calculated dynamically during the execution of the loop based on the results of already processed chunks. First values are obtained from the execution times of chunks from an initial batch of arbitrary size. Given R (do not mistake parameter R with the R language), μ_i and σ_i , representing respectively the remaining tasks, the mean and the standard deviation of execution time in a processor i , the rest of the chunk sizes are calculated as follows:

$$D_i = \sum_{i=1}^P \frac{\sigma_i^2}{\mu_i}, \quad T_i = \left(\sum_{i=1}^P \frac{1}{\mu_i} \right)^{-1}$$

$$K_{AF_i} = \frac{D_i + 2T_i R - \sqrt{D_i^2 + 4D_i T_i R}}{2\mu_i}$$

This method provides good results for large executions, once the statistic values of the mean and standard deviation have been stabilized. For some irregular executions, when the statistical values take so long to properly represent the behavior of the application, the results obtained in our experiments show that in some cases they can even be worse than those obtained with non-adaptive scheduling schemes.

4.4. Extended Contributions for Parallel Loop Scheduling

The scheduling schemes presented previously provide mechanisms to conveniently distribute loop iterations under different assumptions and prerequisites of the underlying parallel systems. Although they already provide good solutions for most common scheduling problems, new user requirements and technological evolutions continuously provide new goals and environments where new conditions and parameters are introduced, generating new opportunities where further improvements can be achieved. In this section we review some examples of other proposals that

extend previous well-known schemes or propose new approaches to deal with new objectives and problems.

4.4.1. Extensions of Previous Schemes

There exist a large number of contributions that have evolved from previous well-known schemes, either by combining different schemes or by introducing new concepts that eventually improve previous performance results. Next we provide some remarkable examples.

Chronopoulos et al. [CA+01] proposed an extension of previous work on multiprocessor systems to support heterogeneous distributed systems. First, they propose a combination of previous schemes, FSS and TSS, to combine the strengths of both schemes, called **Trapezoid Factoring Self-Scheduling (TFSS)**, which is suitable for distributed systems. They use the idea of batches or stages introduced by FSS, meaning that the iterations are scheduled in groups of P equally sized chunks. While in FSS it was proposed to schedule half of the iterations at each batch, here the size of the next batch is the sum of the next P chunks that would have been computed by the TSS scheme, i.e. $B_{TFSS_j} = \sum_{i=k}^{i=k+P} K_{TSS_i}$. The batch is then equally divided among the P processors, as in the FSS scheme. Next, in order to support distributed environments, they implement TFSS and other previous schemes using a master-worker paradigm and include the *virtual computing power* of the workers to support heterogeneous nodes, as it was done previously by the same authors in a previous proposal called Distributed Trapezoid Self-Scheduling (DTSS). The main difference with respect to TSS, introduced to take into account the speeds of the processors to adapt to the actual load of the distributed system, is the exchange of the number of processors P with the available computing power obtained using the ratio of virtual computing power and number of processes in the run-queue of each worker. Their implementations with MPI show its applicability in environments with dedicated and non-dedicated nodes using mixtures of fast and slow nodes.

Díaz et al. [DR+06] provided a general formulation of the self-scheduling problem that derived in the proposal of a new self-scheduling scheme, **Quadratic Self-Scheduling (QSS)**. The basis of their proposal follows. Let $K(t)$ be the chunk function giving the chunk size for the t^{th} task, with t defined within the interval $[0, T]$. Two conditions must be fulfilled by any form of the function $K(t)$. First, the exigency of a decreasing chunk size requires that $\frac{dK(t)}{dt} < 0$, and second, given N the total number of iterations, $N = \int_0^T K(t) dt$. Considering $K(t)$ an unspecified function of t such that $K(t) = f(t)$, $K(t)$ can be expressed as an expansion of $f(t)$ with an approximation of Taylor series, that in compact form can be formulated as: $K(t) =$

$a + bt + ct^2 + \dots$. Limiting the expansion of the last equation to the constant term we obtain the scheme PSS ($a = 1$) or STATIC ($a = N/P$). Retaining the linear term we can obtain the TSS scheme, with the following formula: $K(t) = K_0 + \frac{(K_T - K_0)}{T}t$, where $K_T = 1$ and $K_0 = N/2P$. A more flexible model is obtained retaining the quadratic term, finally resulting in:

$$K(t) = a + bt + ct^2$$

That equation defines the QSS scheme. To apply QSS the coefficients a, b and c must be defined. The following equations can be used to obtain these values:

$$\begin{aligned} a &= K_0 \\ b &= \frac{4K_{T/2} - K_T - 3K_0}{T} \\ c &= \frac{2K_0 + 2K_T - 4K_{T/2}}{T^2} \\ T &= \frac{6N}{4K_{T/2} + K_T + K_0} \end{aligned}$$

The chunk size for the first, middle and last task (respectively $K_0, K_{T/2}, K_T$) can be defined by the user to tune the number of tasks produced and hence optimize the load balance to overhead ratio by selecting an appropriate value of $K_{T/2}$. Values of $K_{T/2}$ bigger than $\frac{K_0 + K_T}{2}$ reduce the number of tasks and hence the overhead, while smaller ones increases the number of tasks achieving better load balancing and allowing a more accurate distribution of the final tasks.

Gap-aware Self-scheduling (GAS) [KN+05], based in GSS, considers the problem of scheduling parallel tasks of an individual job on a multiprogrammed parallel system. In particular they address the problem of minimizing the maximum completion time of DOALL loops. Several schemes assume a fixed number of processors. However the latter is not valid in context of multiprogramming. This can potentially give rise to “gaps” in processor availability, i.e., a processor may not be continuously available to the same job. For example, in Figure 30, processor P2 is not available for $t \in (750, 850)$.

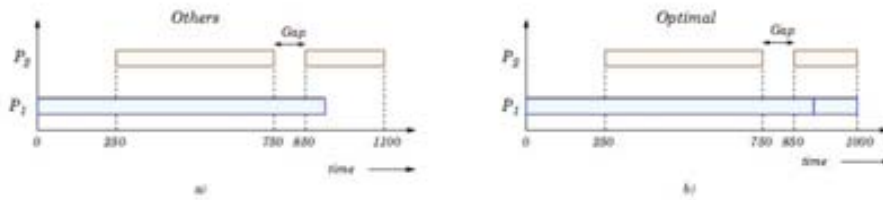


Figure 30: a) Schedule obtained from existing techniques, b) Optimal schedule (source [KN+05])

The effect of varying number of processors and the presence of *gaps* was not explicitly considered previously. They propose GAS to capture the effect of the presence of gaps in processors availability on self-scheduling. At each scheduling step, GAS computes the chunk size based on the number of remaining iterations and gaps in processor availability. In order to capture the effects of gaps a *displacement factor*, is defined for online modulation of the chunk size. The displacement factor is computed based on the differences of finishing time of previous executions. The idea is that at any point in time, the amount of workload assigned to each processor should be chosen such that the remaining workload is “sufficient” to balance the workload evenly, i.e., the difference in finishing times of the processors at the end of the schedule is minimal. In front of the evidence that a processor will not be completely available its iterations are saved for other processors. Using this information, an additional correction factor called *lag*, used to “relax” the exponential decay of chunk size and avoid scheduling too much iterations at early steps, and W_{min} , the minimum chunk size defined by the user, they modify the equation on GSS to calculate chunk sizes to adapt to different processors availability at runtime.

In [KN+06], Kejariwal et al. propose an execution **History-aware Self-Scheduling** scheme (HSS) to handle irregular parallel loops on heterogeneous multiprocessors systems. First, the chunk size is computed based on the variance in workload distribution across the iteration space. At each scheduling step, HSS computes the amount of workload to be allotted to an idle processor based in the remaining iterations and processor speed. Next, it determines a set of iterations that best fits with the above workload, where the number depends on the workload distribution of the remaining iterations. The key characteristic is the dynamic adaptation of the chunk size based on the statistical deviation of the workload estimates of the previously executed iterations from their corresponding actual workloads. It requires to determine the expected workload of each iteration on each processor, what is done offline via loop profiling [Sar89]. They profile the loop with multiple training sets, determining also the execution probability of each basic block in each iteration. The expected execution time or workload of an iteration is the sum of the execution times of all the basic blocks weighted with their respective execution probabilities for that iteration. As in [KN+05], they adapt the GSS chunk size equation, using in this case the workload estimations information and including also the *lag* correction factor and W_{min} , the minimum chunk size or workload per task.

4.4.2. Extensions Applied to Distributed Environments

The application of loop scheduling schemes to new environments like Clusters and Grids has promoted the research for new proposals to adapt and deal with new requirements. Here we provide some examples of recent contributions.

In [DR+09], Díaz et al. described two families of self-scheduling algorithms, evolved from TSS, that can be applied in computational Grids. The first considers an explicit form of the chunks distribution function, QSS, which has been already described in this section. From the second, **Exponential Self-Scheduling** (ESS) and **Root Self-Scheduling** (RSS) are proposed. ESS evolves the propositions of QSS by considering that the slope of the chunk size distribution function $K(t)$ is proportional to the chunk size, so a parameter k representing this proportion and representing the working environment must be defined. A second approach, considering that the slope (negative) is inversely proportional to $K(t)$, provides the second self-scheduling method, RSS. These schemes were tested in a static environment of dedicated computers in an Internet-based Grid. After convenient manual calibration of its parameters its tests reveal that QSS still outperforms ESS (slightly) and RSS, which shows a poor performance.

Adaptive Chunk Self-Scheduling (ACSS) [LJ+08] was proposed to achieve load balancing and reduce the scheduling overhead of DOALL loops, where each iteration is considered a simple task that can be scheduled independently on the Grid. They propose the utilization of multiple thread mechanisms and a master-worker paradigm to overlap communications and processing. By interleaving the transfer of input data and results between the master and worker nodes with the computation at the workers, the synchronization waiting times are reduced and hence the total execution time. They also use a weighting mechanism to adapt the chunk size assigned to each worker to its expected performance, in that case calculated based on results of performance evaluation benchmarks, the number of processes in the running queue, and network bandwidth. By doing so, they evolve the CSS scheme and adapt its utilization to heterogeneous environments like Grids.

Yang et al. [YC03] proposed a scheduling scheme, **α Self-Scheduling Scheme**, suitable for extremely heterogeneous PC clusters where previous schemes like FSS, GSS and TSS could not provide satisfactory results alone. They argue that static and dynamic schedulers could be combined to take advantage of both scheduling methods, the reduced overhead of the first and the load balancing on heterogeneous environments of the second. They propose to partition the problem in two stages. At first stage, partition statically the $\alpha\%$ of the workload according to their performance weighted by CPU clock. At second stage, partition the following $(100 - \alpha)\%$ of workload according to known self-scheduling schemes. The parameter α can be adjusted to different ratios of static-dynamic scheduling. The authors suggest that $\alpha = 80$ results in the best average performance. On later contributions, applying the same idea to static environments of dedicated computers, i.e. institutional Grids, they introduce several changes to its initial propositions. First, they replace the utilization of CPU clock speed as a performance index by other performance benchmarks or tools like HINT [YC+04], HPL [YS+06], Globus MDS and Ganglia [SY+07], and

HPC Performance Analyzer [YC09]. Second, they introduce the parameter SWR (Static Workload Ratio) to estimate the proportion of the workload which can be scheduled statically and hence automating the selection of the parameter α . To obtain SWR five random iterations are chosen and its iterations time computed. SWR is calculated as the ratio of the minimum and maximum execution times obtained of all sampled iterations. This generalized approach lead to a new scheme called **Performance-based Loop Scheduling (PLS)** [SY+07].

Finally, in [BC+09], Banicescu et al. provide a comparative analysis of three dynamic loop scheduling (DLS) methods, FSS, WF and AWF, adapted for scheduling irregular applications in large-scale heterogeneous distributed systems like Grids. They propose metrics for quantifying their robustness with respect to variations of two parameters, load and processor failures. Besides of dealing with fault tolerance, a necessary characteristic for loop scheduling in large-scale systems, they also conclude that hierarquical approaches, based in super-master – master architectures on top of several standard master-worker structures, are more suitable than that of the centralized management approach to ensure the robustness of DLS methods on large-scale heterogeneous distributed systems.

4.4.3. Other Current Loop Scheduling Schemes

In this subsection other approaches to the problem of scheduling parallel loops, although do not have a direct impact in our work, are described to illustrate the great variety of research fronts related with parallel loop scheduling.

4.4.3.1. Scheduling Schemes Considering Dependencies

These scheduling schemes, since they consider dependencies, like in DOACROSS loops, are mainly intended, although not restricted, for multiprocessor computers. Although they are not within the goals of this thesis, we think it is important to provide a short review of the advances produced in this area in order to evaluate a future extension of our work in this direction.

Shirako et al [SZ+09] addressed the problem of chunking parallel loops that may contain synchronization operations such as barrier, signal or wait statements. They present a transformation framework that uses a combination of transformations from past work (e.g. loop strip-mining, interchange, distribution and unswitching) to obtain equivalent set of parallel loops that chunk together statements from multiple iterations while preserving the semantics of the original parallel program. These transformations result in reduced synchronization and scheduling overheads, thereby improving performance and scalability.

Liu et al [LS+09] proposed a method for optimal transformation of loops to maximize the iteration-level parallelism achieved on DOSERIAL and DOACROSS loops on multiprocessors systems. The proposed algorithm solves it optimally by migrating the weights of parallelism-inhibiting dependences on dependence cycles in two phases. In the first phase, they introduce a dependence migration algorithm to find a retiming function for a given dependence graph such that β in the graph is maximized. β represents the maximum inter-iteration dependence distance, i.e. the maximum number of iterations that can be processed in parallel for a given loop. In the second phase, they apply a loop transformation algorithm to generate the optimal code for the given loop based on the retiming function found. Their experimental results show that the proposed methods produce optimal solutions and effectively improve loop parallelism compared to previous work.

Ciorba et al [CR+08] introduced a synchronization mechanism that provides inter-processor communication in distributed systems, thus, enabling traditional self-scheduling algorithms to handle efficiently nested loops with dependencies. They extend and generalize previous work by constructing a general synchronization mechanism S , which inserts SPs (synchronization point operations) in the execution flow so that workers perform the appropriate data exchanges. In addition, they define a weighting mechanism W , aimed at improving the load balancing and thus, the performance of non-adaptive self-scheduling algorithms on non-dedicated heterogeneous systems. An extended paradigm of the standard master-worker was required to enable direct communication between workers to handle dependencies. Communication between workers is performed by direct exchanges, and not through the master, which has a global view of the system's load and decides upon allocating the tasks to each worker. Besides of the tasks, based on accounting information, it provides the communication sets to the involved workers that require the exchange of data. The existence of SPs leads to a wavefront execution. They tested their proposals extending CSS, FSS, GSS and TSS, showing how it was possible to adapt these non-adaptive schemes for heterogeneous environments and nested loops with dependencies.

4.4.3.2. Scheduling Schemes Considering the Underlying Architecture

While other contributions focused on generalizing the characteristics of the involved computing nodes, so it easier to handle heterogeneity, others, with different performance objectives, are focused on their internal characteristics, usually multiprocessor computers. Here we provide two interesting examples.

Kejariwal et al. [KN+09] presented a novel profile-guided compiler technique for *cache-aware* scheduling of iteration spaces of parallel loops on multicore systems. First they proposed a novel profitability analysis model for selecting loops for

multithreaded execution. It is important to establish how to efficiently capture the cache behavior since the cache subsystem is often the main performance bottleneck in multicore systems. Second, they propose a cache-aware technique for scheduling parallel nested loops. Specifically, the proposed technique in [KN+09] captures the effect of the variation in the number of cache misses across the iteration space. Finally, they propose a unified approach to capture the effect of the variation in the number of cache misses and computation – the amount of computation per iteration is measured in terms of the number of retired instructions – across the iteration space. They demonstrate the efficiency of their methods using 4-way multiprocessors.

The work exposed in [DC+08] focused on *energy-oriented* OpenMP static and dynamic parallel loop scheduling. They argue that dynamic voltage/frequency scaling (DVS), and effective low-power technique that explicitly trades off performance for energy savings) cannot obtain the maximum energy savings and parallel loop rescheduling should be combined. They propose three algorithms: first, Energy-Saving Static Scheduling (ESSS), which using DVS scales down the processor with less load to save energy so that the time to finish the task at that processor stays the same that the processor with maximum load. Second, Energy-saving Optimal Static Scheduling (EOSS), which obtains the maximum energy saving through combining loop rescheduling to balance the workload on all the processors and later apply DVS based on the worst execution time. And third, Shut-down Based Dynamic Scheduling (SBDS) which shut downs the processors when are idle, usually once they have finished their tasks and are waiting for the others.

4.5. Improving Parallel Loop Scheduling

4.5.1. Preliminary Considerations

In this section we propose a new scheduling scheme designed for the execution of parallel loops in dynamic environments of non-dedicated distributed computers.

We have discussed that the proliferation of multicore computers in desktop environments is introducing important changes in the way regular PCs are used to run applications. Since increasing the number of available processing units per processor is the current trend to increase computer performance, applications are being adapted with parallel technologies to take advantage of the new performance improvements. We have already provided methods for R in that direction. Additionally, since grid and volunteer systems based on desktop computers are proving themselves as real and powerful alternatives for parallel computing, even in office environments, the aggregated computing power obtained can provide similar performance levels that years ago were restricted to dedicated clusters, and that it is a tendency that over the

coming years will become more evident with the increase of core densities per processor. In order to effectively take advantage of such aggregated computing power in dynamic environments, current and new methods, better suited for these environments, have to be adapted and developed. We have evaluated well-known scheduling schemes suitable for parallel loops identifying their potentials and limitations. Here we describe a new scheduling scheme that tries to overcome the problems found in previous contributions called ATLS [VS10d], which stands for Adaptive Turnaround-based Loop Scheduling. It has been specifically designed for processing parallel loops in dynamic environments made up of volunteered computers, for example desktop computers, where its number and capacities are unknown before performing a calculation and, given the resources are not dedicated, can change in quantity and quality at any time during the computation.

Our proposal, by tracking several performance change ratios at runtime, is able to properly adjust the load distribution using no prior information of the loop features nor the involved processors and their environmental running conditions. The results obtained during the experiments performed to validate ATLS will show that it is possible to improve former contributions of well-known parallel loop scheduling schemes in dynamic environments. The implementation of the scheduler has been done for the R language but, as it is exposed, independently of the R language properties, it can easily be adapted to any other language, platform and parallel loop based application.

4.5.2. Proposal of a New Scheduling Scheme

The reasons that have motivated each design decision are explained in this section, together with the details of our proposal, the scheduling scheme ATLS.

We already argue previously that quantification of the computing power using performance indicators based on intrinsic characteristics of computers, such as processor frequency, can not completely be trusted [DL06]. Due to that the performance ratio used in our proposal, using a black box approach, is based on **the task turnaround completion time**, and is calculated as follows:

$$v = \frac{T}{K}$$

where T represent the time elapsed since the task was scheduled to a worker until it was successfully completed and returned back. The variable K represents the number of iterations assigned to that task, i.e. the chunk size. The average performance ratio, v_{avg} , given n finalizations and the current performance ratio v is updated as follows:

$$v_{avg_n} = \frac{v + v_{avg_{n-1}} * (n - 1)}{n}$$

Also the minimum v_{avg} is kept for each processor for further calculations in the variable v_{min} .

Our proposal to mitigate the negative effects of unpredicted changes in the worker node performance is the utilization of a **system-wide confidence indicator**, CI , based on the changing ratios of the performance index of each participating node. The initial value of the confidence indicator, like any of the following variables except if stated otherwise, is $CI_0 = 0$, since all nodes are unknown at the beginning and therefore untrusted. As the computation evolves, its values are updated as follows:

$$CI_n = CI_{n-1} + w_i * (C_r - CI_{n-1})$$

Here two new variables are introduced. The **relative weight** of the worker node i , w_i , and the **current system changing ratio**, C_r . The idea is to update the previous value of CI with the difference between the previous CI and the current system changing ratio, adjusted with the relative weight of the worker node involved in a given scheduling step. By doing so we adjust the system variable CI only with the proportional contribution of each worker, indicated by its relative weight. The value of w_i and C_r is updated whenever a working node returns successfully a job (ontime) or when its finalization deadline is reached (overdue) as follows:

$$w_i = \frac{v_{avg_i}}{\sum_{j=1}^P v_{avg_j}}, \quad C_r = \frac{\sum_{i=1}^P v_{avg_i} * \alpha_i}{\sum_{i=1}^P v_{avg_i}}$$

The variable α_i represents the **current changing ratio for the processor i** while the variable α_{avg_i} represents its average changing ratio and it is updated at each finalization event, positively when a task is returned ontime or negatively when its estimated finalization time, i.e. its estimated deadline, is reached. The calculation of α_n , at the n^{th} time, is as follows:

$$\alpha_n = \begin{cases} \frac{\alpha_{n-1} + 1}{2}, & \text{if a task is finished} \\ \frac{(j-1) * \alpha_{n-1} + \frac{\alpha_{n-1}}{2}}{j}, & \text{if a deadline is reached} \end{cases}$$

The event when a task is done is worker driven. To estimate the finalization time for a task, a deadline λ_i is defined at the time of dispatching a task to the worker i . In an utopic scenario, knowing the real performance ratio v_{real} , the constant overhead of the system h_{ct} , the overhead per iteration h_{iter} and the future delay possibly introduced

by the environment or variable iterations, also constant ϵ_{ct} and per iteration ϵ_{iter} , the ideal finalization time λ_i' can be calculated as follows:

$$\lambda_i' = (v_{real} * K) + (h_{iter} + \epsilon_{iter}) * K + h_{ct} + \epsilon_{ct}$$

However, except for the value of the chunk size K , for the other variables, given the variable nature of the dynamic environments, it is not possible to make any safe estimation of their values. Our proposal to overtake this problem and calculate the finalization time λ_i is based on the following equation:

$$\lambda_i = (v_{avg} * K) + (\xi_i * K) + 0$$

Given the method we use to calculate the performance ratio, it is clear we are introducing an error by including in this ratio a fraction that corresponds to the system overhead, irregular iterations and other external factors originated in other elements of the environment with no direct relation with the ongoing calculation. Nevertheless, as discussed previously, we don't have any other observation to support a different performance ratio.

However, we can also observe the differences between the estimated deadlines and the real elapsed times. Any time a task finishes, before or after its deadline, the time difference, positive or negative, corresponds either to the system overhead or the external factors, for the current or the previous performance ratio values, and can be recovered and kept aside for further estimations.

The variable ξ_i is defined as the **burden ratio**. With this variable we maintain the maximum ratio of extra time per iteration observed during the ongoing calculations. We assume that this extra time, until a higher value appears, can be used to accommodate the estimations of future completion times and, as a consequence, we avoid to declare a scheduled task as lost before it is really needed. Its values are calculated, based on the finalization time, as follows:

$$\xi_n' = \begin{cases} \frac{T_{n-1} - T_n}{K_{n-1}}, & \text{before } \lambda_n \text{ (ontime)} \\ v_n - v_{min}, & \text{after } \lambda_n \text{ (overdue)} \end{cases}$$

$$\xi_i = \begin{cases} \xi_i', & \text{if } \xi_i' > \xi_i \\ \xi_i, & \text{if } \xi_i' \leq \xi_i \end{cases}$$

Before we can define the equations to obtain the batch and chunk sizes, as it is also used in the AWF method, we still have to describe an additional concept to understand our proposal, the **relative capacity of the system** at a given time n or RC_n . This variable represents the number of iterations that can simultaneously be performed by all the available working nodes within a time-frame so all nodes

finalize at the same time. The basic idea is that, assuming there are enough iterations for all the available nodes, in order to obtain the best efficiency, all the available resources must be used at the same time. Therefore, at least 1 iteration must be assigned to the slowest node. During the time-frame required for the slowest node to process its assigned iteration, the rest of the working nodes are able to perform one or more iterations. RC_n represents therefore the sum of the iterations each worker is able to perform during the time-frame defined by the slowest node.

The batch B and chunk size K , at the n^{th} time, can be defined as follows:

$$B_{ATLS_n} = \left\lfloor \frac{\text{remaining}}{RC_n} * CI_n * \frac{\#workers_{active}}{\#workers_{expected}} \right\rfloor$$

$$K_i = \begin{cases} 1, & 1st\ time \\ 2, & 2nd\ time \\ \left\lfloor B_{ATLS_n} * \frac{v_{slowest}}{v_i} \right\rfloor, & otherwise \end{cases}$$

The last variables introduced with the fraction $\#workers_{active}/\#workers_{expected}$ are used to limit the total number of scheduled iterations by the proportion of active workers. By doing so, given that the workers after the request for collaboration arrive at unknown times, we reserve a portion of work avoiding scheduling too much iterations before all the workers have answered to the request for collaboration.

4.6. Concluding Remarks

In this chapter we have reviewed several well-known scheduling schemes. As the next chapter will reveal with the experimental results, some of them, although being quite simple provide the best results for running environments where several parameters like the number of available processing units is known beforehand and the relative capacity of the participating nodes do not change during the computation. However, these conditions are not fulfilled in all cases, and dynamic environments, with irregular execution patterns and variable characteristics of the available computing resources must be considered. A proposal for these situations in the form of a new scheduling scheme is provided.

We have proposed a new scheduling scheme, called ATLS, based on basic information retrieved at runtime, so no prior information nor details about the participating nodes is required by the scheduler to properly define the tasks assignments that will eventually minimize the overall execution time of parallel loops. The dynamic scheduler implemented in our prototype already provides a useful mechanism for load balancing of loops but it could be adapted with other type of

problems like job schedulers dealing with bag-of-tasks problems. Besides, some of the internal variables defined can be reused for supporting fault tolerant capabilities, like the confidence indicator *CI*, which can be used to provide a hint of the stability of the system.

Chapter 5

Evaluation of Proposals

5.1. Introduction

In this chapter we describe the experiments performed to validate our proposals, implemented in the prototype R/parallel, following the chronology of developments undertaken in our work. From the first results obtained using single desktop multicore computers, to the final results obtained evaluating adaptive scheduling schemes in distributed environments.

5.2. Evaluation of R/parallel with Multiprocessor Computers

In order to assess the capabilities of our proposal, i.e. to take advantage of the available processing power of a multiprocessor computer when running our extension of the R language interpreter, we show in this section the results obtained after performing a set of experiments selected for this purpose.

The experiments has been performed using two different SMP computers: first, two common desktop computer with a single-core and current quad-core processor (i.e. with 1 and 4 processing units respectively) where we have tested if our proposal fits with real R bioinformatics cases while providing improved results, and second, a server with two quad-core processors (i.e. with 8 processing units), where the scalability and efficiency of R/parallel is evaluated.

5.2.1. Evaluating Applicability to Real Cases

Figure 31 shows with an example how easy it is to parallelize a loop in R with R/parallel. In this example, a long vector of gene expression data (i.e. traits) is analyzed through a loop to find quantitative trait loci (QTL) underlying variation in gene expression using a multiple QTL model (MQM) approach [Jan93]. Once a programmer has finished coding and testing his function as usual, he only needs to add the lines shown (i.e. the `runParallel()` function and the IF-ELSE conditional structure) to run it faster in parallel.

```
qtlMapping <- function( map, genotypes, traits )
{
  result <- NULL
  for( idx in 1:nrow( traits ) )
  {
    tmpResult <- MQM( map, genotypes, traits[idx,] )
    result <- rbind( result, tmpResult )
  }
  return( result )
}

qtlMapping <- function( map, genotypes, traits )
{
  result <- NULL
  if( "rparallel" %in% names( getLoadedDLLs() ) ) {
    runParallel( resultVar="result", resultOp="rbind" )
  }
  else {
    for( idx in 1:nrow( traits ) )
    {
      tmpResult <- MQM( map, genotypes, traits[idx,] )
      result <- rbind( result, tmpResult )
    }
  }
  return( result )
}
```




Figure 31: Example of code used to test R/parallel (source [VS08])

Adding the lines explained, the execution time when processing 37685 traits from 73 individuals is reduced, using a quad-core processor, from approximately 4 hours to 1 hour.

Practical applications of parallel computing are to increase the number of finished tasks given a fixed time or to decrease the time needed to perform a long task. To achieve this, the initial problem is partitioned into independent tasks which are computed simultaneously using several processing units. With R/parallel, as previously explained, partitioning is applied to loops, and multiprocessing is used to get access to all the available processing units (i.e. cores in current desktop processors). The benefits of partitioning and multiprocessing are evaluated with two more real cases. The observed speedups demonstrate that loops without data

dependencies can be executed more efficiently using R/parallel. Obviously, with short calculations the speedup is minimal because of the additional overhead raised by the parallelization.

Typical bioinformatics cases where parallel computations are more often used are permutation tests or heuristic searches of multivariate spaces where, due to time constraints, the best result has to be computed before a deadline. Next experiment illustrates the increase of completed analyses (i.e permutation tests) by using all the available processing units. The function `qt1Threshold.sma()` from the package `affyGG` [AV+08] is used with a quad-core processor to analyze a large number of permuted data sets using the same statistical analysis to compute (approximate) significance thresholds. Incrementing the number of parallel processes (i.e. workers) the utilization of more cores has been enabled and therefore the overall performance increased. Figure 32 shows the results of this experiment.

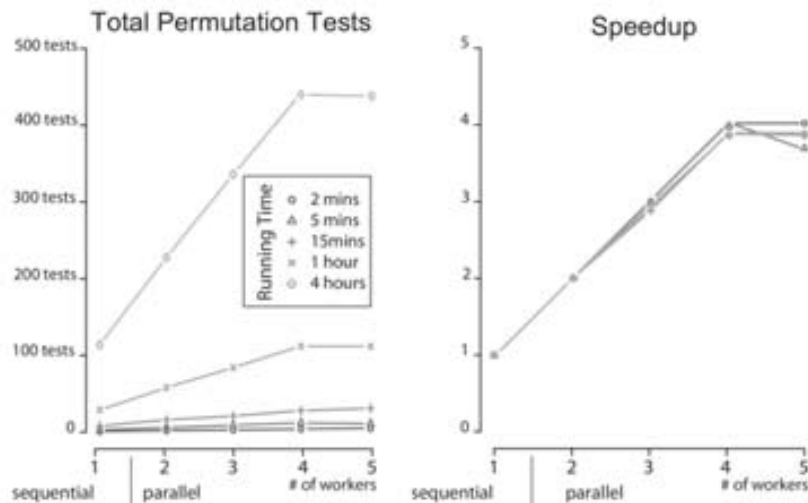


Figure 32: Increasing the amount of work performed before a time limit

It can be observed that the speedup increases almost linearly with the number of used cores. Setting more workers (5) than existing cores (4) does not improve the results.

The next experiment evaluates a case which objective is to reduce the elapsed time required to process a fixed workload. The case illustrates the results obtained after parallelizing a program. The function `qt1Map.xProbe()` from the package `affyGG` is used with a single-core processor to compute the same statistical analysis over large data sets. In this case, due to the way memory is managed in R, with linked lists, and as a result of partitioning, small and faster tasks (with faster data indexing) are created. As a consequence, in cases like this, it shows super linear speedup, even with a single processing unit, the total execution time is reduced. Figure 33 shows how the

super linear speedup is more accentuated when using 4 processing units and setting up from 2 to 5 worker processes.

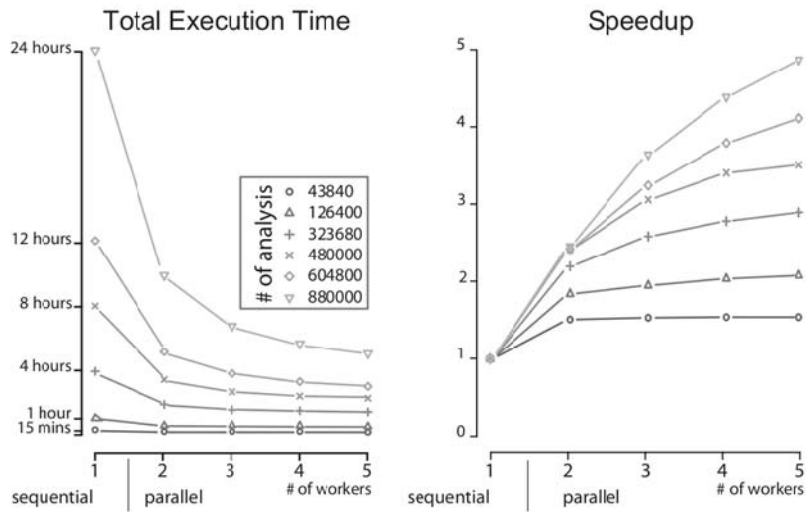


Figure 33: Decreasing the required time to process a fixed workload

5.2.2. Evaluating Scalability and Efficiency

Once the applicability of our proposal has been proved successful using several real cases, we are interested in evaluating the scalability and efficiency of R/parallel using a multiprocessor computer, so we set up a new experiment with a larger workload and the double of processing units used at previous experiments.

The test environment consist of one server equipped with 2 quad-core processors (i.e. 8 cores available) and 16 GB of main memory running the operating system Red Hat Enterprise Linux Server release 5 and the R language interpreter, version 2.8.1. Figure 34 shows the results obtained of total execution time.

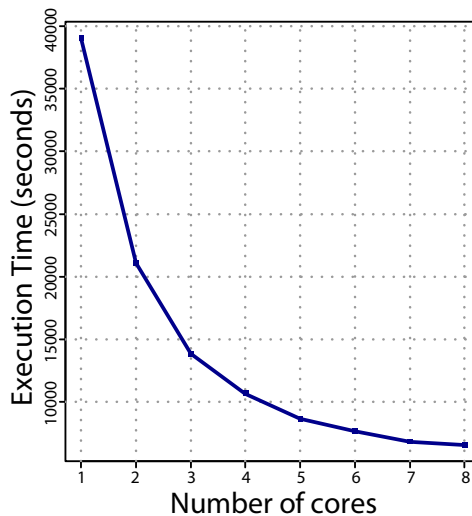


Figure 34: Decrease of the total execution time parallelizing `qtlMap.xProbeSet()`

The tests performed have been done using the function `qtlMap.xProbeSet()` from the R add-on package `affyGG` [AV+08]. `affyGG` has been developed to perform bioinformatics QTL analysis of samples obtained using Affymetrix microarrays. The input data has been simulated using real data obtained from samples of 30 recombinant inbred mice from [BW+05] to obtain a total execution time of the R function of more than 10 hours without using any parallel solution. By this way, adding progressively more cores to the computation (the number of workers can be set optionally), when running with R/parallel we can observe how the scalability and efficiency of this solution evolves as we add more cores.

It can be observed that the total execution time decreases dramatically as we repeat the experiment and we add more processing units. Since parallel loops are included within the class of embarrassingly parallel problems, this output is expected. Figure 35 shows the corresponding speedup obtained for this experiment.

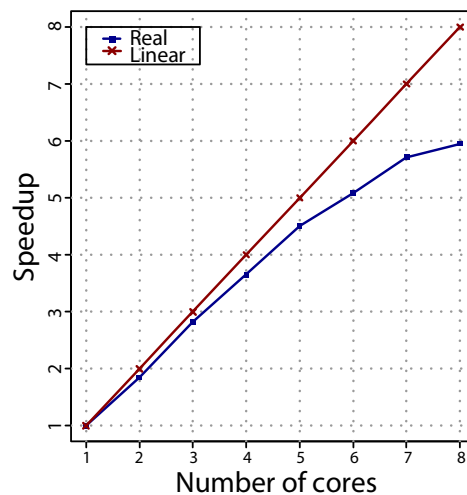


Figure 35: Speedup decays with introduction of processing units

Looking at the speedup, although initially close to the linear speedup, it is clear that, because of the overhead introduced with the management and control of the parallel execution, increasing the number of processing units, the performance growth rate is affected negatively. Figure 36 shows the efficiency figures obtained.

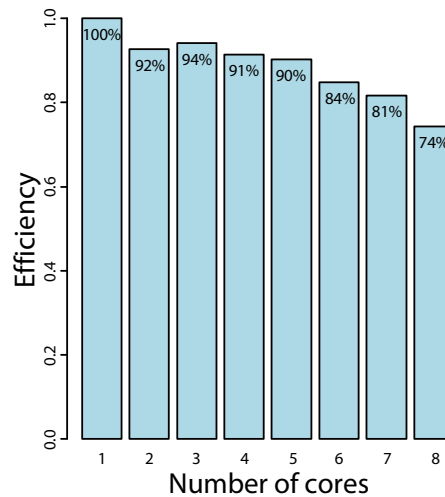


Figure 36: Efficiency figures obtained with 8 cores

The less efficient case is observed when using 8 cores. Besides of the system processes, we also have to take into account the master process. When reaching the maximum number of available cores the machine is overloaded because of the competence between all the processes trying to get their corresponding slice of processor time. As a consequence, the overall performance is affected and the results, although still reducing the total execution time, show a worse efficiency using 8 cores than using other smaller configurations.

Nevertheless, the results demonstrate that even with a few available cores, our proposal, by enabling the available computational power of nowadays multicore processors, and with so little effort by the R user, is able to run parallel loops in R scripts substantially faster than previously without our contribution.

5.3. Evaluation of R/parallel with Distributed Systems

In this section we describe the experiments undertaken and its corresponding results, to assess, with a real and working system that our proposal for distributed computing with R is feasible, evaluating the results obtained using homogeneous and heterogeneous environments, with non-adaptive and adaptive scheduling schemes.

5.3.1. Evaluating Distributed Computing in Homogeneous Environments

Although our proposal has been designed for heterogeneous environments, before analyzing how well it fits in these environments, first we need to evaluate its results using a less complex running scenario, an homogeneous environment where we can determine if our proposal is feasible for distributed computing.

For that experiment we have set up a test environment depicted in Figure 37 built with five identical computers connected through a gigabit ethernet network and an internal configuration of two quad-core processors and 16 GB of main memory each, adding a total of 40 cores, and running R version 2.8.1 under the operating system Red Hat Enterprise Linux Server release 5.

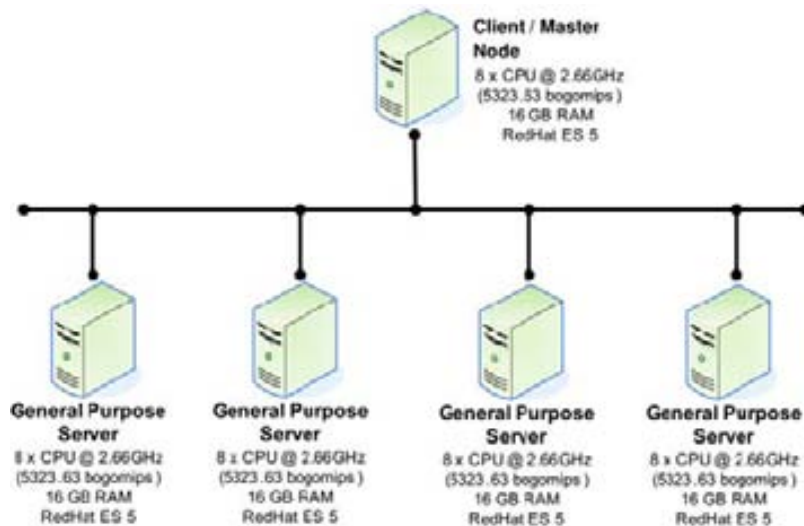


Figure 37: Network diagram and configurations of homogeneous test bed with 40 processing units

Another practical consideration when comparing these results with real scenarios is that, since our test environment is built with dedicated computers, its overall performance is expected to be higher than the results that eventually can be obtained with common desktop machines. Therefore, the performance figures obtained must be interpreted only as an upper bound to what is expected in for example office environments.

The problem selected for our performance tests, reproduced in Figure 38, uses the R package R/qlt [BW+03]. We iterate a 100 times over the function `scantwo()` that calculates just one heavy load permutation and aggregates its results in a vector (i.e. `opermT`) for later utilization.

```

library(rparallel)
library(qtl)
data(hyper)

test1 <- function( nPermutations, localWorkers )
{
  hyper <- calc.genoprob(hyper, step=2.5)
  opermT <- NULL

  if( "rparallel" %in% names( getLoadedDLLs() ) )
  {
    runParallel( resultVar="opermT", resultOp="c",
                 nWorkers=localWorkers, useNetwork=T )
  }
  else
  {
    for( i in 1:nPermutations )
    {
      permBuff <- scantwo(hyper, n.perm=1)
      opermT <- c( opermT, permBuff )
    }
  }
  return( opermT )
}

```

Figure 38: Script used to parallelized loops with R/qtl package functions

The only basic difference between the structure shown in Figure 38 and the one used in the previous non-distributed version of R/parallel (see Figure 21), is the introduction of the argument `useNetwork` to activate the utilization of remote workers. The argument `nWorkers` is used to define, if desired, the number of local workers to be spawn. In the remote side exists an identical optional argument to define the number of volunteered workers (by default defined with the number of processing units present in the computer, i.e. the capacity slots). Figure 39 shows the obtained total execution time after performing 100 iterations, starting with 1 core and increasing up to the 40 available cores.

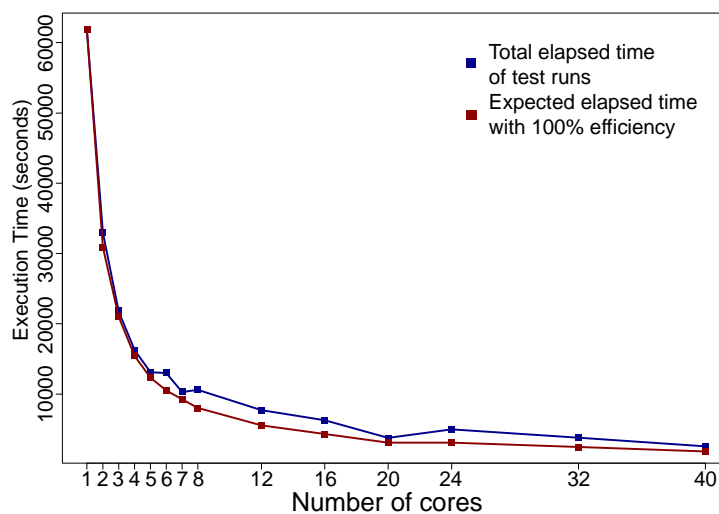


Figure 39: Performance results using 40 workers

As it can be observed, the execution time decreases so quickly with the addition of the first 8 cores. After that, the improvement obtained per added core is in comparison very low. This degradation of the increased performance obtained per processing unit added, as defined by Amdahl's law [Amd67], is caused in this experiment by the sequential section of our program, in our case, the time required to calculate 1 iteration. Although this limitation depends on the program being parallelized, it illustrates the fact that many applications are unable to efficiently take advantage of large amounts of computing nodes and that the highest improvements are obtained with the first aggregated nodes.

5.3.2. Evaluating Parallel Loop Scheduling Schemes

Before we are able to evaluate our proposal using heterogeneous environments we need to select a parallel loop scheduling scheme. In Chapter 4 several scheduling schemes have been reviewed. Next we evaluate the results obtained using a selection of schemes in a controlled homogeneous environment where we modify different parameters to observe their influence over the obtained performance results. Later on, once we learned about the behavior of these schemes on homogeneous environments we will evaluate their results also in heterogeneous environments.

The test environment is made up of the same dedicated and identical computers used at the previous experiment (see Figure 37), but this time changing the R script as well as the experimental conditions. The problem to be solved during the experiments consists in preprocessing a set of DNA sequences obtained by pyrosequencing [MM+05] with a Genome Sequencer FLX from Roche. Before performing any main analysis, raw data obtained from NGS devices [Bub08] like this usually requires some filtering and transformations steps in the retrieved files in order to adapt and prepare the data to the specific needs of each particular analysis. In this experiment, the input data includes DNA sequences, already converted to FASTA format files, which have to be preprocessed to find the original sample sequence, identify the contributing individual, and define other interesting parameters like sequence lengths or read qualities. In our case, we use some functions from the R package Biostrings, which belongs to Bioconductor [GC+04], a popular software project for the analysis and comprehension of genomic data which is extensively used among the bioinformatics community.

Since the same operations are performed for each DNA sequence, and each sequence processing is independent from the others, a parallel loop can be set up and run with our prototype to take advantage of parallel computing. Figure 40 shows a code snippet of this R script, highlighting the IF-ELSE statement used to parallelize the loop boxed in color grey. Later on, on each different experiment, the number of

sequences to be processed is modified in order to change the workload and observe the behavior of each scheduler.

```

library(Biostrings)
library(rparallel)

# Main function
miRNAFilter <-function( inputfna=NULL, inputqual=NULL,
                        outputcsv="output.csv", scheduler="static" )
{
  [...]
  # Parallel section of the function *
  parallelProcessing <-function( )
  {
    [...]

    if( "rparallel" %in% names( getLoadedDLLs() ) )
    {
      runParallel( resultVar=c( "startL5b", "endL5b", "startL3b", "endL3b",
                               "miRNAseqb", "miRNAlenb" ),
                  resultOp=c( "c", "c", "c", "c", "c", "c" ),
                  listenPort=2009, useNetwork=T,
                  scheduler=scheduler )
    }
    else {
      for( idx in 1:nSamples )
      {
        # 1. Find Linkers -3 fields
        initL5 <-matchPattern( linker5i_9, sTCA[[idx]], max.mismatch=1 )
        finL5 <-matchPattern( linker5f_8, sTCA[[idx]], max.mismatch=3 )
        initL3 <-matchPattern( linker3i_5, sTCA[[idx]], max.mismatch=1 )
        finL3 <-matchPattern( linker3f_5, sTCA[[idx]], max.mismatch=1 )
        [...]
      } #end-for
    } #end-else
    [...]
    return( list( startL5b, endL5b, startL3b, endL3b, miRNAseqb, miRNAlenb ) )
  } # parallelProcessing *

  # Just call the parallel function enclosing the parallel loop
  results <-parallelProcessing()
  [...]
}

```

Figure 40: Code snippet of a parallelized clipping procedure of DNA sequence reads

Although the main objective of our implementation is to find a suitable scheduler for highly changing environments, it is important to observe the results also in homogeneous environments. Since system changes are not continuously happening during computations, it is important to validate which schedulers perform reasonably well also in invariable running conditions using identical computers. The results of these experiments are shown in Table 3.

	TOTAL EXECUTION TIME <i>(seconds)</i>			THROUGHPUT <i>(iterations/second)</i>		
# workers	4	16	32	4	16	32
# iterations	1000			1000		
Sequential	83,64	83,64	83,64	11,96	11,96	11,96
Static	27,86	13,73	15,2	35,90	72,82	65,78
PSS	103,15	33,75	24,48	9,69	29,63	40,86
FSS	29,88	15,56	14,87	33,46	64,27	67,26
WF	30,28	15,39	15,52	33,02	65,00	64,45
AWF	30,3	15,46	14,8	33,00	64,69	67,55
AF	30,41	16,41	15,68	32,89	60,95	63,79
	MAX	103,15		MAX	72,82	
	MIN	13,73		MIN	9,69	
# iterations	10000			10000		
Sequential	859,03	859,03	859,03	11,64	11,64	11,64
Static	224,98	73,61	53,79	44,45	135,86	185,92
PSS	1175,75	325,67	339,02	8,51	30,71	29,50
FSS	228,06	76,12	56,92	43,85	131,37	175,69
WF	228,06	78,80	57,84	43,85	126,91	172,89
AWF	311,22	77,20	57,30	32,13	129,53	174,53
AF	250,63	77,12	61,93	39,90	129,68	161,48
	MAX	1175,75		MAX	185,92	
	MIN	53,79		MIN	8,51	
# iterations	100000			100000		
Sequential	9657,42	9657,42	9657,42	10,35	10,35	10,35
Static	2764,48	839,78	618,89	36,17	119,08	161,58
PSS	14681,31	6420,16	6238,22	6,81	15,58	16,03
FSS	2712,33	878,22	653,93	36,87	113,87	152,92
WF	2717,67	847,99	633,51	36,80	117,93	157,85
AWF	2715,83	846,84	646,53	36,82	118,09	154,67
AF	3230,45	869,49	683,64	30,96	115,01	146,28
	MAX	14681,31		MAX	161,58	
	MIN	618,89		MIN	6,81	

Table 3: Total Execution Time and Throughput results of several parallel loop scheduling schemes using homogeneous environments

It can be observed that for each scheduling scheme, the same experiment has been run several times with three different workloads: 1000, 10000 and 100000 iterations, and with three different number of workers: 4, 16 and 32 workers. The scheduling scheme that provides the best results is the STATIC scheme while the worst, due to the high overhead introduced in distributed environments is the PSS scheme. Regarding the factoring based schemes, both non-adaptive and adaptive ones, we observe that their results are quite similar, although the non-adaptive schemes (i.e. FSS and WF), being this an static environment without changes, provide slightly better results than the adaptive schemes (i.e. AWF and AF). Among them, WF provides the best average result figures while AF presents the worst results for this experiment. Next, showed in Table 4 we also provide the scalability and efficiency observed in this set of experiments.

	SEQ vs PAR SPEEDUP (T_{seq}/T_{par})			EFFICIENCY (%)		
# workers	4	16	32	4	16	32
# iterations	1000			1000		
Sequential	1	1	1	100,00%	100,00%	100,00%
Static	3,00	6,09	5,50	75,06%	38,07%	17,19%
PSS	0,81	2,48	3,42	20,27%	15,49%	10,68%
FSS	2,80	5,38	5,63	69,97%	33,60%	17,58%
WF	2,76	5,44	5,39	69,05%	33,98%	16,84%
AWF	2,76	5,41	5,65	69,01%	33,81%	17,66%
AF	2,75	5,10	5,34	68,77%	31,86%	16,67%
	MAX	6,09		MAX	75,06%	
	MIN	0,81		MIN	10,68%	
# iterations	10000			10000		
Sequential	1	1	1	100,00%	100,00%	100,00%
Static	3,82	11,67	15,97	95,46%	72,94%	49,91%
PSS	0,73	2,64	2,53	18,27%	16,49%	7,92%
FSS	3,77	11,29	15,09	94,17%	70,53%	47,16%
WF	3,77	10,90	14,85	94,17%	68,14%	46,41%
AWF	2,76	11,13	14,99	69,00%	69,54%	46,85%
AF	3,43	11,14	13,87	85,69%	69,62%	43,35%
	MAX	15,97		MAX	95,46%	
	MIN	0,73		MIN	7,92%	
# iterations	100000			100000		
Sequential	1	1	1	100,00%	100,00%	100,00%
Static	3,49	11,50	15,60	87,33%	71,87%	48,76%
PSS	0,66	1,50	1,55	16,45%	9,40%	4,84%
FSS	3,56	11,00	14,77	89,01%	68,73%	46,15%
WF	3,55	11,39	15,24	88,84%	71,18%	47,64%
AWF	3,56	11,40	14,94	88,90%	71,28%	46,68%
AF	2,99	11,11	14,13	74,74%	69,42%	44,15%
	MAX	15,6		MAX	89,01%	
	MIN	0,66		MIN	4,84%	

Table 4: Scalability and Efficiency results of several parallel loop scheduling schemes using homogeneous environments

For the case of 1000 iterations it is clear that the problem is too small for the test application used since the total execution time and the associated speedup practically does not change once 16 workers are used. That situation has a negative effect in the efficiency which shows very low values (under 20%) for the 32 workers experiments. It can also be observed that the difference between the experiments with workloads of 10000 and 100000 iterations are quite similar. That will allow us to compare speedup and efficiency results obtained with workloads within the same range (provided the other experimental conditions are not changed significantly).

The conclusion of this set of experiments is that, taking into account the results obtained, the STATIC scheduling scheme, at least in homogeneous environments, and knowing beforehand the number of available processors is the one that provides the best results. However, if we have to choose a scheduling scheme that does not

require the number of processors, the WF scheme seems the right choice. However, given that in further experiments we want to evaluate the results obtained in changing environments, the best adaptive scheme, and the selected for our next experiment, is the AWF scheduling scheme.

5.3.3. Evaluating Distributed Computing in Heterogeneous Environments

The problem to be solved during these experiments consists in the same problem of preprocessing DNA sequences, but this time using a subset of 47154 reads. In order to evaluate to which extent we can benefit from aggregating the computing power of non-dedicated computers we have used three different types of computing resources: 1 standard PC, 2 general purpose servers and a cluster with 30 computing nodes. The whole system adds up 140 processing units (i.e. cores). Figure 41 shows a diagram of the computers used at these experiments including some details of their internal configuration.

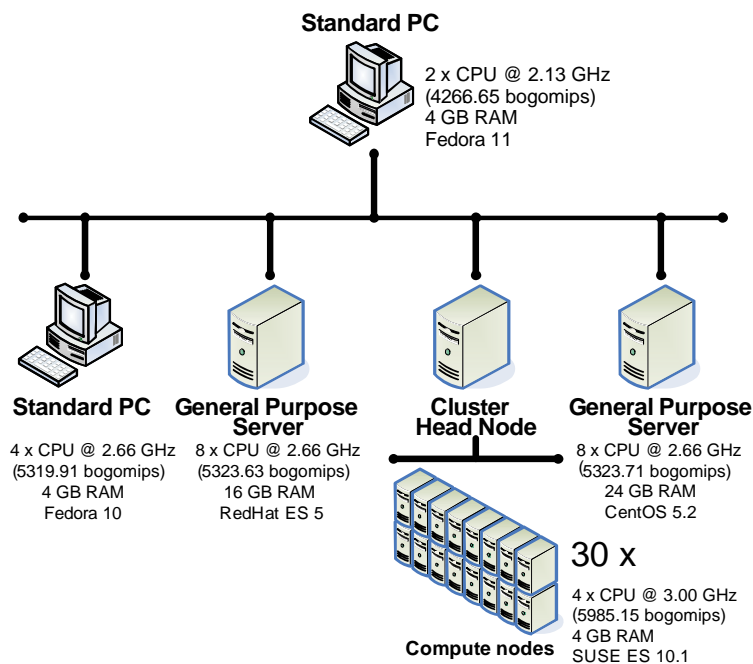


Figure 41: Network diagram and configurations of heterogeneous test bed with 140 processing units

The scheduling scheme used at this set of experiments is the AWF scheme. The objective is to evaluate its performance using heterogeneous environments like the one described, but also to inspect if there are significant differences when applied to homogeneous environments. To do so we will use two subsets of processing units,

one using all the available computers, so defining an heterogeneous system, and the other only using computers from the compute cluster.

It must be taken into consideration that the servers, as well as the compute clusters, are usually shared with more users, but for running our experiments, in order to obtain reproducible results, we have used them without interferences from other users.

The processing of the 47154 DNA sequences in a standard PC like the one depicted in Figure 41 with R v2.9.1 and no parallel improvements requires 1372 seconds. In a hypothetic scenario where we have an additional faster PC at hand, for example a quad-core computer, we can take advantage of their resources and run a parallel version of the original loop. Only requiring from the client side to add the address of the collaborating node, for example, typing:

```
> add.volunteer("node1.subnet.com")
```

and setting a volunteer listener on the quad-core PC, just typing on the other side:

```
> install.volunteer()
```

```
> grant.access(node="colleague1.subnet.com")
```

and running again the program. Doing that we can reduce the total execution time from 1372 to 386 seconds. In order to achieve additional performance, two more general purpose servers, with two quad-core processors each, were added in a similar way, reducing the elapsed time to 203 seconds. Finally, the same listener was run from the user account in the head node of a compute cluster managed by Sun Grid Engine, allowing the access to up to 30 additional nodes of highly tuned computers with two dual-core processors. Adding these additional computing resources, the elapsed time was reduced down to 197 seconds.

These initial results reveal a problem of scalability for the cases used at these experiments. After using 20 processing units, the performance almost does not improve, even doubling the number of faster nodes from the cluster. What have happened is that the performance is limited by the slowest processor, that in this case corresponds to the standard PC. In order to demonstrate this limitation, the experiments have been repeated, but this time only using the nodes from the cluster to be compared with previous results. Figure 42 shows a comparison of the total execution times obtained using several heterogeneous computers, versus the elapsed time obtained using the same number of processing units, but in this case the faster homogeneous nodes from the cluster.

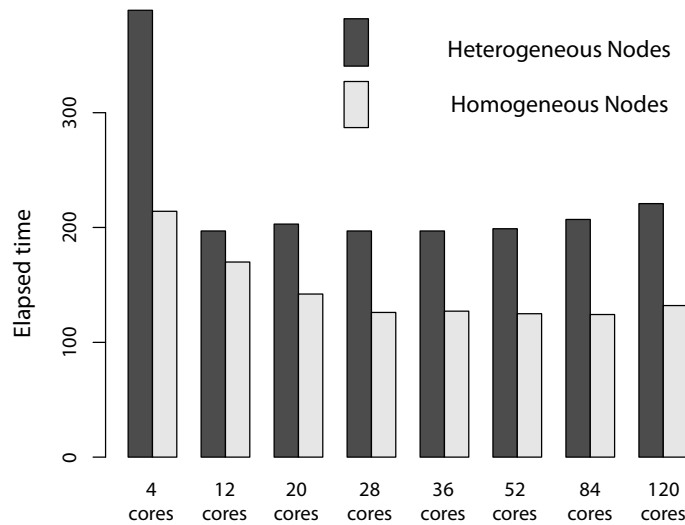


Figure 42: Total execution times with 120 workers using homogeneous and heterogeneous environments

The pending question at this point is why the implemented dynamic scheduler (i.e. AWF) was unable to provide a more efficient load balance which assigned more work to the faster processors and avoid this inefficiency with the presence of significantly faster computing nodes. The answer comes from two issues.

The first and most important reason is because the implementation is designed for best-effort environments where resources are contributed under request but their availability and arrival time are unknown. Since access to distributed resources is never guaranteed, the computation must start as soon as the first nodes answer the requests and join the computation. Although the scheduler assigns very small chunks of work at the first stages of the computation (one and two iterations consecutively), after a stabilization period that depends on the iterations bounds of the first iterations used to measure the relative performance of each node, the scheduler begins to dispatch bigger chunks among the available working nodes. It can happen that at this time, a considerably large number of volunteer nodes still have to arrive. For the time these delayed nodes join the computation, a major portion of the iterations have been already assigned, and they receive, from the remaining iterations, proportionally smaller jobs to what they should have received given their performance ratio. This behavior of the scheduler, assigning large chunks at the beginning and smaller, down to one iteration, as we reach the final of the computation was described as an important characteristic of schedulers evolved from GSS and Factoring scheduling schemes. It is intended for reducing the system overhead, distributing the major part of the workload at the beginning, and reducing the unevenness by aligning the finalization time of the participating nodes.

The second issue arises as a consequence of the previous one plus the delay introduced by the job scheduler at the SGE managed cluster responsible for dispatching the submitted jobs, and hence raise additional workers at the faster compute nodes. During the execution of these experiments, arbitrary delays, up to 5 seconds, were observed for jobs encoding worker launch scripts, already submitted, idle in the SGE queues, and waiting for reaching a compute node and subsequently start a worker to join the distributed computation. As a consequence, the fast nodes of the cluster arrive once several big tasks have been already assigned to slower nodes. As a result, the system has to wait for the finalization of the big tasks assigned to the slower nodes. Figure 43 shows the sequence and size of tasks arrivals, i.e. tasks processed and returned by the workers, for the two experiments where 36 cores have been used.

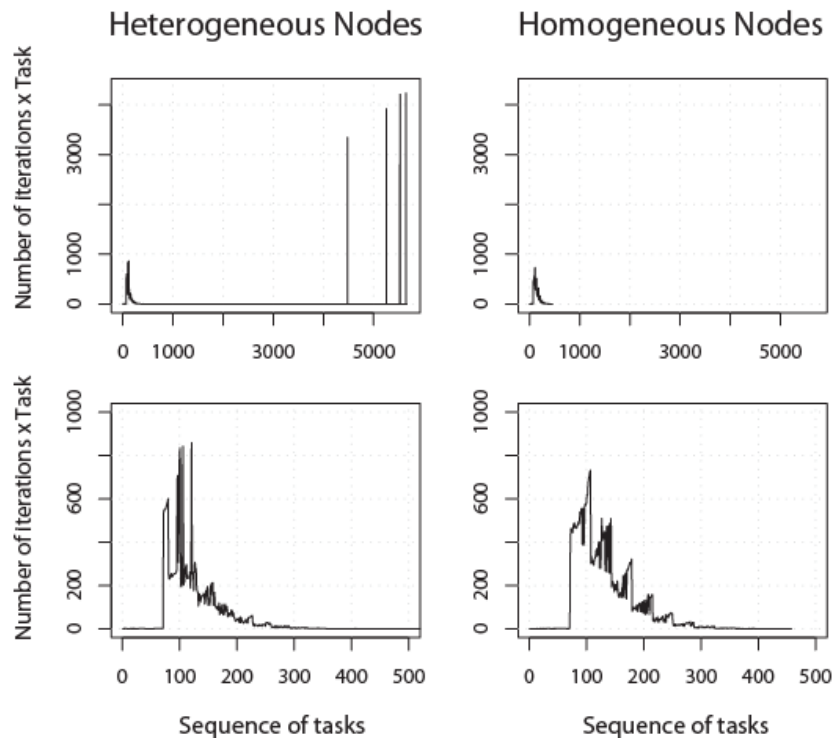


Figure 43: Ordered sequence and size (i.e. number of iterations) of task arrivals. Second row illustrates a zoom of the first 500 tasks completed

In the first row the complete sequence of tasks, for both cases, are shown. It can be observed that for the case of the heterogeneous nodes, the curve is considerably longer and four peaks appear at the right end. These peaks correspond to the four tasks assigned to the slower nodes which correspond to the four workers running in the quad-core desktop computer. During the waiting time, after all the iterations have

been assigned, empty tasks (NOP operations) are assigned to further requesting workers.

The second row of Figure 43 shows the first finalized 500 tasks for the same experiments in order to allow a more detailed analysis of the elapsed distribution of iterations. In the case of the homogeneous nodes, it can be observed that the area under the curve is larger than the one for the heterogeneous nodes, i.e. within the first 500 tasks, in the homogeneous environment more iterations have been processed. Since for the case of the heterogeneous nodes, as we have already seen, there are 4 large tasks still being processed, there is less workload to be distributed, and therefore, a smaller area is obtained.

Finally, independently of the not-so-efficient assignments for the heterogeneous case, it can be observed that both experiments show the same triangle behavior that can be deduced from the previous discussion, a large amount of work is distributed at early stages of the computation and it is gradually reduced while we reach the end.

At this point some issues arise: first, what kind of results can be obtained using non-adaptive schemes in an heterogeneous environment? And second, what happens if the workload changes? Will AWF still show the same behavior? To answer these questions we designed a new experiment, in that case to evaluate again AWF with the STATIC scheme.

The program used in these experiments, running R version 2.10.0 and our prototype, consists in preprocessing a different set of DNA sequences, this time ranging from 10000 to 434000 reads. The program has to find the original sample sequence, identify the contributing individuals (the donors of pig tissue in that case), and define as before sequence lengths and qualities. The code used is the same showed previously in Figure 41.

Another change introduced is an additional cluster with 10 relatively old uniprocessor computers to increase the heterogeneity of the test-bed used at the experiments. Figure 44 shows the new experimental environment.

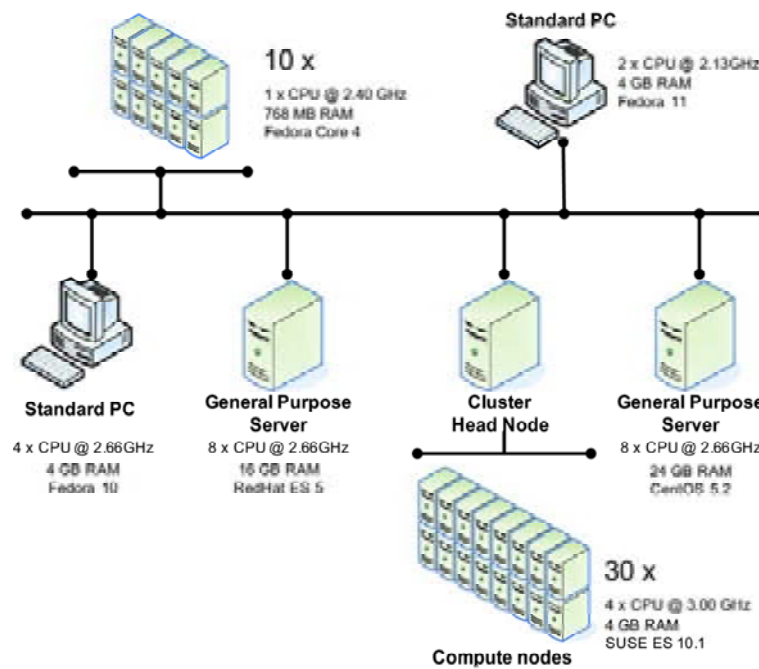


Figure 44: Network diagram and configurations of heterogeneous test bed with 150 processing units

In the first experiment we have processed 434000 sequences with 80 and 20 workers using a mixture of fast and slow nodes. As it can be observed in Figure 45 for the case of 80 workers the STATIC scheme, knowing beforehand the number of working nodes (what is not always possible), provides way better results than the AWF scheme.

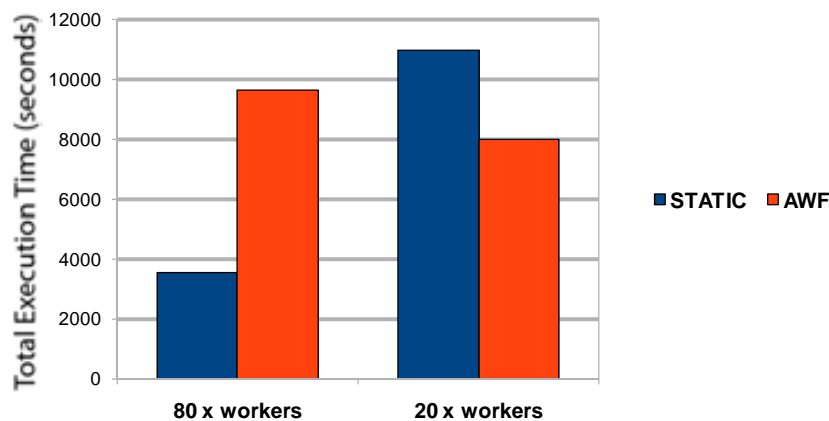


Figure 45: Experimental results processing 434000 sequences with 80 and 20 workers using a mixture of fast and slow nodes

Although AWF provides better results than STATIC for the case of 20 workers, which have a higher degree of heterogeneity in comparison to the 80 workers case (where

most of the nodes come from the compute cluster), there are obviously heterogeneous situations that can be handled more efficiently by adaptive schemes like AWF. We can guess that maybe there is a problem related with the proportion of quickly accessible but slow nodes (from the 10x node cluster) versus the delayed access but very fast nodes (from the 30x node cluster). In order to try to find out an answer for that issue, we run another experiment. This time processing 10000 sequences varying the proportion of slow and fast nodes from 0-10 to 10-0 exchanging 2 nodes of each type at a time. The obtained results are shown in Figure 46.

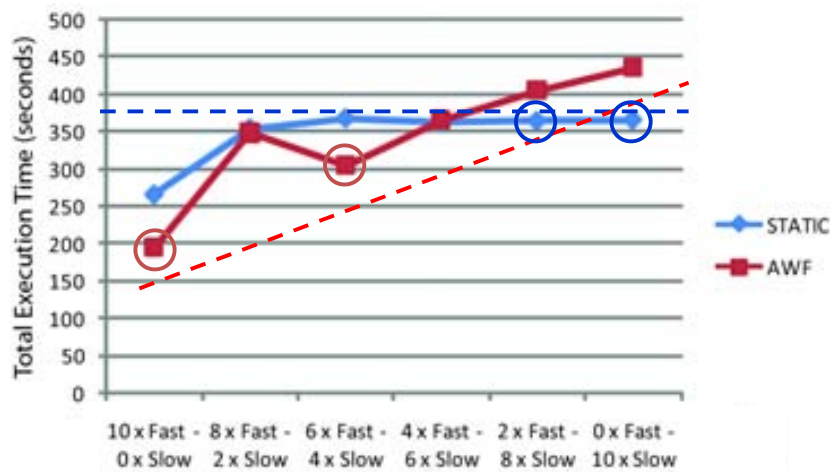


Figure 46: Experimental results processing 10000 sequences varying the proportion of slow-fast nodes from 0-10 to 10-0 exchanging 2 nodes of each type at a time

As it can be observed with the red slashed line, AWF balance the workload but not as efficiently as possible. When using slow nodes the total execution time obtained with AWF is penalized. That is caused by the fact that answering first the slow nodes (more responsive), the AWF has to take a scheduling decision and assigns too much iterations at the beginning. When the fast nodes (behind SGE queues) are finally available there is less work to assign and the performance is affected. The horizontal blue slashed line shows that the performance obtained by STATIC is limited by the slowest processor, since all workers receive the same subset of the workload.

Finally, taking into account the best scheme for each experiment, indicated with red and blue circles, we observed that in average, for this experiments and knowing the number of processors, both schemes can provide equally good results. At this point we ask ourselves, given that STATIC is way easier to implement, if using the STATIC scheme and guessing the future number of available workers we can get better results than if using the AWF scheme. The next experiment, which results are illustrated in Figure 47 provides the answer.

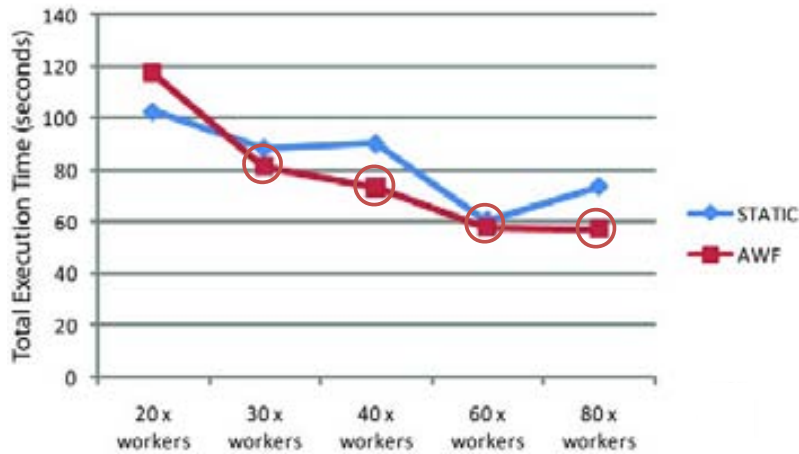


Figure 47: Experimental results processing 10000 sequences with 20, 30, 40, 60 and 80 homogenous workers with AWF and a fixed number of P=40 workers configured for STATIC

In that case 10000 sequences are processed again, but this time with 20, 30, 40, 60 and 80 workers from the fast cluster and with a fixed number of 40 workers configured for the STATIC scheduler. The results show that when using less workers than configured, the STATIC scheme in one case can perform better than AWF up to a 12.4%. However, even in this homogeneous environment, in case we have more nodes than configured, the difference goes to 22.7% in favor of AWF. As a conclusion for this section we can summarize that even for heterogeneous environments, if a good approximation of the number of nodes is available, the STATIC scheme can in some cases provide better results than AWF. However, this number of nodes is not always known beforehand. With an uncertain number of workers the results demonstrate that an adaptive scheduling scheme like AWF, in most of the cases delivers very good results as it was expected.

These experiments have shown that running parallel loops with R it is also possible to improve the processing times by aggregating the performance power of distributed heterogeneous computers. However, the limited knowledge about the future availability of resources, inherent to best-effort environments, prevents the scheduler from defining more efficient task assignments. Nevertheless, in these kind of scenarios, the main objective is to provide improved performance ratios taking advantage of existing resources. The results show that our prototype implementing adapted well-know parallel loop scheduling schemes achieves good performance figures. However, as we have already seen, there are several aspects that can be improved, and hence new contributions provided.

5.4. Evaluation of the New Scheduling Scheme ATLS

The objectives of the experiments described in this section are two: first, evaluate the scheduling scheme proposed, i.e. ATLS, in several running environment involving

the execution of parallel loops, and second, partially included in the first objective, evaluate the results obtained with our prototype using dynamic environments.

The experiments have been performed in a controlled scenario made up of 5 identical computers like the one depicted in Figure 37. One computer is used as the master component, from where the user launches his or her programs, while the other four are used as the volunteered computers with the worker components, providing a total of 32 worker nodes. The program run at the experiments is the same R script used in previous experiments (see Figure 40), but in this case using a workload from ~1000 to ~100000 DNA sequences reads. During the experiments several parameters have been modified to evaluate its influence on the design of the proposed scheduling scheme. They are basically: the number of iterations or workload, the number and type of workers and the schemes selected at the scheduler.

5.4.1. Evaluating ATLS in a Static and Homogeneous Environment

Although ATLS has been designed for dynamic environments, first of all we must determine that its performance is satisfactory also in static environments. The test-bed described without modifications serves our requirements. Figure 48 shows the obtained total execution time and its corresponding speedups for different setups.

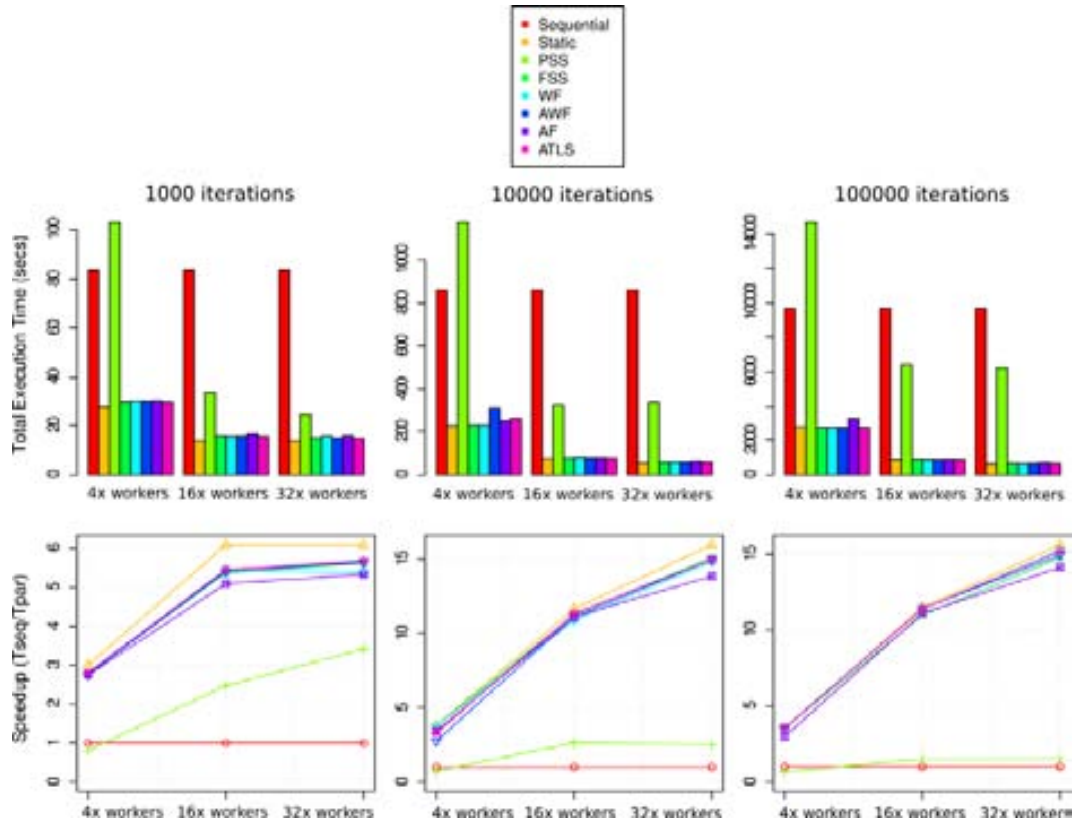


Figure 48: Total Execution Time and Speedup using 4, 16 and 32 workers in an homogeneous environment

As it was expected for an environment without changes, the simple scheme STATIC provides the best performance. In that case the schemes AF and AWF, designed for dynamic environments, provide respectively an average result 10.36% and 8.88% worse than STATIC while for ATLS the difference is smaller with a result 4.68% worse. FSS and WF, best fitted for static environments, obtain results 2.93% and 3.16% worse than STATIC, performing slightly better than ATLS in that case. Finally, it can be observed that the efficiency (speedup / number of workers) is very similar for ~ 10000 and ~ 100000 iterations, decreasing down to $\sim 50\%$. Taking that into consideration, next experiments are focused in using 32 workers and a workload of ~ 100000 iterations.

5.4.2. Evaluating ATLS in a Static and Heterogeneous Environment

Here we evaluate the results using different performance ratios per worker to emulate an heterogeneous environment. To achieve this the program `cpulimit` [Mar09] is used to limit the maximum percentage of share of processor time a worker process receives when running. Using this tool the processor usage has been limited for the 32 workers with the following values: 8x 90%, 8x 75%, 8x 25% and 8x 10%. To simplify the comparison of results, PSS and FSS have been discarded given its results does not provide additional information for our comparison. The rest of the setup is maintained from the previous experiments. The total execution time elapsed in this experiment is illustrated in Figure 49.

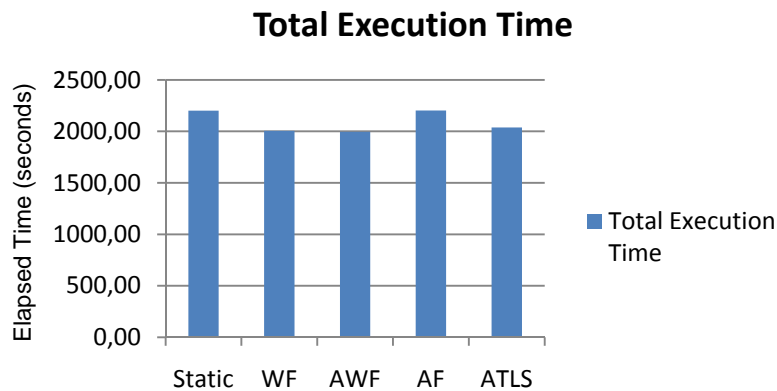


Figure 49: Total Execution Time using an heterogeneous environment with 32 workers

The scheme AWF has shown the best performance in a static environment with heterogeneous workers (1996 seconds), closely followed by WF (0.46% worse). With

a 10.23% worse than AWF, STATIC is penalized for assigning the same task size to non-equal workers. For the case of AF the results are similar, a 10.31% worse. The main reasons for this behavior can be found observing the bar plot with the number of iterations processed by each worker (vertical axis), as it is illustrated in Figure 50. Horizontal axis represents the 32 workers used at the experiments. It must be noticed that the master component do not have any extra information about the characteristics or identities of the workers. Just their registered task turnaround completion time. As a consequence, at this point we cannot find out the mapping between the worker identified by the master, to which it has dispatched the number of iterations show below, and the process running the worker in the remote side, which has their share of CPU limited.

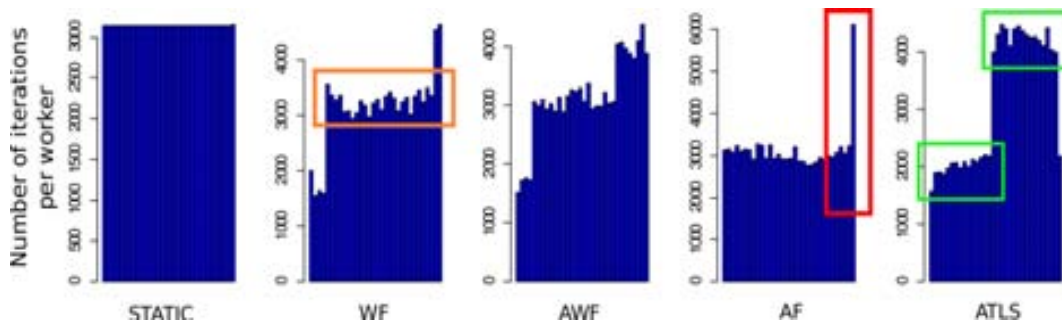


Figure 50: Number of iterations per worker dispatched by each scheduler

First, almost all workers are processing ~ 3000 iterations, like in the case of STATIC, AF or WF indicated with an orange box. Besides, for AF, indicated with a red square, an additional reason must be considered. There is one worker with almost the double of iterations assigned. Since there are 8 workers with the highest limitation (and best performance) and only one has received such task, we can deduce the decision of scheduling such amount of iterations it is not performance related. In fact, it is related with the sequence in time at which workers arrive and start its collaboration with the master. AF is too eager to schedule most of the iterations at the beginning, before all workers arrive, and schedules bigger tasks assuming no more workers are expected. Since AF does not provide better results than the other schedulers, what we see is an overloaded worker that delays the whole computation.

For the case of ATLS, it performs on average just a 2.06% worse than AWF in this experiment. Additionally, as it can be observed in green boxes, the scheduler with ATLS has correctly identified two classes of workers (i.e. with high and low CPU limits), which shows a better adjustment to the heterogeneous resources. However, in this experiment, a static environment, the scheduling decisions taken by AWF, less sensitive and eager, ends up with better final results.

5.4.3. Evaluating ATLS in a Dynamic Environment

Next we evaluate the results in different experiments where the performance of workers varies during the computation. Again we use the `cpulimit` tool, but this time programming not only the share of CPU but also the instant in time at which the restriction will start and end. By doing so we are able to, modifying dynamically the worker performances during the computation, emulate the external interferences that eventually restrict their processing capacities, quite common in shared multiprogrammed computers.

Three experiments have been set up: step-down, step-up and mixed. At the first experiment, step-down, all workers start at the same full capacity, but after 240 seconds their performance is limited during 900 seconds with the following values: 8x 50%, 8x 25%, 8x 10% and 8x 5% (number of workers x CPU upper limit). For the second, step-up, the opposite limitation has been programmed. All workers start with the same restrictions, but after 240 seconds the processes run at full capacity. For the third one a mixture of behaviors has been programmed (see Figure 51). This includes one machine with 8 step-down type limitations, one with 8 step-up type limitations, one with 8 pulse type limitations (i.e. concatenations of steps) with constant limits during 120 seconds per pulse and a last one with 8 pulse type limitations with variable CPU limits.

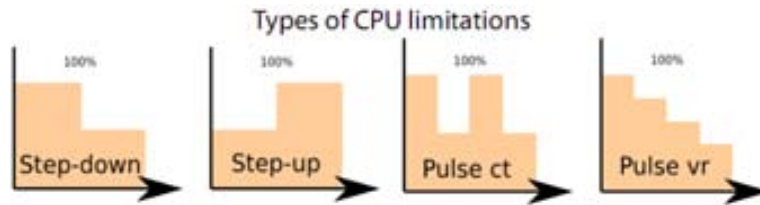


Figure 51: Limitation types programmed: step-down, step-up, pulse constant and pulse variable

The average total execution times obtained in this emulated dynamic environment are shown in Table 5. Since the order and type of workers varies among repetitions of the same experiments, the standard deviation of the obtained total execution times, i.e. σ , is also provided in the table.

	<i>Step-Down</i>		<i>Step-Up</i>		<i>Mixed</i>	
	μ	$\% \sigma$	μ	$\% \sigma$	μ	$\% \sigma$
STATIC	1543.27	0.24%	1005.58	0.62%	1432.10	0.62%
WF	1483.58	3.27%	1025.18	8.97%	1203.00	1.55%
AWF	1415.35	4.36%	988.77	0.72%	1197.37	1.15%
AF	1521.38	0.85%	1213.52	3.05%	1746.67	15.23%
ATLS	1476.66	0.53%	946.25	1.13%	1049.53	1.46%

Table 5: Average μ and percentage of standard deviation $\% \sigma$ of the total execution times obtained during the experiments

We observe that for the step-down experiment the best result is obtained by the scheme AWF, followed by ATLS with a result 4.33% worse. As expected in a dynamic environment, the worst result is obtained by STATIC, performing a 9.04% slower. For the step-up case, the best scheme is ATLS followed by AWF which performs a 4.43% slower. Finally, for the mixed environment, the best result is obtained again by ATLS followed by AWF, but this time a 14.09% slower for this experiment. To better understand the differences obtained in these results we can analyze the information provided with Figure 52.

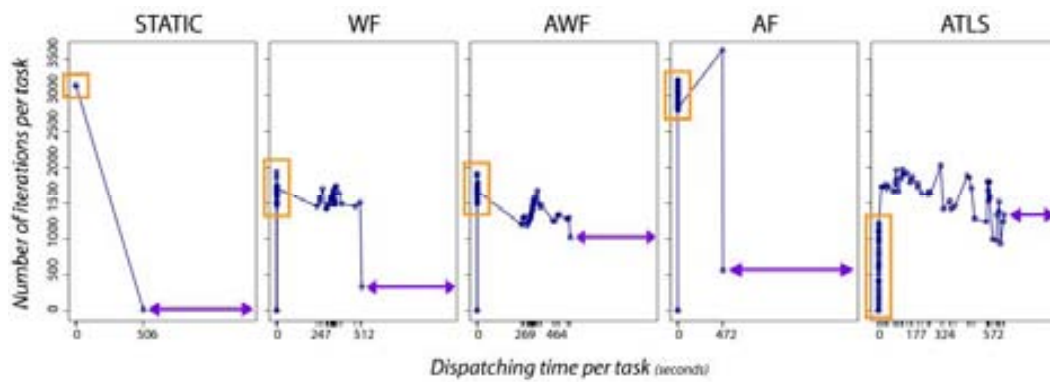


Figure 52: Number of iterations per task at dispatching time in a dynamic environment

Marked with an orange box it is indicated the first main batch of tasks. By observing the height of the box and its vertical position we can guess how eager is each scheme. STATIC dispatches almost all the iterations at the first round with equal sizes (there is one task left due to rounding adjustments). AF also schedules most of the tasks too early. At the other side, ATLS is more conservative at the beginning and schedules considerably less iterations per task with a wider distribution of the number of tasks. In the middle we found WF and AWF with similar distributions.

The purple arrows give us another indication. Its length shows the proportion of time of each individual experiment where the scheduler is just waiting, since all the iterations have already been dispatched. This is important since the sooner the scheduler runs out of iterations the sooner it will lose its ability to react against environmental changes. For this indication, it can be observed that ATLS is the scheme that more delays the dispatch of the last tasks, resulting in a better ability to adapt to unexpected changes.

Finally, the green ellipses shown in Figure 53 indicate the number of tasks finalizing close to the end. In a perfect situation all the scheduled tasks should finalize exactly at

the same final time so we ensure all the available nodes are effectively used. ATLS shows the highest density of tasks finalizing close to the end while STATIC and AF waste a considerable amount of time waiting for the last task.

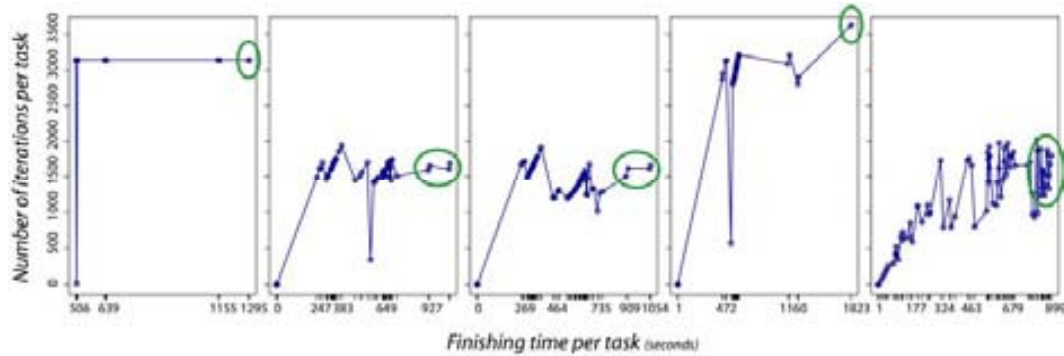


Figure 53: Number of iterations per task at finishing time in a dynamic environment

As the contents of this section have showed, the proposed scheduling scheme, ATLS, while providing satisfactory results in environments with none or low frequency of changes, is the better suited, compared with other schemes analyzed, for dynamic environments with a high frequency of changes. It is also worth to mention that the scheme AWF, although not as well suited as ATLS for highly changing environments, it also provides very good results in dynamic environments, in some cases even a slightly better than ATLS. Nevertheless, the experimental results show that ATLS provides good performance figures in all the scenarios evaluated, outperforming the others in dynamic environments like the one used at the last experiment due to the reasons exposed in this section.

5.5. Concluding Remarks

In this chapter we have evaluated all the proposals included in this work. The methods proposed for parallelizing existing bioinformatics applications programmed in R and based in parallel loops have been successfully parallelized and its completion times minimized using the available resources.

Next, well-known scheduling schemes have been evaluated. As the experimental results have confirmed, the performance obtained using one or another scheme is highly dependent on the characteristics of the running environment being used. In static and homogeneous environments, classical methods like the STATIC scheme provide the best results. In static and heterogeneous environments, factoring based schemes have shown very good results. Finally in dynamic environments, adaptive schemes, suitable for changing environments have been evaluated in different test environments. As result of the strengths and weakness of these schemes evaluated in

several scenarios, a new scheduling scheme for parallel loops, called ATLS, has been presented.

As the experiments show, our proposal provides in average better results than previous well-known contributions in dynamic environments of non-dedicated computers. The design of the scheme, since it is only based in the information obtained monitoring the turnaround completion time of each task scheduled to each worker, allows its implementation in virtually any running platform. However, still better performance can be obtained. In case an external source of information from where insights about future events was available (e.g. the queue of a job scheduling system can provide a hint of the utilization of shared elements like the network), it would be possible to react beforehand against incoming changes. Additionally, as it has been observed, before a distributed computation finishes there is a fraction of time where all the iterations have been scheduled and slower nodes must be awaited. Although the objective is to minimize this fraction of time, due to unexpected changes, it is almost always present, hence a rescheduling component can be added to take advantage of the idle nodes which eventually can process the pending iterations before the slower nodes finish. The same rescheduling component can also be used to provide fault tolerance to the system and handle faulty nodes. These and other future work will be discussed in the last chapter.

Chapter 6

Conclusions and Future Work

6.1. Conclusions

In this thesis we have analyzed several problems found by users without previous experience in parallel computing and how the barriers and obstacles that prevent the utilization of HPC solutions can be avoided or mitigated. In particular we considered the execution of parallel loops using non-dedicated environments with R, a programming language extensively used by the bioinformatics community.

In order to allow users without proper knowledge and skills, and with severe limitations of time and resources, adapt their applications and enable the utilization of accessible computing resources, several goals were defined. Along the chapters of this dissertation we have described and evaluated our proposals, which have been proved to be correct taking into account the results obtained within the experiments performed. Next we summarize these contributions, indicating the publications accepted in peer reviewed journals and conferences.

First new methods to transform existing sequential applications based in parallel loops and programmed in R were proposed. With a minimum knowledge of parallel programming, we have been able to transform sequential R programs into parallel versions, ready to be run in a parallel system. We also described the mechanism used to enable the execution of these parallel applications, again with a minimum effort from the user, in symmetric multiprocessing computers like for example multicore

desktop computers. The outcome was a first prototype called R/parallel which allow us to evaluate our proposal in real systems. The papers accepted describing the methods and mechanisms developed are:

- G. Vera, R. Jansen, and R. Suppi, "R/parallel - speeding up bioinformatics analysis with R," *BMC Bioinformatics*, vol. 9, p. 390, 2008.
- G. Vera and R. Suppi, "Towards the evolution of legacy applications to multicore systems - experiences parallelizing R," in *Proceedings of the First International Conference on Bioinformatics (BIOINFORMATICS 2010)*, Valencia, Spain, 2010, pp. 250-256.

Next, as an evolution of previous work, new methods for R to integrate multiple processing units of different types, mainly using existing non-dedicated computing resources not ready for their utilization were proposed. Also further extensions of R/parallel, to support the distributed computation in R were described and evaluated. The papers accepted describing the methods, mechanisms and results are:

- G. Vera and R. Suppi, "Integrated parallel computing for R in heterogeneous best-effort environments," in *3rd Palestinian International Conference on Computer and Information Technology (PICCIT 2010)*, Hebron, Palestina, 2010, p. In press.
- G. Vera and R. Suppi, "Integration of heterogeneous and non-dedicated environments for R," in *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010)*, 2010, p. In press.

Finally, and taking advantage of the experience and knowledge obtained with previous contributions, a new self-scheduling method called ATLS was described and evaluated. The results showed that even integrating non-dedicated resources of heterogeneous types in changing environments, with a minimum information about the running environment it is possible to provide better performance results than those obtained with previous well-known scheduling schemes suitable for execution of parallel loops. The paper accepted describing and evaluating the results obtained by this scheme is:

- G. Vera and R. Suppi, "ATLS – A parallel loop scheduling scheme for dynamic environments," in *International Conference on Computational Science (ICCS 2010)*, Amsterdam, The Netherlands, 2010, p. In press.

The conclusion of this work is that even in the presence of technical limitations and lack of knowledge or time by potential users it is possible to develop methods and implement software mechanisms that, with a minimal information about their environment and a high degree of uncertainty about the quantity and quality of the available resources, and with a minimal effort from the user, are able to conceal the complexity inherent in classical parallel computing systems.

By doing so, with our contributions we have proved that it is possible to provide feasible solutions to adapt existing sequential applications, in particular parallel loop based ones programmed in R, and make an efficient use of the available computing resources, thus providing a new solution for users that previously were unable to embrace parallel computing solutions to fulfill their needs.

6.2. Future Work

Once a research line is started it does not take too much time, as we acquire more knowledge about the problem being explored, to discover that multiple alternative research paths can be followed. Our case is not an exception and several aspects, at the end of this work are open and available for further research. Next we enumerate some of the most important lines where further research can be done:

- Extension of parallel loops execution with dependencies. Although parallel loops are present in many scientific algorithms, it can be argued that also loops with dependencies should be considered. There exists a considerable amount of literature and contributions in this respect that could be reviewed to attempt their adaptation to the running environments considered in our work. Besides, we have left to the user the responsibility of ensuring that his or her loop is dependence free, what, depending on the user, cannot be guaranteed. By adding at least dependence analysis checking we could avoid a misuse of our proposal.
- Extension to other parallel structures. There exist many other situations where parallelism could be extracted. The easier ones to consider are task parallelism and hierarquical loops.
- Extensions to other resources providers. In this work we have tested a few of the many sources of additional computing resources that can be integrated with our solution. An emerging an interesting option could be to consider Cloud systems.

- Extension to fine grained data parallelism. In our current implementation the whole state of the ongoing calculation is stored and distributed among the workers. That implies that all the data must fit within the client and workers' memory. To enable the handling of workloads bigger than the client memory, mechanisms to delay the load of data and distribute it to workers could be researched and included in R/parallel.

Finally, an important line for future work is the incorporation of support for fault tolerance. We cannot obviate the importance of this mechanism when using non-dedicated environments. It is possible that even though trying to assign the right chunk size to each working node based in its current performance ratio, due to unexpected changes on its behavior, it may take by far too much time to finalize its assigned task. Maybe, due to failure or because the computer has been switched off, this time can be extend *ad infinitum*. In such situations, when a task assignment fails due to several reasons, the failed iterations have to be rescheduled. To handle this problem, a queueing subsystem as it is illustrated in Figure 54 is proposed.

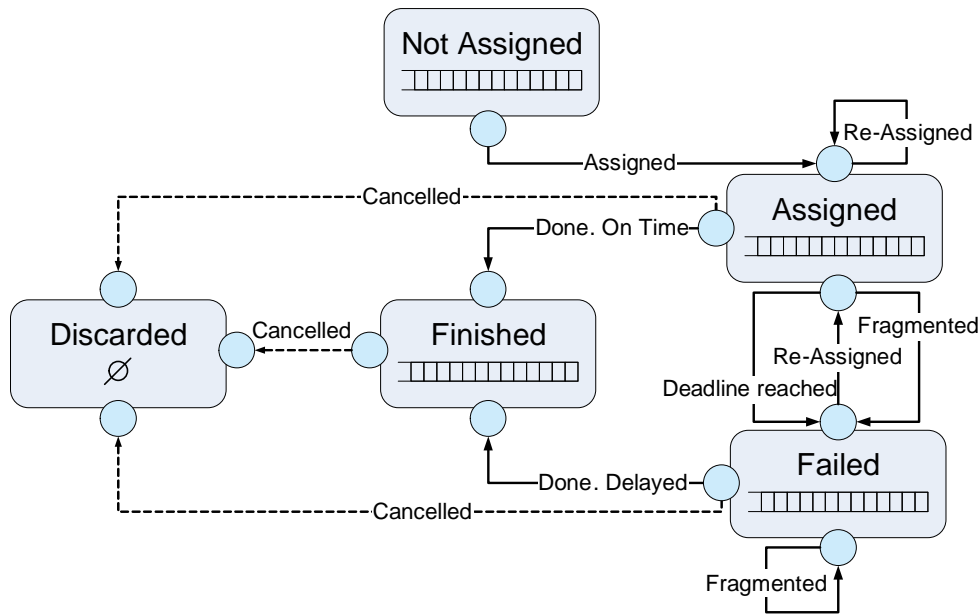


Figure 54: Queue subsystem to support fault tolerance

Initially the workload with all the iterations could be maintained in the *Not Assigned* queue. Every time a worker node request a task, a subset of continuous iterations of size K_i can be retrieved from this queue and a task, with the defined number of iterations, or less if there are no more available, is dispatched to the invoking node. To keep track of this event, the information relative to this assignation could be maintained in the *Assigned* queue, and the *Not Assigned* queue updated. Later, if the

worker node has successfully finalized its assignment before the estimated deadline, the information can be moved from the *Assigned* to the *Finished* queue. The main objective of the *Finished* queue is to maintain the information about all the finalized tasks to later, in a final step, reduce the partial results returned by each worker to build the final values, applying the same order to the reduction operations as it would happened in a sequential execution. However, it is also possible that the deadline for a task is reached.

In that case, the information about that task is moved from the *Assigned* queue to the *Failed* queue. If the task is finished once it has reached the *Failed* queue, its information is moved to the *Finished* queue. In the final stages of the computation, it can happen that a few task assignments have not yet finished. It doesn't matter if their records are maintained in the *Assigned* queue because it was expected this time or in the *Failed* queue because it is being delayed. The fact is that we can end up with idle working nodes while waiting for other tasks that maybe will never finish. The rational option here, when there are no more pending iterations for assignment in the *Not Assigned* queue, is to reschedule iterations to other working nodes. The first iterations that should be rescheduled are the ones contained in the *Failed* queue since they have already shown symptoms of irregular behavior. In this case, if the whole task size fits the defined K_i for the invoking worker, it is re-assigned and the task information updated and moved back from the *Failed* queue to the *Assigned* queue. But it can happen, specially at the last steps of the computation that the task size is larger than K_i . This can be quite frequent since scheduling schemes are usually designed to behave in that way to reduce load imbalance. In such case, the selected task has to be split in two. A new task of size K_i is assigned and its information annotated in *Assigned*, and another task, with the remaining iterations is introduced in the *Failed* queue. The original task is still maintained in *Failed* since it can happen that, although more delayed than expected, the task arrives before the re-assigned ones finishes. For the cases that there are no pending tasks in the *Not Assigned* or *Failed* queues, tasks from the *Assigned* queue can be re-assigned in the same way defined for the *Failed* queue. The idea in that case, more than preventing a feasible failure, in a speculative approach, is to duplicate the work assignment and consider the results of the faster worker node. The only inconvenient produced by re-scheduling tasks and the generated fragmentation is that once any of these tasks finishes, the other related tasks (full or partial duplications) must be discarded. It can even happen that a duplicated task with a partial copy, besides of the ones in *Assigned* and *Failed* queues, is already in the *Finished* queue and has to be discarded since its originating task has been completed and it contains more iterations than the previously finished.

Working with a queueing system for tracking the tasks assignments introduces some extra overhead. However, usually the worst cases appear at the final stages, when several working nodes are delayed and its assigned tasks have to be re-scheduled.

This can produce too much fragmentation and every time an original task arrives, the related copies have to be discarded and removed from the queues. Nevertheless, in this kind of situations, without an equivalent subsystem like the proposed here, the time would be otherwise spent just waiting. In that sense it can be understood that the extra effort can be profitable and therefore should be considered for future work.

Bibliography

- [CB+05] A.J.Chakravarti, G.Baumgartner, and M.Lauria, "The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 35, pp. 1-12, 2005.
- [AV+08] R. Alberts, G. Vera, and R. C. Jansen, "affyGG: computational protocols for genetical genomics with Affymetrix arrays," *Bioinformatics*, vol. 24, pp. 433–434, 2008.
- [AC+10] E. Allen, D. Chase, and J. Hallet. *The Fortress language specification*. 2010. Available: <http://labs.oracle.com/projects/plrg/>
- [Amd67] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," presented at the Proceedings of the April 18-20, 1967, spring joint computer conference, Atlantic City, New Jersey, 1967.
- [And04] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID '04)*, Washington, DC, USA, 2004, pp. 4-10.
- [AC+02] D. P. Anderson, J. Cobb, *et al.*, "SETI@home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, pp. 56-61, 2002.
- [ANSI94] ANSI Working Committee, "Parallel Processing Model for High Level Programming Languages, ANSI X3H5," Document Number X3H5/94 SD2, Apr 1994.
- [AB+06] K. Asanovic, R. Bodik, *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California, Berkeley Report Num: UCB/EECS-2006-183, December 18 2006.

- [BS95] D. Bakken and R. Schilchting, "Supporting Fault-Tolerant Parallel Programming in Linda," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, pp. 287-302, March 1995.
- [BC+09] I. Banicescu, F. M. Ciorba, and R. L. Cariño, "Towards the robustness of dynamic loop scheduling on large-scale heterogeneous distributed systems," in *Eighth International Symposium on Parallel and Distributed Computing (ISPDC'09)*, 2009, pp. 129-132.
- [BV02] I. Banicescu and V. Velusamy, "Load balancing highly irregular computations with the adaptive factoring," in *Proceedings International Parallel and Distributed Processing Symposium*, 2002, pp. 87–98.
- [BV+03] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring," *Cluster Computing*, vol. 6, pp. 215–226, 2003.
- [BD+95] A. Baratloo, P. Dasgupta, and Z. Kedem, "Calypso: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms," in *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, 1995.
- [BC+09] R. D. Bjornson, N. J. Carriero, *et al.*, "NetWorkSpace: A Coordination System for High-Productivity Environments" *International Journal of Parallel Programming*, vol. 37, pp. 106-125, 2009.
- [Bod09] D. Bode. *Rsg: Interface to the SGE Queuing System*. 2009. Available: <http://cran.r-project.org/web/packages/Rsg/index.html>
- [BV+08] M. J. Bridges, N. Vachharajani, *et al.*, "Revisiting the sequential programming model for the multicore era," *IEEE Micro*, vol. 28, pp. 12-20, 2008.
- [Bri96] P. Briggs, "Automatic parallelization," *SIGPLAN Not.*, vol. 31, pp. 11-15, 1996.
- [BW+03] K. W. Broman, H. Wu, *et al.*, "R/qtl: QTL mapping in experimental crosses," *Bioinformatics*, vol. 19, pp. 889-890, 2003.
- [Bub08] A. v. Bubnoff, "Next-generation sequencing: the race is on," *Cell*, vol. 132, pp. 721-723, 2008.
- [BD+94] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," in *Proceedings of Supercomputing Symposium*, 1994, pp. 379–386.
- [Buy99] R. Buyya, *High Performance Cluster Computing: Programming and Applications, volume 2*. New Jersey, USA: Prentice Hall PTR, 1999.
- [BW+05] L. Bystrykh, E. Weersing, *et al.*, "Uncovering regulatory pathways that affect hematopoietic stem cell function using 'genetical genomics'," *Nature Genetics*, vol. 37, pp. 225–232, 2005.
- [CC+04] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The Cascade High Productivity Language," in *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, 2004, pp. 52-60.
- [CG+05] P. C. Carvalho, R. V. Glória, *et al.*, "Squid – a simple bioinformatics grid," *BMC Bioinformatics*, vol. 6, p. 197, 2005.
- [CR+08] F. M. Ciorba, I. Riakiotakis, *et al.*, "Enhancing self-scheduling algorithms via synchronization and weighting," *Journal of Parallel and Distributed Computing*, vol. 68, pp. 246-264, 2008.

- [CG+97] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1997.
- [Cyt86] R. Cytron, "Doacross: Beyond vectorization for multiprocessors," in *International Conference on Parallel Processing (ICPP'86)*, 1986, pp. 836-844.
- [CC+03] A. Chien, B. Calder, *et al.*, "Entropy: architecture and performance of an enterprise desktop grid system," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 597-610, 2003.
- [CK+07] S. Choi, H. Kim, *et al.*, "Characterizing and Classifying Desktop Grid," in *IEEE Cluster Computing and the Grid (CCGrid'07)*, 2007, pp. 743-748.
- [CA+01] A. T. Chronopoulos, R. Andonie, *et al.*, "A Class of Loop Self-Scheduling for Heterogeneous Clusters," in *Proceedings of the 3rd IEEE International Conference on Cluster Computing (CLUSTER 2001)*, Newport Beach, California, USA, 2001, pp. 282-291.
- [DM98] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computing in Science and Engineering*, vol. 5, pp. 46 - 55, 1998.
- [DR+06] J. Díaz, S. Reyes, *et al.*, "A Quadratic Self-Scheduling Algorithm for Heterogeneous Distributed Computing Systems," in *Proceedings of the 5th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar'06)*, 2006, pp. 1-8.
- [DR+09] J. Díaz, S. Reyes, *et al.*, "Derivation of self-scheduling algorithms for heterogeneous distributed computer systems: Applications to internet-based grids of computers," *Future Generation Computer Systems*, vol. 25, pp. 617-626, 2009.
- [DC+08] Y. Dong, J. Chen, *et al.*, "Energy-Oriented OpenMP Parallel Loop Scheduling," in *International Conference on Parallel and Distributed Processing with Applications (ISPA2008)*, 2008, pp. 162-169.
- [DF+03] J. Dongarra, I. Foster, *et al.*, *Sourcebook of parallel computing*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2003.
- [DL06] J. Dongarra and A. Lastovetsky, "An overview of heterogeneous high performance and grid computing," in *Engineering the Grid*, 2006.
- [ES+04] K. Ebcioglu, V. Saraswat, and V. Sarkar, "X10: Programming for hierarchical parallelism and non-uniform data access," in *International Workshop on Language Runtimes, (OOPSLA 2004)*, 2004.
- [FR+97] D. Feitelson, L. Rudolph, *et al.*, "Theory and Practice in Parallel Job Scheduling," in *Proceedings of 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, 1997, pp. 1-34.
- [Fly66] M. Flynn, "Very high speed computing systems," in *Proceedings of the IEEE*, 1966, pp. 1901-1909.
- [FK97] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, pp. 115-128, Summer 1997.
- [FK99] I. Foster and C. Kesselman, *The grid: blueprint for a new computing infrastructure*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 1999.

- [FK03] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*: Morgan Kaufmann Publishers Inc., 2003.
- [FS+07] J. Fu, M. Swertz, *et al.*, "MetaNetwork: a computational protocol for the genetic study of metabolic networks," *Nature Protocols*, vol. 2, pp. 685-694, 2007.
- [GF+04] E. Gabriel, G. E. Fagg, *et al.*, "OpenMPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings of 11th European PVM/MPI Users Group Meeting*, 2004, pp. 97–104.
- [Gel95] D. Gelernter, "Parallel Programming in Linda," Department of Computer Science, Yale University, Technical Report 359, January 1985.
- [GC+04] R. Gentleman, V. Carey, *et al.*, "Bioconductor: open software development for computational biology and bioinformatics," *Genome Biology*, vol. 5, p. R80, 2004.
- [Gen01] W. Gentzsch, "Sun Grid Engine: Towards Creating a Compute Power Grid," in *First IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, 2001, pp. 35-36.
- [GNU07] GNU Software Foundation. *GNU general public licence*. 2007. Available: <http://www.gnu.org/licenses/gpl.html>
- [Gro09] D. Grose. *High throughput distributed computing using R : the multiR package*. 2009. Available: <http://e-science.lancs.ac.uk/multiR>
- [Han93] P. B. Hansen, "Model Programs for Computational Science: A Programming Methodology for Multicomputers," *Concurrency: Practice and Experience*, vol. 5, pp. 407-423, 1993.
- [HT96] R. Henderson and D. Tweten, "Portable Batch System: External reference specification.," NASA Ames Research Center, Technical Report, 1996.
- [HP07] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Francisco: Morgan Kauffman, 2007.
- [HH+08] J. Hill, M. Hambley, *et al.*, "SPRINT: A new parallel framework for R," *BMC Bioinformatics*, vol. 9, p. 558, 2008.
- [HT+08] W. Huber, J. Toedling, and M. Ritchie. *tilingArray – Analysis of high-density oligonucleotide tiling arrays* 2008. Available: <http://bioconductor.org/packages/2.2/bioc/html/tilingArray.html>
- [HS+96] S. F. Hummel, J. Schmidt, *et al.*, "Load-sharing in heterogeneous systems via weighted factoring," in *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy 1996, pp. 318 - 328.
- [HS+92] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: a method for scheduling parallel loops," *Communications of the ACM*, vol. 35, pp. 90-101, August 1992.
- [IG96] R. Ihaka and R. Gentleman, "R: A Language for Data Analysis and Graphics," *Journal of Computational and Graphical Statistics*, vol. 5, pp. 299-314, September 1996.
- [Jam08] F. Jamitzky. *ROMP: OpenMP binding for GNU R*. 2008. Available: <http://code.google.com/p/romp/>
- [Jan93] R. C. Jansen, "Interval Mapping of Multiple Quantitative Trait Loci," *Genetics* vol. 135, pp. 205-211, 1993.
- [KN+05] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos, "An Efficient Approach for Self-Scheduling Parallel Loops on Multiprogrammed Parallel

- Computers," in *The 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, 2005, pp. 441-449.
- [KN+06] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos, "History-aware Self-Scheduling," in *Proceedings of the 2006 International Conference on Parallel Processing (ICPP'06)*, 2006, pp. 185 - 192
- [KN+09] A. Kejariwal, A. Nicolau, *et al.*, "Efficient Scheduling of Nested Parallel Loops on Multi-Core Systems," in *International Conference on Parallel Processing (ICPP'09)*, 2009, pp. 74-83.
- [KP+09] J. Knaus, C. Porzelius, *et al.*, "Easier Parallel Computing in R with snowfall and sfCluster," *The R Journal*, vol. 1, pp. 54-59, May 2009.
- [KW85] C. P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1001-1016, 1985.
- [Lam74] L. Lamport, "The parallel execution of DO loops," *Communications of the ACM*, vol. 17, pp. 83-93, Feb. 1974.
- [LR02] N. Li and A. J. Rossini. *rpvm: R interface to PVM (Parallel Virtual Machine)*. 2002. Available: <http://cran.r-project.org/web/packages/rpvm/index.html>
- [LJ+08] P. Li, Q. Ji, *et al.*, "An Adaptive Chunk Self-Scheduling Scheme on Service Grid," in *IEEE Asia-Pacific Services Computing Conference (APSCC'08)*, 2008, pp. 39-44.
- [LL+88] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - a hunter of idle workstations," in *8th International Conference on Distributed Computing Systems*, 1988, pp. 104-111.
- [LS+09] D. Liu, Z. Shao, *et al.*, "Optimal Loop Parallelization for Maximizing Iteration-Level Parallelism," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*, 2009, pp. 67-76.
- [Lov93] D. B. Loveman, "High performance Fortran," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, pp. 25-42, 1993.
- [Lun85] S. F. Lundstrom, "A decentralized control, highly concurrent multiprocessor," in *12th Annual international Symposium on Computer Architecture* Boston, Massachusetts, United States, 1985, pp. 145-151.
- [ML+07] X. Ma, J. Li, and N. F. Samatova, "Automatic Parallelization of Scripting Languages: Toward Transparent Desktop Parallel Computing," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA, 2007, pp. 298-304.
- [Mar06] E. Mardis, "Anticipating the \$1,000 genome," *Genome Biology*, vol. 7, p. 112, 2006.
- [MM+05] M. Margulies, M. Egholm, *et al.*, "Genome sequencing in microfabricated high-density picolitre reactors," *Nature*, vol. 437, pp. 376-380, 2005.
- [Mar09] A. Marletta. *Cpu Usage Limiter for Linux*. 2009. Available: <http://cpulimit.sourceforge.net/>
- [Met10] M. L. Metzker, "Sequencing technologies - the next generation," *Nature Reviews Genetics*, vol. 11, pp. 31-46, 2010.
- [Moo65] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, p. 4, 1965.

- [PK+80] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Transactions on Computers*, vol. 29, pp. 763-776, 1980.
- [PW86] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, pp. 1184-1201, December 1986.
- [Pla94] O. Plata and F. F. Rivera, "Combining static and dynamic scheduling on distributed-memory multiprocessors," in *Proceedings of the 8th international conference on Supercomputing (ICS'94)*, Manchester, England, 1994, pp. 186-195.
- [Pol88] C. D. Polychronopoulos, *Parallel Programming and Compilers*: Kluwer Academic Publishers, 1988.
- [PK87] C. D. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, vol. C-36, pp. 1425-1439, December 1987.
- [PK+06] J. Pukacki, M. Kosiedowski, *et al.*, "Programming Grid Applications with Gridge," *Computational Methods in Science and Technology*, vol. 12, pp. 47-68, 2006.
- [RD08] R Development Core Team, *R Language Definition*. Vienna, Austria: The R Foundation for Statistical Computing, 2008.
- [Sar89] V. Sarkar, "Determining average program execution times and their variance," in *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, 1989, pp. 298-312.
- [Sar98] L. F. G. Sarmenta, "Bayanihan: Web-Based Volunteer Computing Using Java," in *Proceedings of the Second International Conference on Worldwide Computing and Its Applications*, 1998, pp. 444-461.
- [NWS09] Scientific Computing Associates. *NetWorkSpaces for R*. 2009. Available: <http://nws-r.sourceforge.net>
- [SM+09] M. Schmidberger, M. Morgan, and D. Eddelbuettel, "State of the Art in Parallel Computing with R," *Journal of Statistical Software*, vol. 31, 2009.
- [Sei85] C. L. Seitz, "The cosmic cube," *Communications of the ACM*, vol. 28, pp. 22-33, 1985.
- [SY+07] W.-C. Shih, C.-T. Yang, and S.-S. Tseng, "A performance-based parallel loop scheduling on grid environments," *Journal of Supercomputing*, vol. 41, pp. 247-267, 2007.
- [SZ+09] J. Shirako, J. Zhao, *et al.*, "Chunking Parallel Loops in the Presence of Synchronization," in *Proceedings of the 23rd international conference on Supercomputing (ICS'09)*, Yorktown Heights, New York, USA, 2009, pp. 181-192.
- [SC92] L. Smarr and C. E. Catlett, "Metacomputing," *Communications of the ACM*, vol. 35, pp. 44-52, 1992.
- [SW+07] C. Smith, G. Warnes, *et al.* *Rlsf: Interface to the LSF Queuing System*. 2007. Available: <http://cran.r-project.org/web/packages/Rlsf/index.html>
- [Sun90] V. S. Sunderam, "PVM: a framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315-339, 1990.
- [SL05] H. Sutter and J. Larus, "Software and the Concurrency Revolution," *Queue*, vol. 3, pp. 54-62, 2005.

- [TY86] P. Tang and P. Yew, "Processor self-scheduling for multiple-nested loops," in *International Conference on Parallel Processing (ICPP'86)*, 1986, pp. 528-535.
- [MPI93] The MPI Forum, "MPI: A Message Passing Interface," in *Proceedings of Supercomputing 93*, 1993, pp. 878-883.
- [PCF91] The Parallel Computing Forum, "PCF parallel Fortran extensions," *SIGPLAN Fortran Forum*, vol. 10, pp. 1-57, 1991.
- [Pr102] The Perl Foundation. *Perl 5.8.0 release announcement*. 2002. Available: <http://dev.perl.org/perl5/news/2002/07/18/580ann/>
- [RF10] The R Foundation. *The Comprehensive R Archive Network*. 2010. Available: <http://cran.r-project.org>
- [TR+09] L. Tierney, A. J. Rossini, and N. Li, "Snow : A Parallel Computing Framework for the R System " *International Journal of Parallel Programming*, vol. 37, pp. 78-90, 2009.
- [Tre01] O. Trelles, "On the parallelisation of bioinformatics applications," *Briefings in Bioinformatics*, vol. 2, pp. 181-194, 2001.
- [TN93] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 87-98, 1993.
- [Urb09] S. Urbanek. *multicore: Parallel processing of R code on machines with multiple cores or CPUs*. 2009. Available: <http://cran.r-project.org/web/packages/multicore/index.html>
- [VS08] G. Vera, R. Jansen, and R. Suppi, "R/parallel - speeding up bioinformatics analysis with R," *BMC Bioinformatics*, vol. 9, p. 390, 2008.
- [VS10d] G. Vera and R. Suppi, "ATLS – A parallel loop scheduling scheme for dynamic environments," in *International Conference on Computational Science (ICCS 2010)*, Amsterdam, The Netherlands, 2010, p. In press.
- [VS10b] G. Vera and R. Suppi, "Integrated parallel computing for R in heterogeneous best-effort environments," in *3rd Palestinian International Conference on Computer and Information Technology (PICCIT 2010)*, Hebron, Palestina, 2010, p. In press.
- [VS10c] G. Vera and R. Suppi, "Integration of heterogeneous and non-dedicated environments for R," in *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010)*, 2010, p. In press.
- [VS10a] G. Vera and R. Suppi, "Towards the evolution of legacy applications to multicore systems - Experiences parallelizing R," in *First International Conference on Bioinformatics, BIOINFORMATICS 2010*, Valencia, Spain, 2010, pp. 250-256.
- [WS+07] D. Wegener, T. Sengstag, *et al.*, "GridR: An R-based grid-enabled tool for data analysis in ACGT clinico-genomics trials," in *IEEE International Conference on e-Science and Grid Computing*, 2007, pp. 228-235.
- [YC03] C.-T. Yang and S.-C. Chang, "A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters," in *International Conference on Computational Science (ICCS'03)*, 2003, pp. 1079-1088.
- [YC+04] C.-T. Yang, K.-W. Cheng, and K.-C. Li, "An Efficient Parallel Loop Self-Scheduling on Grid Environments," in *IFIP International Conference on Network and Parallel Computing (NPC 2004)*, 2004, pp. 92-100.

- [YC09] C.-T. Yang and L.-H. Cheng, "A Performance-based Dynamic Loop Partitioning on Grid Computing Environments," in *11th IEEE International Conference on High Performance Computing and Communications (HPCC'09)*, 2009, pp. 512-519.
- [YS+06] C.-T. Yang, W.-C. Shih, and S.-S. Tseng, "A Dynamic Partitioning Self-Scheduling Scheme for Parallel Loops on Heterogeneous Clusters," in *International Conference on Computational Science (ICCS'06)*, 2006, pp. 810-813.
- [Yu02] H. Yu. *Rmpi: Interface (wrapper) to MPI (Message-Passing Interface)*. 2002. Available: <http://cran.r-project.org/web/packages/Rmpi/index.html>
- [ZZ+93] S. Zhou, X. Zheng, *et al.*, "Utopia: A load sharing facility for large, heterogeneous distributed computer systems," *Software: Practice and Experience*, vol. 23, pp. 1305-1336, 1993.