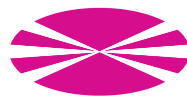

Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

A parallel tool for the identification of differentially methylated regions in genomic analyses

Autor: Alejandro Fernández Fraga
Director: Jorge González Domínguez
Director: Juan Touriño Domínguez

A Coruña, xuño de 2021.

To my family, for helping me in every step of the way

Acknowledgements

To my family, for trusting me and always being by my side, no matter the situation.

To my friends, for turning this time into the best years of my life and for always meeting me with a coffee and a smile when I most needed them.

To my mentors, Jorge and Juan, for believing in me to carry out this thesis and for always meeting me with great advice and uplifting words, even in the difficult circumstances we had to face due to the COVID-19.

Abstract

Parallel and High Performance Computing (HPC) has gained attention in the last years as a mean to accelerate several kind of computationally expensive applications. Bioinformatics is one of the fields that benefits from this acceleration, since it demands a high computational power to analyse the biological data obtained from experiments. Due to the cost reductions related to obtaining biological data, more and more tools are able to extract conclusions out of this data are coming out, with capabilities to visualize, analyse and extract, but they come with high execution times and computational requirements.

In particular, methylation analysis is one of the bioinformatics fields that fits into this description, since this process is associated to different biological functions, and abnormal methylation levels can indicate the presence of certain diseases. For instance, the existence of regions with different methylation levels is a common characteristic for several types of cancer. Therefore, discovering differentially methylated regions is an important research field in genomics, as it can help to anticipate the risk to suffer from some diseases. Nevertheless, the high computational cost associated to the discovery of differentially methylated regions prevents its application to large-scale datasets. Hence, a much faster application is required to further progress in this research field.

During this bachelor's thesis an optimized version of RADMeth, a tool for the identification of differentially methylated regions based on beta-binomial regression, has been developed and arranged to take advantage of the features of HPC systems. The different optimization techniques implemented were developed by applying a workload distribution among the processing elements using domain decomposition and by keeping in mind the typical architecture of HPC systems composed of several nodes (each of the nodes being a multicore system) so the novel tool takes advantage of both levels by a hybrid MPI/OpenMP implementation.

This way execution time was significantly reduced, Performance was tested on a cluster composed of 16 nodes, with 64 GB of memory and 16 cores per node (256 nodes in total). Obtained results were very satisfactory, obtaining speedups up to 194x.

Resumen

La computación paralela y de altas prestaciones (HPC por sus siglas en inglés) está ganando atención en los últimos años como medio para acelerar varios tipos de aplicaciones con un coste computacional elevado. Una de las disciplinas que se beneficia de esto es la bioinformática, que requiere una gran potencia computacional para analizar los datos de experimentos biológicos. Debido a la reducción de costes asociados a la obtención de datos biológicos, más

y más herramientas capaces de visualizar, analizar y extraer conclusiones de estos datos salen a la luz, pero vienen con elevados tiempos de ejecución y requisitos computacionales.

Concretamente uno de los campos que cumple estas características es el análisis de la metilación, ya que este proceso está asociado con diferentes funciones biológicas y niveles raros de metilación pueden ser un indicativo de la presencia de enfermedades. Por ejemplo, la existencia de regiones con diferentes niveles de metilación es una característica presente en muchos tipos de cáncer. Por tanto, el descubrimiento de regiones con diferentes niveles de metilación es un importante campo de investigación. Sin embargo, llevar a cabo este análisis sobre grandes cantidades de datos es un proceso computacionalmente costoso, por lo que se requiere de una herramienta mucho más rápida para prograsar en este campo de investigación.

En este trabajo se ha desarrollado una optimización de RADMeth, una herramienta que identifica regiones diferencialmente metiladas basada en regresión beta-binomial, adaptándola para aprovechar las ventajas de los sistemas HPC. Las paralelizaciones implementadas fueron desarrolladas aplicando una distribución de la carga de trabajo entre los elementos de procesado usando descomposición de dominio y teniendo en cuenta que los sistemas HPC suelen ser sistemas multinodo con nodos multinucleo, por lo que la nueva herramienta aprovecha las ventajas de ambos con una aproximación híbrida basada en MPI Y OpenMP.

De esta forma se consiguió reducir el tiempo de cómputo de forma significativa. Las pruebas de rendimiento se realizaron en un cluster, con 16 nodos y 64 GB de memoria y 16 núcleos por nodo (256 núcleos en total). Los resultados obtenidos fueron muy satisfactorios, consiguiendo aceleraciones de hasta 194x.

Keywords:

- Differential Methylation
- Whole Genome Bisulfite Sequencing
- Beta-Binomial Regression
- Bioinformatics
- MPI
- OpenMP
- High Performance Computing
- Parallel Computing

Palabras clave:

- Metilación diferencial
- Secuenciación de Bisulfito del Genoma Completo
- Regresión Beta-Binomial
- Bioinformática
- MPI
- OpenMP
- Computación de Altas Prestaciones
- Computación Paralela

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Document Structure . . . 2

2 Identification of differentially methylated regions 3
2.1 Basic concepts . . . 3
2.2 RADMeth . . . 5
2.2.1 Overview . . . 5
2.2.2 Input data . . . 7
2.2.3 Output data . . . 8
2.2.4 Computation phase . . . 9
2.3 Other methods . . . 12
2.3.1 Fisher's Exact Test and Hidden Markov Models . . . 13
2.3.2 Algorithms based on smoothing . . . 13
2.3.3 Algorithms based on beta-binomial distributions . . . 13
2.3.4 Algorithms based on regression . . . 14
2.3.5 Parallel algorithms . . . 15
2.3.6 Conclusions . . . 15

3 Design and Implementation 17
3.1 Target architecture . . . 17
3.1.1 MPI . . . 19
3.1.2 OpenMP . . . 22
3.2 Data and workload distribution . . . 24
3.2.1 Domain decomposition . . . 24
3.2.2 Adjustment of domain decomposition to the identification of differentially methylated regions . . . 25

3.3 Optimization techniques . . . 26
3.3.1 AllInOneGo File Procesing . . . 26
3.3.2 Parallel Input/Output . . . 28

4 Experimental evaluation **33**
4.1 Test environment . . . 33
4.1.1 System . . . 33
4.1.2 Datasets . . . 37
4.2 Previous scalability concepts . . . 38
4.3 One node test . . . 39
4.3.1 Selection of the best schedule . . . 40
4.4 Scalability tests . . . 43
4.4.1 Input/Output tests . . . 43
4.4.2 ParRADMeth, pure MPI version . . . 44
4.4.3 ParRADMeth, hybrid version without *Hyperthreading* . . . 45
4.4.4 ParRADMeth, hybrid version with *Hyperthreading* . . . 46
4.5 General conclusions . . . 48

5 Planning and organization **51**
5.1 Project planning . . . 51
5.1.1 Phase 1: Analysis of the state of the art and understanding of the original tool . . . 51
5.1.2 Phase 2: Design and implementation of a basic parallel tool with the whole functionality . . . 51
5.1.3 Phase 3: Addition of optimization techniques . . . 52
5.1.4 Phase 4: Performance evaluation . . . 52
5.1.5 Phase 5: Documentation and report writing . . . 52
5.2 Project metrics . . . 53
5.2.1 Time . . . 53
5.2.2 Budget . . . 54

6 Conclusions **57**
6.1 Conclusions . . . 57
6.2 Relation to the bachelor's title . . . 58
6.3 Future work . . . 58

A User Guide **61**
A.1 Prerequisites . . . 61
A.2 Compilation . . . 61

A.3 Execution 62

List of Acronyms **63**

Bibliography **65**

List of Figures

2.1 CpG site on the left vs. Non-CpG site on the right 4
2.2 Example of simple linear regression. Independent variable vs. dependent variable 5
2.3 Work flow of the MethPipe package for analyzing bisulfite sequencing dataset 6
2.4 Example of CpG site with coverage and methylation levels 7
2.5 Example line on .meth file 7
2.6 Example of proportion table 8
2.7 Example of a design matrix with two factors 8
2.8 Example of a RADMeth's output file 9
2.9 Beta distribution for different values of α and β 10
2.10 RADMeth's computation phase pseudocode 12

3.1 Abstraction of a distributed memory system with several cores and one memory module per node 18
3.2 Simple communication with Send and Recv 20
3.3 Collective communication with Gather 20
3.4 Collective communication with AllGather 21
3.5 OpenMP ”parallel” directive example 23
3.6 OpenMP ”parallel for” directive example 23
3.7 ParRADMeth's computation phase pseudocode 27
3.8 ParRADMeth's parallel input phase pseudocode 29
3.9 ParRADMeth's concurrent input phase pseudocode 30
3.10 ParRADMeth's parallel output phase pseudocode 31

4.1 *Pluton* cluster general structure 34
4.2 Static schedule's speedups. Processing elements (cores) vs. speedup over RADMeth 41

4.3 Dynamic schedule's speedups. Processing elements (cores) vs. speedup over RADMeth 42
4.4 Guided schedule's speedups. Processing elements (cores) vs. speedup over RADMeth 42
4.5 Speedups in the I/O phases for the Hansen dataset. Processing elements (cores) vs. speedup over RADMeth 44
4.6 Speedups for the pure MPI version. Processing elements (cores) vs. speedup over RADMeth 45
4.7 Speedups of the hybrid version without using *Hyperthreading*. Processing elements (cores) vs. speedup over RADMeth 46
4.8 Speedups for the hybrid version when using *Hyperthreading*. Processing elements (cores) vs. speedup over RADMeth 47

5.1 Gantt chart showing the arrangement of the different phases of the project . . 56

List of Tables

4.1 compute-0 nodes specification 35
4.2 Datasets specification 37
4.3 Execution time from the hybrid tool in one node using different schedules (in seconds) 41
4.4 Execution times in the I/O phases for Hansen dataset 43
4.5 Execution time for the pure MPI version (in seconds) 44
4.6 Execution time from the hybrid version without using *Hyperthreading* (in seconds) 46
4.7 Execution time from the hybrid version when using *Hyperthreading* (in seconds) 47
4.8 Summary of execution times for different versions of the tool (in seconds) . . 48

5.1 Total hours inverted in the project detailed by task 54
5.2 Total hours invested in the project by each resource 55
5.3 Total costs of the project detailed by resource 55

Chapter 1

Introduction

THIS introductory chapter contextualizes the work and summarizes the motivation of the bachelor's thesis, followed by an overview of the main objectives, and an explanation of the structure of this document.

1.1 Motivation

Parallel and High Performance Computing (HPC) has gained attention in the last years as a mean to accelerate several kind of computationally expensive applications. One field that can take advantage of HPC is bioinformatics, where the datasets to be analyzed can be nowadays extremely large thanks to the reduction in the cost of obtaining biological data. Therefore, many bioinformatics tools used to work with this huge datasets require extremely high computational times.

DNA methylation is a chemical modification of DNA resulting from the addition of a methyl group to a DNA nucleotide. In vertebrates, DNA methylation (which mainly occurs at cytosines within CpG dinucleotides) has been associated with several biological functions. For example, methylation plays a key role in genomic imprinting, X-chromosome inactivation, and it has been associated with the suppression of transposable elements during embryonic development [1]. Some studies have shown correlation between promoter methylation and gene expression [2, 3]. Furthermore, the presence of large-scale abnormally methylated genomic regions [4] is a hallmark feature of many types of cancers [5, 6]. Whole-Genome Bisulfite Sequencing (WGBS) is currently the state-of-the-art technology for obtaining a comprehensive nucleotide-resolution view of the epigenome. A number of approaches currently exist for assessing Differential Methylation (DM) from WGBS data.

The use of parallel computing makes up a good solution to reduce execution times associated to DM analysis methods. During this bachelor's thesis a parallel tool was developed to analyse WGBS data meant to be executed on distributed memory systems such as clusters

or supercomputers. The novel parallel tool is based on RADMeth [7], a sequential tool that performs this analysis based on beta-binomial regression models and Z test. This base tool is known as one of the best methods to achieve highly precise results and it also shows high sensitivity and specificity.

1.2 Objectives

The main goal of this bachelor's thesis is to design and implement a parallel tool for the identification of differentially methylated regions in genomic analyses: ParRADMeth (*Parallel Regression Analysis of Differential Methylation*). This tool allows the use of these algorithms on HPC systems, significantly reducing their execution times. The parallel tool is based on RADMeth, implemented in C++.

HPC systems, which are the target of this bachelor's thesis, are multinode platforms (clusters and supercomputers) where each node is a multicore system. Therefore, taking the most out of the physical resources as our main goal, the design of the parallel algorithm includes techniques using explicit parallelism, that is, processes communicating through message passing, and implicit parallelism, through threads working on shared memory. Concretely, Message Passing Interface (MPI) and OpenMP are used to provide both types of parallelism.

As a secondary goal, the performance of the tool is compared to that of RADMeth using a real environment. For that purpose, several datasets are selected to evaluate the tool and to prove the upgrades that parallel computing provides. Finally, the parallel tool has been released so scientists and analysts can take advantage of it.

1.3 Document Structure

After this introductory chapter, this document continues with a chapter to introduce some previous concepts and give some contextualization to the identification of differentially methylated regions in genomic analyses. Next, third chapter explains in detail the design and implementation of the parallel tool ParRADMeth. After that, scalability tests are explained, together with some conclusions of the results. Fifth chapter shows the different phases and costs of the project. Finally, last chapter summarizes the conclusions and provides some ideas for future improvements.

Chapter 2

Identification of differentially methylated regions

THIS chapter gives an introduction to the identification of Differentially Methylated Regions (DMRs) in genomic analyses, focusing on RADMeth [7], one of the most accurate tools, that performs this analysis using beta-binomial regression to obtain high-precision results. As it was already mentioned, this tool has been proved superior that others in terms of sensitivity and specificity [8]. This chapter also gives an overview of these other tools that follow different approaches to perform a similar analysis.

2.1 Basic concepts

Before starting with the subject of study, it seems appropriate to clarify the meaning of some concepts that will often appear during this chapter and, in general, during the whole document.

- **CpG site**. Deoxyribonucleic Acid (DNA) is a nucleic acid that contains the genetic instructions used in the development and functioning of all living organisms. DNA can be specified naming only the nitrogen-containing nucleobases that compose it (that can be cytosine [C], guanine [G], adenine [A] or thymine [T]). CpG sites are regions of DNA where a cytosine [C] nucleotide is followed by a guanine [G] nucleotide along its 5' $\rightarrow$ 3' (DNA can be read in two directions, one end is called 5' and the other one is called 3'. This indicates the direction in which the cytosine [C] has to appear before the guanine [G]). Figure 2.1 shows an example of a CpG site.

- **Methylation**. A methyl group is a small molecule made of one carbon and three hydrogen atoms. Methyl groups are added or removed from proteins or nucleic acids and may change the way these molecules act in the body. Methylation is a chemical process

Figure 2.1: CpG site on the left vs. Non-CpG site on the right

that modifies DNA through the addition of a methyl group to one or several nucleotids. Methylation of high-density CpG regions has been widely described as a mechanism associated with gene expression regulation [2].

- **WGBS**. Whole Genome Sequencing (WGS) is the process of determining the entirety, or nearly the entirety, of the DNA sequence of an organism's genome at a single time. It provides the most comprehensive collection of an individual's genetic variation. With the falling costs of sequencing technology, WGS has become the leader paradigm in genotyping studies, surpassing the previous leader in this field, DNA microarrays. For humans, WGS provides 3,000 times more data than this previous leading technology. WGBS, is a next-generation sequencing technology used to determine the DNA methylation status of single cytosines [C] by treating the DNA with sodium bisulfite before sequencing. After sequencing, the unmethylated cytosines [C] appear as thymines [T].

- **Regression**. It is a statistical method used in finance, investing, and other disciplines that attempts to determine the strength and character of the relationship between one dependent variable (usually denoted by Y) and a series of other variables (known as independent variables). One example of this is simple linear regression, shown in Figure 2.2, a specific form of regression analysis to finds a linear function that predicts the dependent variable based on a single independent variable. Regression analysis can be used to infer causal relationships between the independent and dependent variables.

- **P-value**. The probability that a particular statistical measure of an assumed probability distribution will be greater than or equal to (or less than or equal to in some instances) observed results. The p-value is used in the context of null hypothesis testing in order to quantify the idea of statistical significance of evidence, the evidence being the observed value of the chosen statistic. Null hypothesis testing is a *reductio ad absurdum* argument adapted to statistics.

- **Sample**. It is related with the concept in the field of statistics, where a sample is a set of cases collected or selected from a statistical population by a defined procedure. In

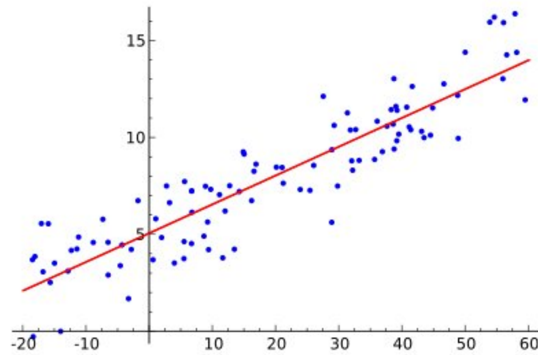

Figure 2.2: Example of simple linear regression. Independent variable vs. dependent variable

this case, a sample is equivalent to a single case. From the data plain point of view, a sample can be seen as a vector of values for the study. In our domain, a sample can be, for example, a virus or a human being.

- **Dataset**. A dataset is a collection of data that stores the relation between the subset of samples considered with the values of each feature to study. In general this relations are represented as a matrix. We will discuss the format of datasets in our domain in Section 2.2.2.

2.2 RADMeth

The main objective of this bachelor's thesis is to develop a parallel tool that gets results as precise as possible. Since RADMeth [7] was one the tools that showed superiority in this field [8], we took the decision of developing a tool that provides exactly the same accurate biological results as RADMeth. It is important to know how this tool works in order to understand the development of ParRADMeth. This section starts with an overview of RADMeth to then focus on the format of the input/output data and which computations it performs to reach the desired results.

2.2.1 Overview

RADMeth is a publicly available software[1] for computing individual differentially methylated sites and genomic regions in data from WGBS experiments. The tool uses beta-binomial regression for high-precision DM analysis over WGBS data, and it can handle medium-size experiments where it becomes critical to accurately model variation in methylation levels

[1] http://smithlabresearch.org/software/methpipe/

among replicates, and it accounts for influence of various experimental factors such as cell types or batch effects.

RADMeth is part of MethPipe [9], a computational pipeline for analyzing bisulfite sequencing data. That means that raw data from this WGBS experiments has to be preprocessed to be used as input data from RADMeth and the results the tool produces can be used as input data for the next phases of the pipeline. Figure 2.3 shows the workflow of the pipeline for the identification of differentially methylated regions in WGBS datasets.

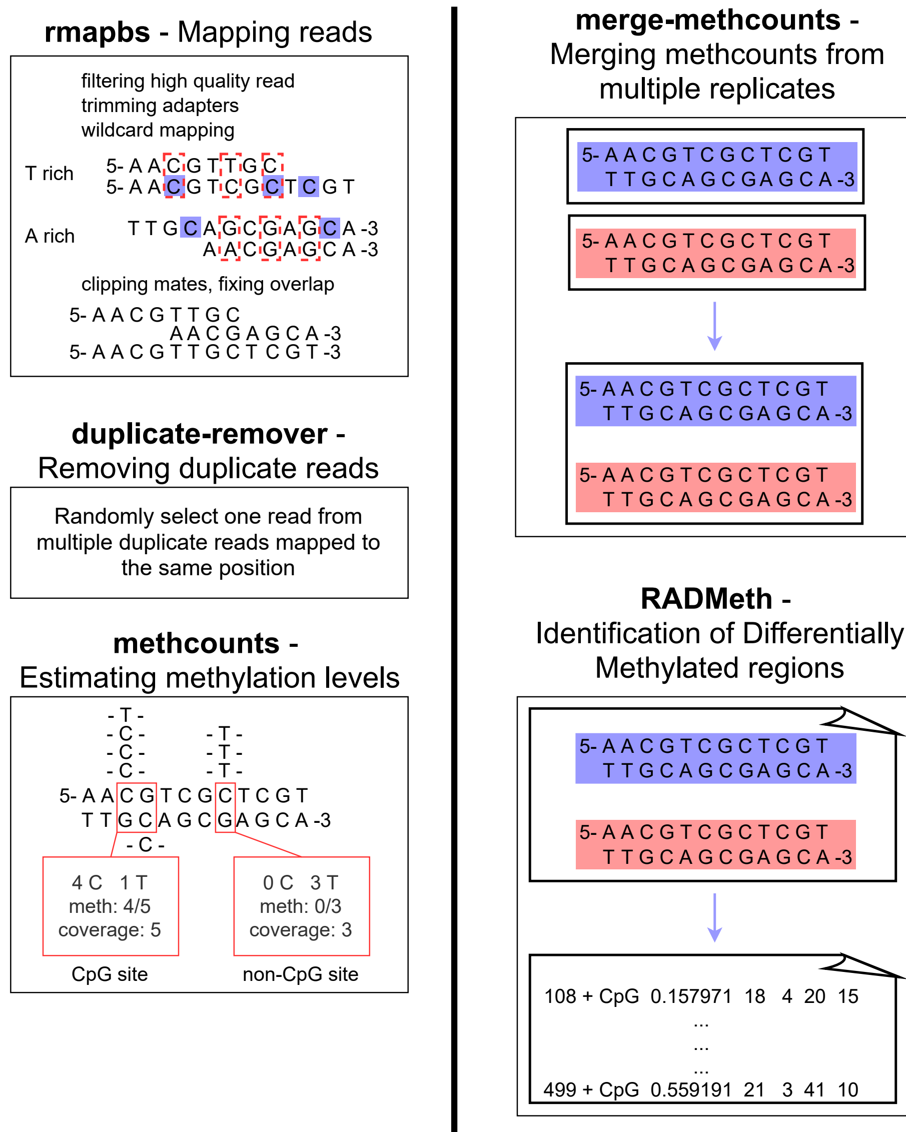

Figure 2.3: Work flow of the MethPipe package for analyzing bisulfite sequencing dataset

2.2.2 Input data

To start understanding how the tool works, it is necessary to explain which data it takes as input. Since the tool is part of MethPipe, the data it takes as input comes as the result of processing raw data from WGBS experiments using the previous tools of the pipeline. Specifically, RADMeth takes two files as input: a **proportion table** and a **design matrix**. The datasets to be analysed can be usually found as a set of *.meth* files that need to be combined using the pipeline to finally build the proportion table, and then manually create the design matrix.

- **.meth files**. These files contain the information about the CpG sites of a sample, including, among others, coverage and methylation levels for the site. Each line contains all the information of a CpG site, so the file has as many lines as CpG sites in the sample. Coverage indicates how many cytosines [C] are in the CpG site before sequencing, and methylation level indicates the percentage on them that after sequencing still appear as cytosines [C], proving that they are methylated. In figure 2.4 we can see a visual representation of a CpG site with five cytosines [C] before sequencing, four of them methylated and one unmethylated (it appears as a thymine [T]), while in Figure 2.5 we can see the corresponding line to that CpG site on a *.meth* file.

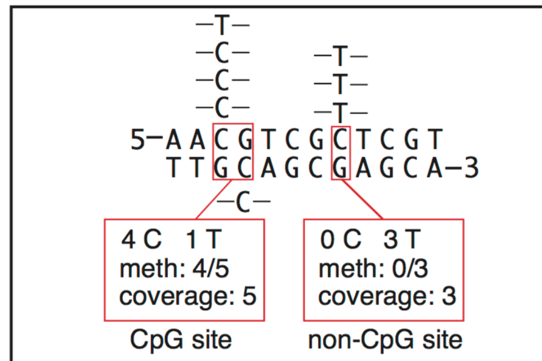

Figure 2.4: Example of CpG site with coverage and methylation levels (http://smithlabresearch.org/software/methpipe/)

```
chr1  108 + CpG 0.8 5
```

Figure 2.5: Example line on .meth file

- **Proportion table**. Once all the *.meth* files are available, they must be merged into a single file, the proportion table. Methpipe provides the tool *merge-methcounts* to

generate the proportion table from the different files. In Figure 2.6 we can see that the format of the proportion table is very similar to the one of the *.meth* files, except for three small differences. First of all, it starts with a header line to indicate the order of the samples in the following lines. Second, each line does not contain information for a Cpg site on a sample, but for the CpG site in all samples of the experiment. And last, the methylation level does not appear as a percentage but as the absolute value of methylated cytosines [C].


```
control_a    control_b    control_c    case_a    case_b    case_c
chr1:108:109    9    6    10   8    1    1    2    2    2    1    14   1
chr1:114:115    17   7    10   0    14   3    5    1    9    1    7    1
chr1:160:161    12   8    10   5    17   4    15   14   13   6    4    4
chr1:309:310    1    1    1    0    17   12   12   8    2    1    19   8
chr1:499:500    8    4    6    5    15   6    14   10   14   11   15   1
```

Figure 2.6: Example of proportion table

- **Design matrix**. It is manually created and describes the structure of the experiment. For example, the design matrix shown in Figure 2.7 shows that samples in this example dataset are associated with two factors: base and case. The first column corresponds to the base factor and will always be present in the design matrix (it can be seen as stating that all samples have the same baseline mean methylation level). The second column is added to distinguish cases from controls.

```
base case
control_a 1 0
control_b 1 0
control_c 1 0
case_a 1 1
case_b 1 1
case_c 1 1
```

Figure 2.7: Example of a design matrix with two factors

2.2.3 Output data

One objective of the work is to provide exactly the same accurate biological results as RADMeth. Therefore, it is important to understand the output format of this tool in order to use the same one for the results of our parallel tool.

RADMeth generates a single output file for each experiment. As in the input proportion table or in the *.meth* files, the output file contains the information for all the CpG sites

analyzed, one line per site, without header line since the results are not at sample but at experiment level. So for each CpG site its line contains:

- **The first four columns**, with general information about the site, such as its position.
- **The fifth column**, which contains the p-value of the experiment for that CpG site.
- **The four last columns** correspond to the total coverage counts and methylated read counts of the case and control groups, respectively.

In Figure 2.8 we can see the first lines of an output file as example of this format.

```
chr1    108    +    CpG    0.157971     18    4     20    15
chr1    114    +    CpG    0.559191     21    3     41    10
chr1    160    +    CpG    0.0951122    32    24    39    17
chr1    309    +    CpG    0.239772     33    17    19    13
chr1    499    +    CpG    0.77014      43    22    29    1
```

Figure 2.8: Example of a RADMeth's output file

2.2.4 Computation phase

Now that we know which data RADMeth takes as input and the result that it produces, it is time to focus on the method the tool uses to reach these results. First of all, the structure of the tool must be understood. The behavior of the tool can be logically divided in three phases:

- **Input phase**, where data is read from input files into the appropriate structures.
- **Computation phase**, where data is processed by the algorithm to produce final results.
- **Output phase**, where results are written into the output file.

Before focusing on the algorithm used on the computation phase it is necessary to understand some concepts.

- **Binomial distribution**. A binomial distribution with parameters n and p is the discrete probability distribution of the number of successes in a sequence of n independent experiments, each one asking a yes–no question, and each one with its own Boolean-valued outcome: success (with probability p) or failure (with probability $q = 1 - p$).

$$P[X = x] = \binom{n}{x} p^x (1-p)^{n-x}$$

- **Beta distribution**. The beta distribution (see Figure 2.9) is a family of continuous probability distributions defined on the interval [0, 1] parameterized by two positive shape parameters, denoted by α and β, that appear as exponents of the random variable and control the shape of the distribution.

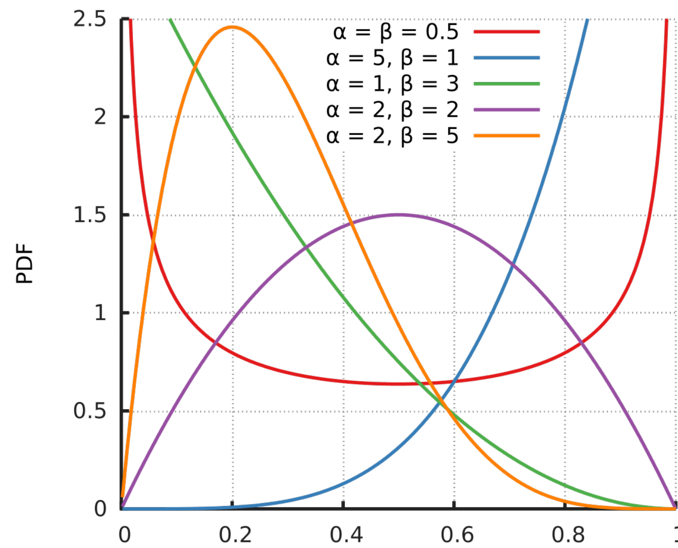

Figure 2.9: Beta distribution for different values of α and β

- **Beta-binomial distribution**. The *b*eta-binomial distribution is the binomial distribution in which the probability of success at each of *n* trials is not fixed but randomly drawn from a beta distribution.

$$P[X = x] = \binom{n}{x} \frac{\mathrm{B}(k + \alpha, n - x + \beta)}{\mathrm{B}(\alpha, \beta)}$$

RADMeth uses a beta-binomial regression model [10], a form of regression based on a beta-binomial distribution, to perform its high-precision DM analysis in WGBS experiments. This regression model is adequate for this task, since for each CpG site of a single sample the binomial distribution is a method that provides high-precision results.

This regression model is fitted separetaly for every CpG site. In fact, to determine if a site is differentially methylated with respect to the test factor (for example, ”case” could be the test factor when computing the experiment defined in the design matrix from Figure 2.7), RADMeth fits two regression models: the full model and the reduced model without the test factor. The significance of DM is determined by comparing the full and the reduced models using the log-likelihood ratio test.

In Figure 2.10 a pseudocode to illustrate this computation phase is shown. The tool reads

each row from the input proportion table to the appropriate structure (Line 3). It calculates the coverage and methylation levels for all samples with and without the test factor (Lines 6-12). Then it checks if the CpG site has either low coverage or the same methylation levels through all samples with or without the test factor (Lines 13-16). If the CpG site passes the checks, RADMeth fits full and reduced regressions and analyse results to obtain the p-value (Lines 17-21). Finally, the results of that CpG site are written to the output file (Line 22). More information about this method can be found in [7].

```
1  test_factor ← GetTestFactor();
2  foreach CpG site cpg_site in the proportion_table do
3      full_regresion ← WriteData(cpg_site);
4      coverage_factor ← 0, coverage_rest ← 0;
5      methylation_level_factor ← 0, methylation_level_rest ← 0;
       /* Agregate coverage and methylation levels for
          samples with and without the test factor       */
6      foreach Sample s in the design_matrix do
7          if s has the test factor then
8              coverage_factor += full_regression[s].coverage;
9              methylation_level_factor += full_regression[s].methylation_level;
10         else
11             coverage_rest += full_regression[s].coverage;
12             methylation_level_rest += full_regression[s].methylation_level;
       /* Calculate the p-value                          */
13     if zero coverage over all case or control samples then
14         p_value ← -1;
15     else if methylation level is identical in all samples then
16         p_value ← -1;
17     else
18         Fit(full_regression);
19         reduced_regression ← CopyWithoutTestFactor(full_regression,
             test_factor);
20         Fit(reduced_regression);
21         p_value ← LoglikratioTest(reduced_regression, full_regression);
22     WriteToOutputFile(cpg_site, p_value, coverage_factor, coverage_rest,
         methylation_level_factor, methylation_level_rest);
```

Figure 2.10: RADMeth's computation phase pseudocode

2.3 Other methods

In the WGBS field there is not a set of clear good practices or a de-facto standard defined. This is the reason why there exist several alternatives to perform its analyses and each of them obtains a different quality of results. Next, some of the most popular approaches for DM analysis are listed, discussing advantages and disadvantages among them.

2.3.1 Fisher's Exact Test and Hidden Markov Models

One of the most straightforward and commonly used methods for comparing epigenomes of a pair of samples is the Fisher's Exact Test [11, 12, 13, 14]. Although it is valid for all sample sizes, in practice it is employed when sample sizes are small. There are also DM detection algorithms based on Hidden Markov models (HMM), and even one tool using a HMM-based method for DM detection is included in the MethPipe pipeline [15, 16]. Existing methods based on Fisher's Exact Test and HMMs are appropriate for comparing a pair of samples at a time, however, they lack the ability to account for variability of methylation levels among replicates.

2.3.2 Algorithms based on smoothing

Another variety of DM detection algorithms are based on smoothing. These methods operate under the assumption that methylation levels vary smoothly along the genome. They use local smoothing to estimate the true methylation level of each site in each sample. Some example implementations of this variety of algorithms are:

- **BSmooth**. BSmooth methylation analysis pipeline [17] is designed to compute DMRs between two groups of samples. After smoothing, this tool performs a statistical test, similar to the t-test, to find DM sites which form DMRs.

- **BiSeq**. BiSeq [18] is a package that provides useful classes and functions to handle and analyze targeted bisulfite sequencing data such as reduced-representation bisulfite sequencing data. In particular, it implements an algorithm to detect DMRs. The package takes already aligned bisulfite sequence data from one or multiple samples. The package is implemented in R, and is open source. Unlike BSmooth, it provides a smoothing-based method that can be used for experiments that go beyond comparing two groups of samples, but it requires a set of candidate regions that may exhibit DM.

Because smoothing-based methods perform smoothing on each sample individually, care must be taken when dealing with regions where methylation levels are difficult or impossible to estimate.

2.3.3 Algorithms based on beta-binomial distributions

A few DM-detection methods are based on the beta-binomial distribution, which is a natural choice for describing methylation levels of an individual site across replicates as it can account for both sampling and epigenetic variability. Based on this idea we can see:

- **DSS package**. This tool [19] implements a method that constructs a genome-wide prior distribution for the beta-binomial dispersion parameter and then uses it to estimate

the distribution of methylation levels in each group of replicates. The differentially methylated sites are determined by testing the means of these distributions for equality.

- **MOABS algorithm**. This algorithm [20] constructs a genome-wide distribution of methylation levels and then uses it to estimate the distribution of methylation levels at individual sites. The significance of DM is subsequently determined by an estimate of the methylation difference between the two groups of replicate samples.

The precision of these methods for a given site depends on how closely the distribution of site's methylation levels is across replicates or the dispersion parameter resembles the genome-wide prior.

2.3.4 Algorithms based on regression

Another category of DM detection algorithms are based on regression. Some example implementations are shown next.

- **BiSeq**. This tool mentioned earlier performs a beta regression after smoothing and so it also fits into this category.
- **limma**. This package [21] provides data analysis, linear models and differential expression for microarray data and was recently also applied to test CpG sites for DM. The package is implemented in R, and is open source. It provides a method based on linear regression to perform the differential analysis.
- **COHCAP**. The tool [22] provides a pipeline to analyze single-nucleotide resolution methylation data. It provides DM for CpG sites, DM for CpG islands, and integration with gene expression data. It is implemented as both a R package and a standalone Java/Perl program pointing to the R script. The pipeline is open source.
- **methylKit**. This tool [23] is a R package for DNA methylation analysis and annotation from high-throughput bisulfite sequencing. Methylation calling can be performed directly from Bismark aligned BAM files. The package is implemented in R, and is open source. It provides a method based on logistic regression to perform the differential analysis, which assumes that the number of reads indicating methylation follows a binomial distribution across replicates.
- **Stats**. It is a package [24] that implements different regression methods and allows us to perform differential analysis over our datasets. It is implemented in R and brings, among others, three regression-based methods to analyse our data. The models used for the analysis are, negative binomial regression, poisson regression and poisson regression with dispersion parameter.

2.3.5 Parallel algorithms

Even though we have seen that there exist several methods to perform DM analysis and that most of them have a high computational cost, only RADMeth offers a parallel approach to be executed on a HPC cluster consisting of splitting the proportion table into smaller tables to be processed separately, and subsequently combining the results. However, this is a naive approach that does not take into account several problems, such as unbalances in the workload. This adds extra importance to this bachelor's thesis, since reducing the execution time is a big concern when dealing with DM analysis.

2.3.6 Conclusions

Two main conclusions can be extracted from this overview of the state of the art:

1. The existing methods for detecting DM lack either the ability to analyze WGBS datasets in complex experimental designs or the ability to account for variation across biological replicates. Methods based on beta-binomial regression can overcome these limitations.
2. Execution times of this methods are high, however no appropriate parallel tool for HPC enviroments exists. By developing one, this bachelor's thesis represents an advance in the field of bioinformatics.

Chapter 3

Design and Implementation

This chapter gives a detailed overview of the process followed to obtain the final version of ParRADMeth, explaining the target architecture, the program structure and the programming techniques used to improve the performance. The code is stored on a GitHub repository [1], and since one of the main goals of the project is to make it available for the scientific community, this work will be released soon. Two versions will be released, a pure MPI version and a hybrid version using both MPI and OpenMP.

3.1 Target architecture

Programs that have a high computational complexity or work with large volumes of data are not made for being executed on personal computers, but on cluster systems composed of several nodes. The target architecture of this bachelor's thesis is a distributed memory system with several nodes interconnected through a network, each of them with a memory module and several cores (see Figure 3.1). Parallel computing on this type of systems usually follow the *Single Program Multiple Data (SPMD)* model, meaning that the workload is divided into different tasks that are split up among multiple processors and run simultaneously with different inputs, so that all nodes and cores cooperate in order to obtain results faster. Computational performance on a cluster depends on several factors such as the number of nodes, the number of cores, the number of cores per node, the network features or the memory transfer rates.

Let's see how systems with these characteristics can be used, getting a little deeper into the architecture. First of all, a node is a unit that can be seen as a computer, that is, it is composed of main memory, processing cores, storage and input/output system. Different nodes form the first level to distribute the workload: each node can execute different tasks of the main program.

[1] https://github.com/afdezfraga/ParRADMeth

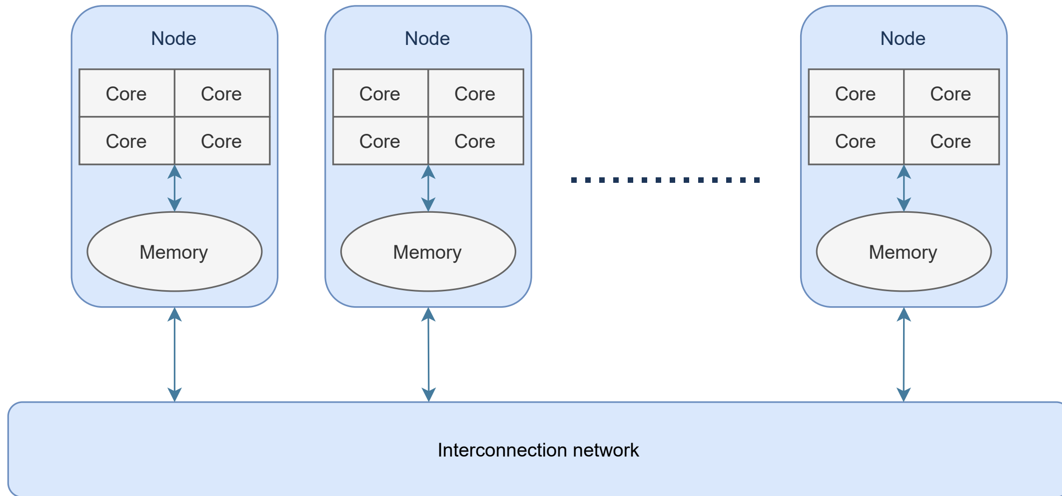

Figure 3.1: Abstraction of a distributed memory system with several cores and one memory module per node

Since each node has its own memory address space, shared data must be send through the network among nodes, that is why this sort of systems are traditionally programmed with the message passing technique. This parallelism level is called explicit, as it is the programmer the one who must specify which data is stored in each memory and when messages are sent between one node and another.

On a lower level there is implicit parallelism, witch appears inside each node. This happens because each node has several processing cores, allowing to share out the tasks among them and to access the node's shared memory. Unlike explicit parallelism, implicit parallelism does not need message passing for communication among tasks, nevertheless some auxiliar mechanisms may be needed to avoid conflicts on shared memory accesses. This level of parallelism is generally easier for the programmer, since there is no need to worry about process communication.

A system with this architecture usually has nodes with hardware accelerators, that is, elements to reach a higher performance on some task that the one that can be reached using a general purpuse CPU. One example are Graphics Processing Units (GPUs). These accelerators can be used as processing units, however this project does not focus on this sort of components, but on applying both explicit and implicit parallelism to make use of all the CPUs of a cluster.

3.1.1 MPI

Message Passing Interface (MPI) is the de facto standard for programming parallel distributed-memory systems and it is, probably, the most widely used programming framework in the HPC community. It follows the SPMD paradigm, i.e., it splits the workload into different tasks that are executed on multiple processors. The MPI specification allows to work with multiple processing elements coordinated through message passing.

A parallel MPI program consists of several processes, each one with associated local memory, that can communicate through the interconnection network by using send and receive routines. The cost of these communications relies on hardware characteristics, especially the interconnection network ones. The MPI standard includes point-to-point message-passing communications, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management of one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics and a profiling interface. All of them are defined for C and Fortran languages. Currently, there are several widely used free implementation such as OpenMPI[2] or MPICH[3]. Additionally, most vendors of HPC hardware offer an MPI implementation optimized for their hardware. The most important MPI routines used in this bachelor's thesis are:

- **MPI_Init, MPI_Finalize and MPI_Abort**. The first functions mark the beginning and the end of the parallel program, respectively. Their role is to create and delete, respectively, all the structures needed by the MPI program to be able to send messages among all processes. As for *MPI_Abort*, it allows a forced ending of all MPI processes being called just by one of them.

- **MPI_Comm_rank and MPI_Comm_size**. They are usually called together and they allow to know actual process identifier and total number of processes on a given program.

- **MPI_Send and MPI_Recv**. Despite the fact that these two functions are not directly used in the development of the project, they establish the base of MPI communications, since they allow the most basic communication between two processes. Their use is shown in Figure 3.2. They are used together, so that a process sending the message uses *MPI_Send* and a process receiving it uses *MPI_Recv*. Both of them are blocking, which means that a process execution stops until the transfer ends. For non-blocking communications there are alternatives such as *MPI_Isend* and *MPI_Irecv*.

[2] https://www.open-mpi.org/
[3] https://www.mpich.org/

Figure 3.2: Simple communication with Send and Recv

- **MPI_Allgather**. Up to now only point to point communications have been discussed. Nevertheless, *MPI_Allgather* is not a point to point communication but a collective operation. Collective operations do not involve only two processes, but a group of them. They are internally implemented to obtain better performance than a simple implementation by means of *MPI_Send* and *MPI_Recv*. We will explain how *MPI_Allgather* works on the basis of its basic version *MPI_Gather*. This function takes elements from many processes and gathers them to one single root process. The elements are ordered by the rank of the process from which they were received. In Figure 3.3 this behaviour is shown. *MPI_Allgather* does the same, but this function gathers all the elements to all the processes, instead of only to one root process, as it is shown in Figure 3.4.

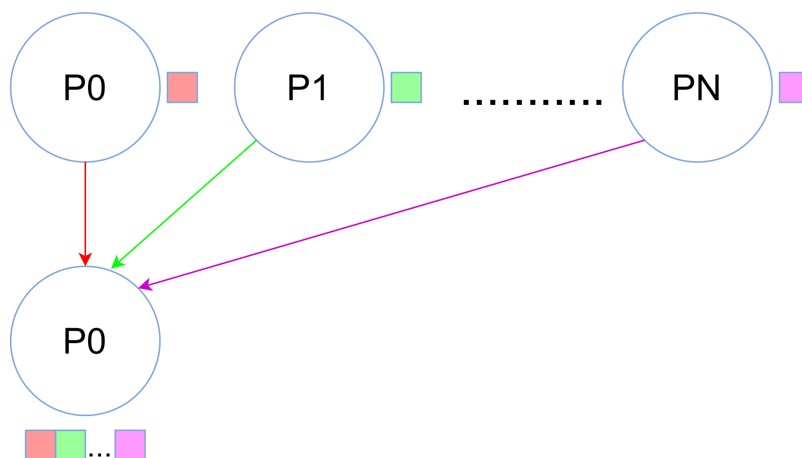

Figure 3.3: Collective communication with Gather

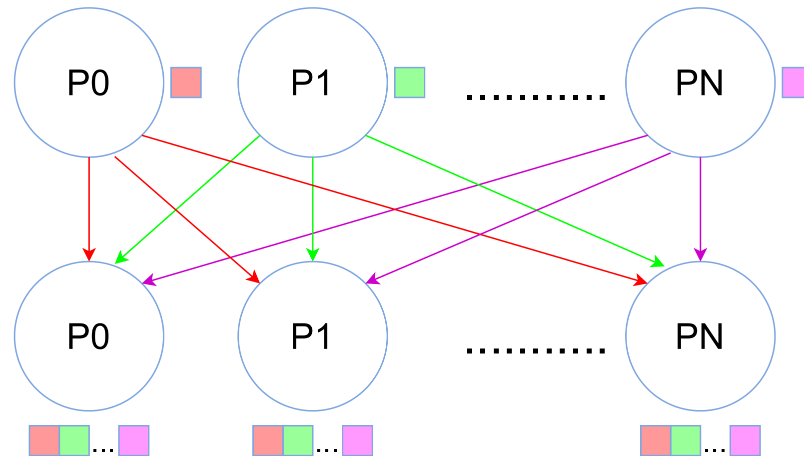

Figure 3.4: Collective communication with AllGather

In addition to these listed routines, several MPI functions related to I/O were also used. MPI-IO [25] is a convenient interface for enabling true parallel I/O on systems that support it, providing mechanisms for performing synchronization, syntax for data movement and means for defining non-contiguous data layout in a file [26]. MPI-IO interface has become the standard mechanism for file I/O within HPC applications. Without MPI-IO two common alternatives to parallel I/O are left.

1. Main process accesses a file and it is in charge of performing all the I/O and gathering/scattering data from/to other processes.

2. Each process opens a separate file and performs I/O to it independently.

These alternatives, even though are simple to code, respectively present poor scalability and challenges with file management. One of the best advantages of MPI-IO over UNIX I/O is that the former has the ability to specify non contiguous accesses in a file and related memory buffers, which is a common need in parallel applications. Next, some MPI-IO functions used during the work are listed:

- **MPI_File_open and MPI_File_close**. These functions are collective operations that allow to open and close a file, respectively, to be accessed in parallel by several processes.

- **MPI_File_get_size**. This function is a data access operation that returns the size in bytes of the file.

- **MPI_File_read_at_all**. This operation is a collective function that allows processes to read in parallel from a file using an explicit offset that can be different for each one.

- **MPI_File_write_at_all**. This operation is a collective function that allows processes to write in parallel to a file using an explicit offset that can be different for each process.

3.1.2 OpenMP

OpenMP[4] is one of the most used approaches for parallel programming on shared memory systems for C, C++ or Fortran applications. The OpenMP specification defines a collection of compiler directives, library routines and environment variables that implement multithreading with the fork-join model. A main thread runs the sequential parts of the program, while additional threads are forked to execute parallel tasks. Threads communicate and synchronize by using the shared memory.

The main advantages of OpenMP are its portability and ease of use. Currently there are compilers for practically all architectures and its use usually involves less changes in the structure of the code than the use of other low-level shared-memory libraries as POSIX threads.

Even though a pure MPI program can take advantage of all the cluster's hardware by using one process per core, using a hybrid approach with processes that create OpenMP threads for the parallelization inside each node has several benefits:

1. Threads are lighter than processes, so creating and destroying them is usually faster. Also context switching among threads of the same process is less expensive.

2. Memory overload reduction, since threads can access the same shared memory structures, while MPI processes need a copy of the structures for each process.

3. Posibility of execution in *Simultaneous MultiThreading (SMT)* mode. SMT is a technique for improving the overall efficiency of superscalar CPUs with hardware multithreading. It is based on the simulation of two logical threads on a single CPU core, in order to merge one thread instructions with the other ones, taking advantage of CPU cycles that would be free in other way. Two concurrent threads per CPU core are common, but some processors support up to eight concurrent threads per core. This technique is better known as *Hypertheading*.

Next, the most relevant directives used during this bachelor's thesis are listed:

- **pragma omp parallel**. This directive spawns a team of OpenMP threads that execute the code region as it can be seen in Figure 3.5.

- **pragma omp parallel for**. This directive is actually a shortcut of other two directives, *pragma omp parallel*, that has been explained before and *pragma omp for*. This second

[4] https://www.openmp.org/

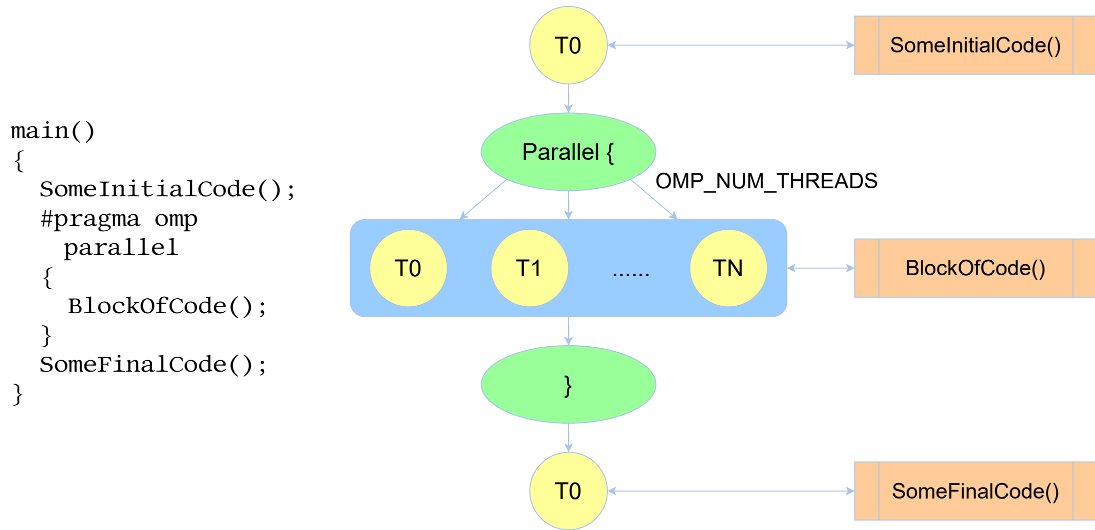

Figure 3.5: OpenMP "parallel" directive example

directive, when called over a for loop inside a block tagged with the *pragma omp parallel* directive, divides loop iterations among spawned threads executing the code block. This directive can be used as shown in Figure 3.6.

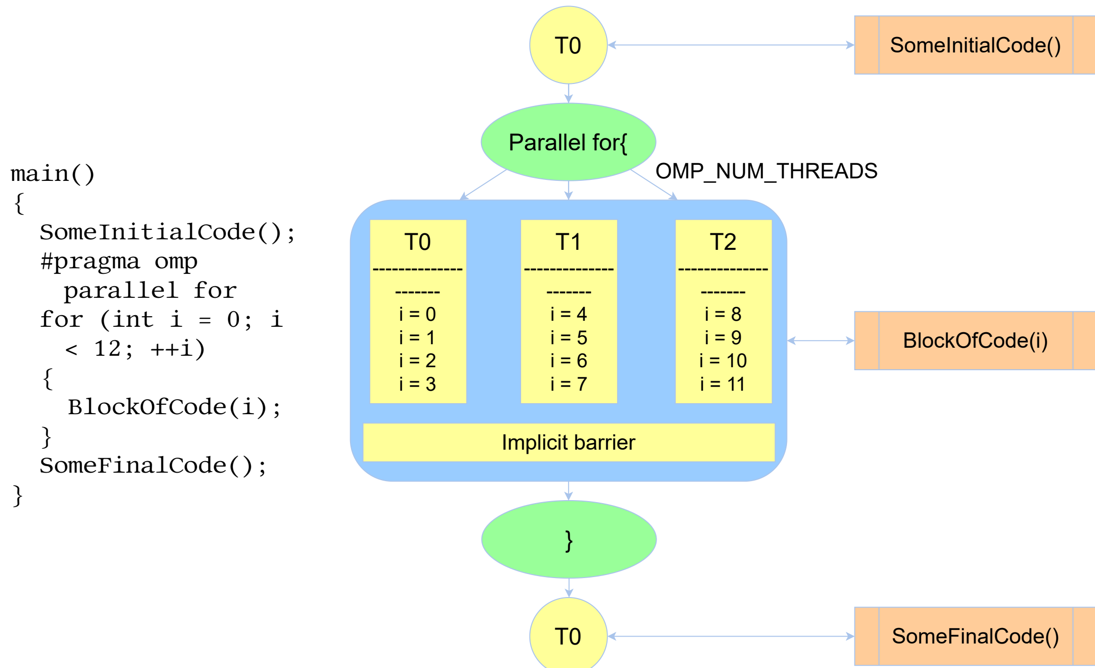

Figure 3.6: OpenMP "parallel for" directive example

These directives can have some clauses to specify some general or data-sharing attributes. The most important attributes in the case of the developed parallel tool are:

- **shared, private and firstprivate**. These three clauses specify data-sharing attributes and receive as parameter a comma-separated list of variables. The first two clauses specify that variables should be shared among all threads or that each thread should have its own instance of a variable, respectively. The *firstprivate* clause, specifies that each thread should have its own instance of a variable, but it also specifies that the variable should be initialized with the value assigned to the variable prior to the parallel construct.
- **schedule**. This clause applies to the *for* directive and specifies how iterations are distributes among threads. The following schedule options are available:
  - **static**. Batches of iterations are distributed statically (distribution is done before entering the loop) in a round-robin fashion.
  - **dynamic**. Batches of iterations are distributed with a first-come-first-served policy until no batch remains.
  - **guided**. The same as *dynamic*, but with batches whose sizes get smaller and smaller, down to 1.
  - **auto**. Let the compiler and/or runtime library decide what is best suited.
  - **runtime**. Deffer the decision at run time by mean of the OMP_SCHEDULE environment variable.

3.2 Data and workload distribution

3.2.1 Domain decomposition

In parallel computing the initial problems must be decomposed into partial tasks that will be assigned to different processing elements (processes or threads). Some techniques are focused on making each processing element work with the data it gets from the previous element and, after performing its task, send it to the next one. This strategy is known as *pipelining*. Nevertheless, RADMeth does not have different processing stages, but the same one applies to different data (CpG sites). In this case the best approach is known as *domain decomposition*.

Domain decomposition is based on two principals. First, each processing element will perform the same sort of calculation and, second, each processing element will work on different data. The advantage it provides with respect to sequential execution is that input data

is distributed among the different elements so the workload is also distributed. In particular, the chosen technique for this program is domain decomposition by blocks applied to MPI processes and dynamic distribution of data among the threads.

Regarding domain decomposition by blocks, given a number n of elements to distribute, and a number p of processes, blocks size can be calculated as:

1. $m = \lceil \frac{n}{p} \rceil$. If n mod p = 0 elements are equally distributed and all processes work with blocks of m elements. This is the optimal case, however it depends on input data and system features, so we cannot rely on it to be reproduced on different systems for different datasets.

2. $m = \lfloor \frac{n}{p} \rfloor$ and $r = n$ mod p. This means that now we are using a block size of m and r elements are left over. This r elements are distributed adding one extra element to the first r processes. The difference among processes is reduced to only one element, in a way that data is better balanced among them.

3.2.2 Adjustment of domain decomposition to the identification of differentially methylated regions

As seen in Section 2.2.4, RADMeth must perform the same operations over the different CpG sites (each one on a different line in the input file). ParRADMeth distributes those CpG sites among the different processing elements and, consequently, the workload is also distributed. In Figure 3.7 the pseudocode of ParRADMeth is shown. The parallel pseudocode is very similar to the sequential pseudocode shown in Figure 2.10, but with some variations related to input and output data to introduce domain decomposition. Concretely, now the proportion table is split into blocks, so each process only computes its corresponding part (Line 2). Also, after doing the required computations for a certain CpG site, the results are not written to the output file, but to an intermediate buffer (Line 23), since some extra synchronization among processes will be needed to write in parallel to the output file. As each process has all the information it needs for the computation phase in its own proportion table block, no need of extra communication is required in this phase, staying the same as it was in the sequential tool (Lines 3-22).

This domain decomposition is implemented in ParRADMeth with two levels of parallelism:

- **Explicit parallelism**. This level uses processes, coordinated by means of message passing. As explained before, at this point each process works over a different block of data, which is kept in memory. The idea behind this level is:

1. For each row of the proportion table, the process creates a full regression structure where it stores coverage and methylation values for each factor. It also creates one reduced regression with the same values as the one seen before, but only for the ones from samples that have the test factor.
2. Then the process checks some extreme cases where we know for sure that the test will not succeed. If any of these cases is detected, then a p-value of -1 will be the result, indicating that the test was not performed.
3. If tests were passed, the process fits the full and reduced regressions.
4. Finally, it checks whether the fitting algorithm succeeds, performs the test to get the p-value and stores the result in a buffer.

- **Implicit parallelism**. At this level, processing elements are threads. A process has initially only one main thread, but it can create more to distribute the workload. In this case, threads are used to distribute the calculation of p-values for different CpG sites of the proportion table assigned to its process. The reason behind this redistribution of the workload is that there is a huge variation among the workload associated to different CpG sites. Because of this variation, a dynamic distribution would be very interesting to balance the workload. However, a dynamic distribution at process level requires many synchronizations and messages that reduce the performance, that is why this is done at thread level only. The main thread also creates a shared structure where the results of all threads can be stored.

3.3 Optimization techniques

ParRADMeth was implemented having in mind the goal of obtaining the best possible performance in terms of execution time while keeping the same accurate biological results as the sequential tool it is based on. This section explains some techniques that were included in the code of the parallel tool in order to improve the performance of the basic hybrid MPI/OpenMP approach explained in Section 3.2.

3.3.1 AllInOneGo File Procesing

The original program works with rows one by one: i.e. it reads one row, it does the computations related to the row, it write the results to the output file, and it goes to the next line. This means that only one row is kept in memory at a time. Although this is the best option in terms of memory requirements, it may not be the best option in terms of execution time, in particular for parallel computing, since it will force the tool to be continuously accessing

```
1  test_factor ← GetTestFactor();
2  proportion_table_block ← ReadPropTableBlock(proportion_table);
3  #pragma omp parallel for
4  foreach CpG site cpg_site in the proportion_table_block do
5      full_regresion ← WriteData(cpg_site);
6      coverage_factor ← 0, coverage_rest ← 0;
7      methylation_level_factor ← 0, methylation_level_rest ← 0;
       /* Agregate coverage and methylation levels for
          samples with and without the test factor       */
8      foreach Sample s in the design_matrix do
9          if s has the test factor then
10             coverage_factor += full_regression[s].coverage;
11             methylation_level_factor += full_regression[s].methylation_level;
12         else
13             coverage_rest += full_regression[s].coverage;
14             methylation_level_rest += full_regression[s].methylation_level;
       /* Calculate the p-value                          */
15     if zero coverage over all case or control samples then
16         p_value ← 0;
17     else if methylation level is identical in all samples then
18         p_value ← 0;
19     else
20         Fit(full_regression);
21         reduced_regression ← CopyWithoutTestFactor(full_regression,
             test_factor);
22         Fit(reduced_regression);
23         p_value ← LoglikratioTest(reduced_regression, full_regression);
24     WriteToOutputBuffer(cpg_site, p_value, coverage_factor, coverage_rest,
         methylation_level_factor, methylation_level_rest);
```

Figure 3.7: ParRADMeth's computation phase pseudocode

the file. That is why ParRADMeth comes with this **AllInOneGo File Processing** technique to process the input proportion table.

This technique consists in reading the whole file at once and keeping the whole proportion table in memory before starting with the computations in order to reduce cache misses. As it will be explained in Section 3.3.2, in a preliminary version of ParRADMeth the main process was in charge of applying this technique and then distributing the data among processes. However, this approach was not the most appropriate, and some optimizations that improve the performance of the technique are shown in the following section.

In addition, the values of the rows of the proportion table were stored as *size_t* in the

sequential version of the program, which seems unnecessarily large because the range of values is limited and small, so they can be represented with just 16 bits. In order to alleviate the memory requirements of the AllInOneGo File Processing, the data type was changed to *unsigned int*. This data type, that usually needs 16 bits, is enough to represent the information of the proportion table. Theferore, although this technique increases the memory requirements compared to the original RADMeth, all datasets used in the experimental evaluation of Chapter 4 fitted perfectly in the memory available in one node of the cluster used for evaluation (64 GB).

3.3.2 Parallel Input/Output

In a preliminary version of ParRADMeth there was a main process that was in charge of reading the input proportion table and distributing the data (blocks of rows) among the other processes, as well as gathering other processes results and writing them to the output file. It means that the phases to read the input and write the output were sequential. After a preliminary benchmarking of this version, I realized that these phases were a bottleneck and significantly degraded the performance of the program. Therefore, these I/O phases were modified in order to include parallel computing on them:

- **Parallel input**. Functions from MPI-IO were used in order to parallelize this input phase, allowing each process to read from a certain offset. The input proportion table has a 2D matrix format where each CpG site is a row and each sample is a column. Each position in the matrix has a pair of coverage and methylation values, as we have seen in Section 2.2.2 . This format is very convenient for parallel processing, since we are interested in having a block of consecutive input rows in each process and that is the way they are physically stored in the file. Therefore, the version of ParRADMeth with parallel input reading makes that each process only reads the block of rows that it will process instead of the whole file. However, this approach is not so straigthforward as not all rows have the same length and the number of rows in a file is not known in advance. Nevertheless, as the size of the file is indeed easy to know with the function *MPI_File_get_size*, this information can be used to distribute bytes among processes in order to create a fair distribution of rows. To avoid one line to be split between two processes, making none of them able to compute it right, an overlapping technique was also implemented: if there is a number p of processes, process $n \in [0, p\text{-}1]$ reads extra bytes to ensure that it will be able to correctly process the row that it may share with process n+1. As was remarked in Section 3.1.1, MPI-IO theoretically obtains better performance that Unix I/O and it has been experimentally checked that this is fulfilled for our specific case. Figure 3.8 shows the pseudocode of this parallel input reading.

```
Input: A string, path_to_input_file, containing the output block of the process
Output: A string, input_block_string, containing the input lines for the process
1  input_file ← MPI_File_open(path_to_input_file);
   /* Figure out who reads what                                    */
2  filesize ← MPI_File_get_size(input_file);
3  start_offset ← CalculateStartOffset(filesize);
4  end_offset ← CalculateEndOffset(filesize);
5  end_offset += overlap;
6  input_block ← MPI_File_read_at_all(input_file, start_offset,
    end_offset);
   /* Avoid half lines at the start and at the end and
      ensure no two process keep the same line in their
      buffers                                                      */
7  true_start ← 0;
8  while input_block[true_start] is not a newline character do true_start++;
9  true_start++;
10 true_end ← end_offset - start_offset - overlap;
11 while input_block[true_end] is not a newline character do true_end++;
12 input_block[true_end + 1] ← '\0';
13 input_block_string = string(input_block[true_start]);
```

Figure 3.8: ParRADMeth's parallel input phase pseudocode

- **Concurrent input**. Even though with the parallel input reading all processes can read from this at the same time, most of the time spent in the input phase is not consumed there, but on string processing to ensure that the input file is a valid one and to parse its contents into the suitable structures. In the pure MPI version of the parallel tool this is easy to do, since data is already fairly distributed. However, in the hybrid version the input bytes are redistributed among threads so they can concurrently process the input string. This redistribution among threads was performed eactly as in the case of processes, i.e., also applying the overlapping technique so there are no split lines left unprocessed between one thread end and the next thread beginning. The parallel and concurrent algorithm for input reading was finally implemented as shown in Figure 3.9.

- **Parallel output**. As was previously discussed in Section 2.2.3, each row in the input proportion table produces a row in the output file, so after a certain process computes all consecutive rows on its input block, it will have a output block of also consecutive rows. This means that the process that gets input block #0 will have to write its output block at the beginning of the output file, and process that gets input block #1 will have to write its output block right after output block #0. Keeping this in mind, the algorithm was implemented as shown in Figure 3.10 to allow processes to write their output in

```
Input: A string, input_block_string, containing the input lines for the process
Output: A buffer, validated_lines, containing an array of preprocessed lines
   /* Figure out who reads what                                           */
 1 block_size ← input_block_string.length();
 2 start_offset ← CalculateStartOffset(block_size);
 3 end_offset ← CalculateEndOffset(block_size);
 4 end_offset += overlap;
 5 thread_block ← CopySubstring(input_block_string, start_offset,
    end_offset);
   /* Avoid half lines at the start and at the end and
      ensure no two threads keep the same line in their
      buffers                                                            */
 6 true_start ← 0;
 7 while thread_block[true_start] is not a newline character do true_start++;
 8 true_start++;
 9 true_end ← end_offset - start_offset - overlap;
10 while thread_block[true_end] is not a newline character do true_end++;
11 thread_block[true_end + 1] ← '\0';
12 thread_block_string = string(thread_block[true_start]);
   /* At this point every thread knows where to start
      processing lines and where to end                                  */
   /* Every thread validates that its lines fits the
      format and stores them on some array separately                    */
13 foreach line l in the thread_block_string do
14 |   ValidateLineFormat(l);
15 |_  thread_lines.append(l);
   /* Each thread places its array of lines in some
      shared buffer so main thread knows where to access
      them all                                                           */
16 threads_shared_buffer[thread_number] ← thread_lines;
   /* After an implicit barrier, main thread joins the
      arrays so lines stay in the same order                             */
17 foreach array a in the threads_shared_buffer do
18 |_  validated_lines.appendArrayOfLines(a);
```

Figure 3.9: ParRADMeth's concurrent input phase pseudocode

parallel:

1. Each process stores its output block in a string buffer.
2. Processes share the length of their output buffers using the MPI function *MPI_allgather*
3. Each process computes the output file offset in which it needs to start writing its output block.

4. Processes open the output file using *MPI_File_open*, write its output block at the same time using *MPI_File_write_at_all* and finally close the file using *MPI_File_close*.

```
Input: A string, my_output_block, containing the output block of the process
Output: The output file with every output block correctly written
1 my_output_length ← my_output_block.length();
  /* Gather all output blocks lengths in every process */
2 output_blocks_lengths ← MPI_allgather(my_output_length);
3 my_write_offset ← CalculateOffset(output_blocks_lengths);
  /* Write in parallel to the output file                */
4 output_file ← MPI_File_open(path_to_output_file);
5 MPI_File_write_at_all(output_file, my_output_block,
   my_write_offset);
6 MPI_File_close(output_file);
```

Figure 3.10: ParRADMeth's parallel output phase pseudocode

Chapter 4

Experimental evaluation

This chapter starts describing the system and the datasets used for scalability tests, as well as the procedure for choosing the best configuration for the final measures. Once the methodology to measure performance is known, it presents the results of the overall evaluation of ParRADMeth, as well as some insights into the performance improvement achieved by each one of the optimization techniques.

4.1 Test environment

4.1.1 System

ParRADMeth's performance was tested on a cluster called *Pluton*[1] and installed at CITIC[2]. This multicore cluster is an heterogeneous system made of nodes connected by a high performance interconnection network (InfiniBand FDR). Figure 4.1 shows cluster's structure. *Pluton* has one only entry point from the outside: its frontend node. From this node code can be compiled or modified, and jobs can be sent to the workload manager, or queueing system, which is in the charge of executing them on compute nodes. The frontend node also works as a NAS server (*Network Attached Storage*) where user files are stored. Compute nodes can access these data remotely through local network.

[1] http://pluton.dec.udc.es/
[2] https://www.citic-research.org/

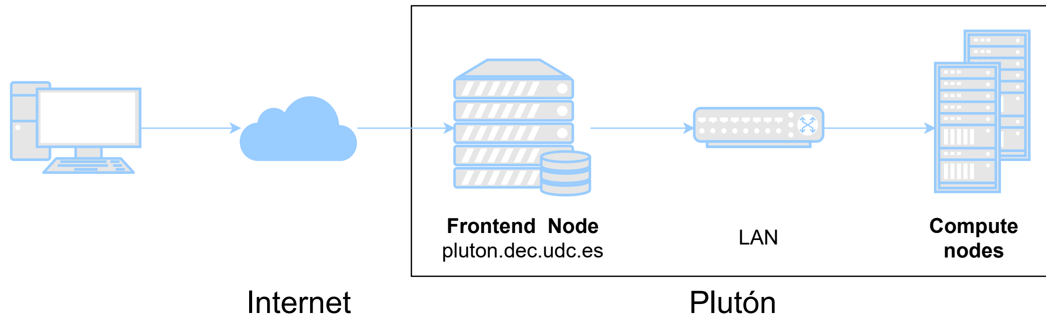

Figure 4.1: *Pluton* cluster general structure

Furthermore, compute nodes provide computing resources: CPUs, memory and accelerators (GPUs). They follow a logical organization by booths where nodes have similar resources. Currently, there are three booths, with the following distribution:

- **compute-0**: 17 compute nodes (compute-0-0 to compute-0-16), making a total of 272 cores, 1088 GB of memory (64 GB per node) and 20 accelerators.

- **compute-1**: 2 compute nodes (compute-1-0 and compute-1-1), making a total of 48 cores, 256 GB of memory (128 GB per node) and without accelerators.

- **compute-2**: 6 compute nodes (compute-2-0 to compute-2-5), making a total of 192 cores, 1536 GB of memory (256 GB per node) and 2 accelerators.

During this bachelor's thesis ParRADMeth's performance was tested on compute booth #0, since it is the one with the highest number of nodes, allowing to test the scalability of the tool. Hardware specifications for compute booth #0 nodes is shown in Table 4.1, highlighting that nodes from this booth have all two octa-core Intel Xeon E5-2660 CPUs, so we can use up to 16 cores per node (up to 32 logical threads taking advantage of *Hyperthreading*).

compute-0-{0-17}	
CPU (model)	2 × Intel Xeon E5-2660 Sandy Bridge-EP (0-0-16) 2 × Intel Xeon E5-2650v2 Ivy Bridge-EP (0-17)
CPU (speed/turbo)	2.20GHz / 3.0GHz (0-0-16) 2.60GHz / 3.4GHz (0-17)
Cores per CPU	8
Threads per core	2
Cores/Threads per node	16 / 32
Cache L1/L2/L3	32KB / 256KB / 20MB
Memoria RAM	64GB DDR3 1600Mhz
Discos	1 × HDD 1 TB SATA3 7.2K rpm
Redes	InfiniBand FDR and Gigabit Ethernet

Table 4.1: compute-0 nodes specification

Paying attention to Table 4.1 it can be seen that nodes are interconnected by a LAN network. This is a GigabitEthernet network, with a maximum throughput of 1 Gbps. In addition, every node has an InfiniBand FDR network interface. InfiniBand is a high performance interconnection network, that is, a very high throughput and a very low latency network, with many specifications. The one being used in this cluster is the FDR specification, allowing to reach a throughput of 56 Gbps and latencies of 1-2 µs.

As it was previously mentioned, we can not access *Pluton*'s compute nodes directly, but through a workload manager, also known as a job scheduler. This software is in charge of assigning cluster's resources to users according to their needs. Users send their jobs to the workload manager from the frontend node indicating the required resources. Then, the job scheduler executes the jobs when it considers it appropriate, always trying to satisfy the requirements of the highest possible amount of users. *Pluton*'s job scheduler is the Slurm Workload Manager[3], in version 19.05.2.

Jobs are the Slurm's execution unit, and they can be either *batch* jobs (executed in background) or *interactive* jobs. When sending a job to the workload manager, Slurm uses parameters to know the required resources. Next, the most relevant ones during testing are shown:

- **Memory**. It indicates the memory required for the job. It can be indicated in total or by core. It is important to keep in mind that this amount is limited by the physical RAM memory.
- **Number of nodes**. It indicates the number of nodes that must be used.

[3] https://slurm.schedmd.com/

- **Number of cores**. It indicates the number of cores that are required by the job for executions with implicit parallelism. It can be indicated in total or by process.
- **Exclusive access**. This option prevents Slurm to execute jobs from other users on nodes where our jobs are being executed. It means that all cores and memory are assigned to our job. It is very useful for performance tests, since it prevents other jobs to interfere with ours, which may lead to inaccurate results.
- **Execution time**. It is compulsory to indicate the estimated execution time for the work. It is important to try to make a good estimation, since this way we contribute to a more efficient functioning of the task scheduler. On the other hand, to guarantee the quality of the service and a good distribution of the hardware among all the users of the cluster, there is a maximum limit of 72h for every job.

Syntax is not specified since it can be found checking Slurm documentation[4].

Another main task of the frontend node, as we previously mentioned, is program compilation. *Pluton* has the *Lmod*[5] tool installed to administrate most of the software with the idea of supporting different versions of the same package. In particular, the following packages were used during this bachelor's thesis.

- **CC**. This module brings a C/C++ compiler, besides some standard libraries, so it is needed for both compilation and execution of the tool. In particular, during this bachelor's thesis the GNU implementation (gcc)[6] was used in version 8.3.0.
- **MPI**. This library, described in Section 3.1.1, specifies the syntax of routine for message passing, and it is also needed for both compilation and execution. In particular, we used OpenMPI[7] implementation, version 3.1.4.
- **GSL**[8]. It is a numerical library for C and C++ programmers that provides a wide range of mathematical routines. It is a dependency from the original sequential tool. Specifically, we used version 2.6.

In addition to those packages, since RADMeth is at the moment part of the MethPipe pipeline, some of their components are RADMeth's dependencies, in particular, *common* and *smithlab_cpp* submodules. Since they are not standard libraries, it is not usual that they are instaled on clusters, so it was part of my work to install those submodules on the cluster. The

[4] https://slurm.schedmd.com/documentation.html
[5] https://lmod.readthedocs.io/
[6] https://gcc.gnu.org/
[7] https://www.open-mpi.org/
[8] https://www.gnu.org/software/gsl/

full pipeline is available in GitHub, in SmithLab's public repository[9]. In particular we used version 4.1.1.

4.1.2 Datasets

Performance tests measure the improvements that parallel techniques and proposed optimizations bring to the tool. Datasets with real biological data and different characteristics have been used in order to check the performance of ParRADMeth in different scenarios. It should be noted that RADMeth is part of a pipeline and the files it needs as input are the output of the previous stage of the pipeline. That said, input datasets are not directly available, but several ".meth" files that have to go through the pipeline to become proper input files for the tool can be found. Next a description of every dataset employed during this bachelor's thesis will be given . In addition, Table 4.2 shows for each dataset the number of CpG sites, the number of samples and the size of the input proportion table given to the tool.

Dataset	CpG sites	Samples	Size
Test Dataset	2.802.194	6	130 MB
Akalin 2012	28.670.426	2	823 MB
Heyns 2012	28.299.639	2	869 MB
Berman 2012	28.149.963	2	880 MB
Hansen 2014	28.217.449	6	1,4 GB

Table 4.2: Datasets specification

- **Test Dataset**. Small dataset that comes with the tool for testing purposes. It has been used to validate the parallel tool, that is, it gets the same biological results as the original sequential tool. This dataset is composed by six ".meth" files, each one with size of 88.5 MB, that become a proportion table with size 130 MB. The sequential tool needs 8 seconds for the input phase, 3075 seconds for the computation phase and 12 seconds for the output phase (0,64% I/O).
- **Akalin 2012** [27]. This dataset is used to compare methylomas of HCT116 cells with those of cells cloned without DNMT1 and DNMT3b. Two compressed ".meth" files, each one with a size of 700MB, contain the information needed for the experiment. Once joined, they become a proportion table with size 823 MB. RADMeth needs 180 seconds in the input and output phases and 10523 seconds in the computation phase (1,68% I/O).
- **Heyns 2012** [28]. Heyns dataset compares WGBS between centenarians and newborns. It is composed by two ".meth" files, one representing a centenarian and another

[9] https://github.com/smithlabcode/methpipe

one representing a newborn. Both become a proportion table with size 869 MB. The sequential tool takes 175 seconds in the input and output phases, representing a 0,93% of the runtime of the tool, and 18665 seconds in the computation phase.

- **Berman 2012** [29]. Dataset composed by two compressed ".meth" files used to compare individuals with colorectal cancer to healthy individuals. After decompression they form a 880 MB proportion table. RADMeth takes 61 seconds in the input phase, 42359 seconds in the computation phase and 117 seconds in the output phase (0,42% I/O).

- **Hansen 2014** [30]. This dataset is used to compare cells immortalized with EBV virus with others activated with CD40. It is the largest dataset, composed by six compressed ".meth" files each with a size of 750 to 850 MB after decompression that form a proportion table with size 1,4 GB. The sequential tool takes a 0,42% of its runtime in input and output phases, 78 seconds in the input phase and 116 seconds in the output phase. It also takes 45703 seconds in the computation phase.

4.2 Previous scalability concepts

Before starting with the tests, some basic concepts will be explained to understand the performance metrics that will be shown:

- **Processing element**. A processing element is the minimum unit of computation. It should be pointed out that we refer to physical units and not to logical ones, since, even if the second ones allow us to take advantage of the first ones, the physical units are those that provide the computing power. For example, we can execute two threads on a single core, but it will not provide the double computing power since they share the same physical resource. Said that, we refer as processing elements to the cores employed during program execution.

- **Speedup**. It is the metric that measures the improvement in execution of the parallel tool in relation with the sequential one. It can be calculated for *np* processing elements with the following formula:

  $$speedup(np) = \frac{T_{sequential}}{T_{parallel}(np)}$$

  where $T_{sequencial}$ is the execution time of the sequential tool and $T_{parallel}$ is the execution time of the parallel one when using *np* processing elements. Ideally speedup takes values between 0 and *np*, but three cases can happen:

  – *speedup(np)* ∈ (0,1]: Parallel execution does not improve execution time. If both times are the same the value of the speedup is equal to one.
  – *speedup(np)* ∈ (1,*np*]: Parallel execution improves execution time. Ideal parallelization will give a speedup value of *np*, that is, sequential execution workload is perfectly balanced among processing elements.
  – *speedup(np)* > *np*: Parallel execution exceeds the maximum theorical improvement. This scenario is known as superlineal speedup and can be achieved because of different factors, as it can be a better memory management, reducing cache misses.

- **Efficiency**. This metric gives an idea of the behavior of the parallel tool, independently of the number of processing elements that are being used on execution. It takes values between 0 and 1 (perfect parallelism), even though it can take higher values when speedup is superlinear. It is usually shown in percentage by multiplying its value by a hundred. It can be calculated, for *np* elements, with the following formula:

$$efficiency(np) = \frac{speedup(np)}{np} = \frac{T_{sequential}}{np * T_{parallel}(np)}$$

- **Scalability**. On an ideal case efficiency will keep constant independently of the number of processing elements, but this does not usually happen on real scenarios. Scalability is defined as the capacity of keeping efficiency when the number of processing elements increases.

4.3 One node test

The hybrid MPI/OpenMP implementation included in ParRADMeth allows the user to choose among different configurations of number of processes and threads in order to obtain the best performance in different architectures. Before starting testing the scalability of the tool it is necessary to select the configuration that gives the best performance in one single *Pluton*'s node and this configuration will be assumed as the best when increasing the amount of nodes. As it has been seen so far, the parameters that can be modified are threads to process ratio, use of *Hyperthreading* and schedule policy in the main *for* loop (both, *Hyperthreading* and schedule policy were described in Section 3.1.2).

Talking about threads to process ratio, it is important to remember that nodes that are used for experimental evaluation have two CPUs with eight cores each. That is, in total in one node there are 16 cores. As the goal is to take advantage of system features to the maximum, the highest number of processing elements available will be used. Consequently, for one node the

equation *nProcesses * Threads_by_Process* = 16 must be true. Moreover, *Hyperthreding* can be used, allowing the use of two threads on each core, so for one node using *Hyperthreding* the equation *nProcesses * Threads_by_Process* = 32 must be true. In spite of that, these are logical threads, so the number of processing elements keeps being 16.

Each row of the input file takes a different time to be processed. On the one hand, some rows are on those extreme cases that do not need to execute the fitting phase (see Figure 3.7, lines 14-17). On the other hand, the fitting algorithm does not take the same time to execute over different data. This leads to situations where even if the number of lines in the input dataset is fairly distributed workload is not. Since threads of the same process can all access every row and there is no need to physically redistribute rows to make each thread work with more or less of them, the higher the threads-by-process ratio, the better the workload balance.

4.3.1 Selection of the best schedule

This workload balance concept is important when thinking about the impact a schedule has on the performance of the tool. Concretely, choosing the proper scheduling scheme is key in order to get the best possible balance in terms of compute time among the threads on the same node. So the better the schedule policy adapts to differences in rows compute time, the better the performance it will deliver and also the more consistency in giving good performances among different datasets.

During these tests, every available dataset was used, as memory is not a concern and they will give a good vision of how consistent a schedule policy performance is.The three schedules were tested (static, dynamic and guided) with one process and 1, 2, 4, 8 and 16 threads without *Hyperhreading* and 32 threads with *Hyperhreading*. In Table 4.3 execution times are shown in different cores (C), with and without the use of *Hyperthreading* (Ht). It can be seen that *dynamic* schedule gets the best performance for almost every experiment. The only exception are executions with only one core (without parallel computation) and for the test dataset, which is too small to raise any conclusion. However, in almost every dataset, specially in the largest ones, it is closely followed by *guided* schedule. In Akalin dataset tests, where workload balance is critical, *dynamic* schedule obtained much better results than both *guided* and *static* schedules. This happens because this dataset has an isolated block of very high computational demanding rows, and when using *guided* or *static* schedules this whole block is assigned to the same thread.

Figures 4.2, 4.3, 4.4 show *speedups* for *static*, *dynamic* and *guided* schedules, respectively, for every number of cores tested. As was previously discussed, on Akalin dataset *dynamic* schedule gets a much better performance than the other two schedules, meaning that on critical situations *dynamic* schedule is the only one able to improve the performance by using

Dataset	Schedule	1C	2C	4C	8C	16C	16C (Ht)
Test Dataset	Static	2429,87	1222,98	646,802	346,816	177,645	134,669
	Dynamic	2452,65	1241,6	645,295	345,663	178,65	134,634
	Guided	3092,73	1548,49	800,794	416,765	218,926	162,896
Akalin 2012	Static	10779,5	5471,83	3226,06	2320,45	1709,09	1740,41
	Dynamic	8498,7	4268,41	2267,63	1210,38	635,544	498,651
	Guided	8482,31	4315,81	2870,94	1962,43	1552,37	1355,65
Heyns 2012	Static	20853,4	9576,94	4982,45	2580,24	1420,36	1068,81
	Dynamic	18702,1	9403,89	4751,03	2465,8	1343,91	1016,51
	Guided	18855,7	9666,95	5029,72	2564,63	1441,23	1123,16
Berman 2012	Static	42185,8	21573,8	11503	5878,87	3364,37	2536,54
	Dynamic	42412,6	16697	8771,63	4682,22	2408,69	1829,07
	Guided	42603,1	16991,4	9249,38	4868,46	2553,51	1967,5
Hansen 2014	Static	45686,80	23143	12213	6419,48	3610,04	2753,90
	Dynamic	46156,80	23069,10	11703,30	6037,69	3226,99	2424,93
	Guided	45927,60	23298,90	12280,40	6606,22	3538,72	2734

Table 4.3: Execution time from the hybrid tool in one node using different schedules (in seconds)

32 threads. So, in the following tests the hybrid version of the tool will always use *dynamic* schedule.

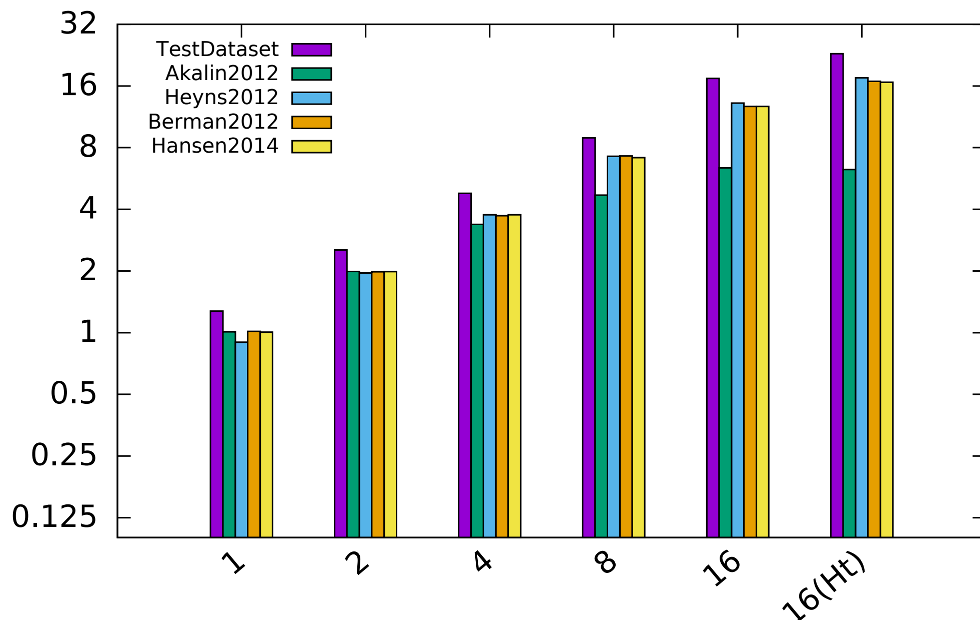

Figure 4.2: Static schedule's speedups. Processing elements (cores) vs. speedup over RADMeth

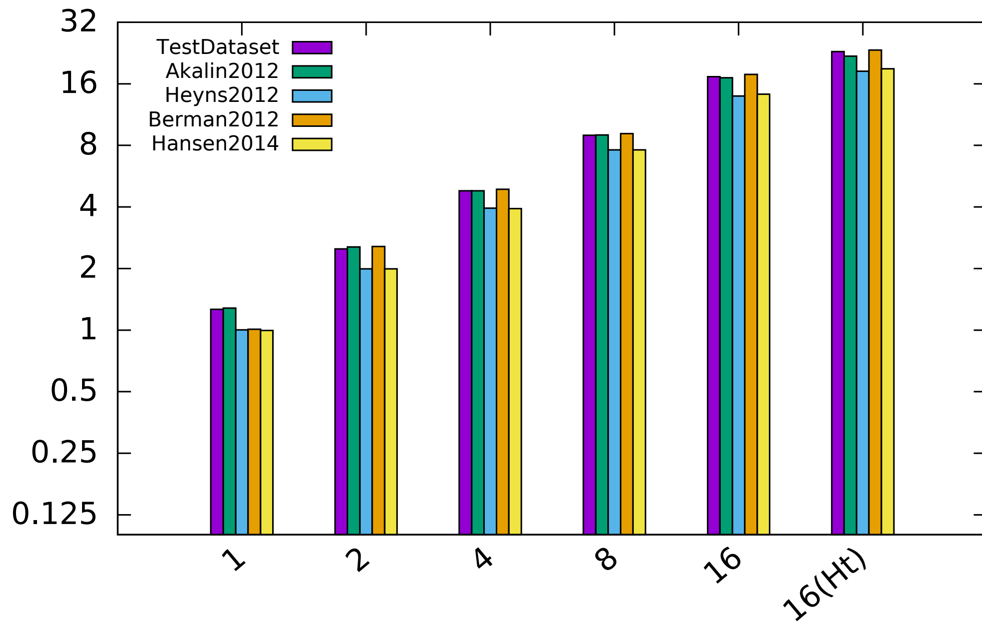

Figure 4.3: Dynamic schedule's speedups. Processing elements (cores) vs. speedup over RAD-Meth

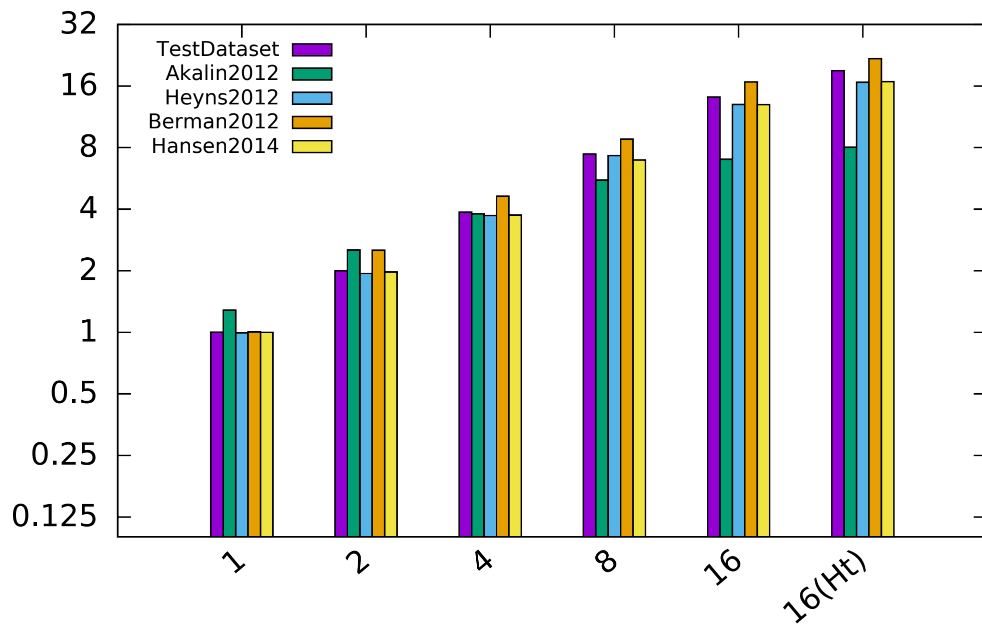

Figure 4.4: Guided schedule's speedups. Processing elements (cores) vs. speedup over RAD-Meth

4.4 Scalability tests

This section will show the results obtained during performance tests using several *Pluton*'s nodes. In every test execution time was calculated for RADMeth, which will be the basis for the calculation of speedups, and for ParRADMeth using 1, 2, 4, 8 and 16 nodes, with the configuration that was explained in the previous section when measuring the performance of the hybrid tool. In particular, the configuration used was: one process for each node, 16 cores for each process (16 threads without *Hyperthreading*, 32 logical threads when using it) and *dynamic* schedule. As shown before, the largest dataset has a size of 1,4 GB, so all of them fit in memory in every experiment.

Execution times measured during these tests are included as tables and bar charts are also included to show speedups. It should be noted that bar charts follow a logarithmic scale base two on the vertical axis.

4.4.1 Input/Output tests

First of all, it is important to mention that during this bachelor's thesis two different approaches to deal with the Input/Output bottleneck were considered. RADMeth performs I/O operations by means of streams, so the first considered approach was to extend this functionality to be able to use streams to read and write to files in parallel among different processes. After some research routines available in the MPI-IO library were considered as a new approach. MPI-IO seems to be a more natural option to implement parallel I/O in ParRADMeth, since it is specifically designed for HPC. In addition, some studies show that MPI-IO provides better performance [25, 26].

With both approaches in mind, it was important to measure their performance and scalability to decide which one fits better into ParRADMeth. Since there is no difference among datasets that concerns the scalability of any approach in I/O phases and for some datasets the time spent during these phases was very small (16 seconds of sequential execution) only the largest dataset, Hansen 2014, was used to compare both methods. Table 4.4 shows the execution times of the I/O phases using the two approaches. Both of them significantly reduced execution times, to less than five seconds in both cases, but MPI-IO was proved to reach lower execution times that the Streams I/O approach.

Approach	1C	16C	32C	64C	128C	256C
MPI-IO	99,0562	9,24324	7,4917	3,77	3,45	2,75053
Streams I/O	158,328	13,8453	8,7971	6,72007	4,6716	3,52449

Table 4.4: Execution times in the I/O phases for Hansen dataset

Figure 4.5 shows speedup values for both methods. We can see that both approaches are

able to scale, but MPI-IO consistently reach higher speedup values. These conclusions apply to both the pure MPI and the hybrid MPI/OpenMP versions, so all the results shown in the following sections were taken with the tool using the MPI-IO approach.

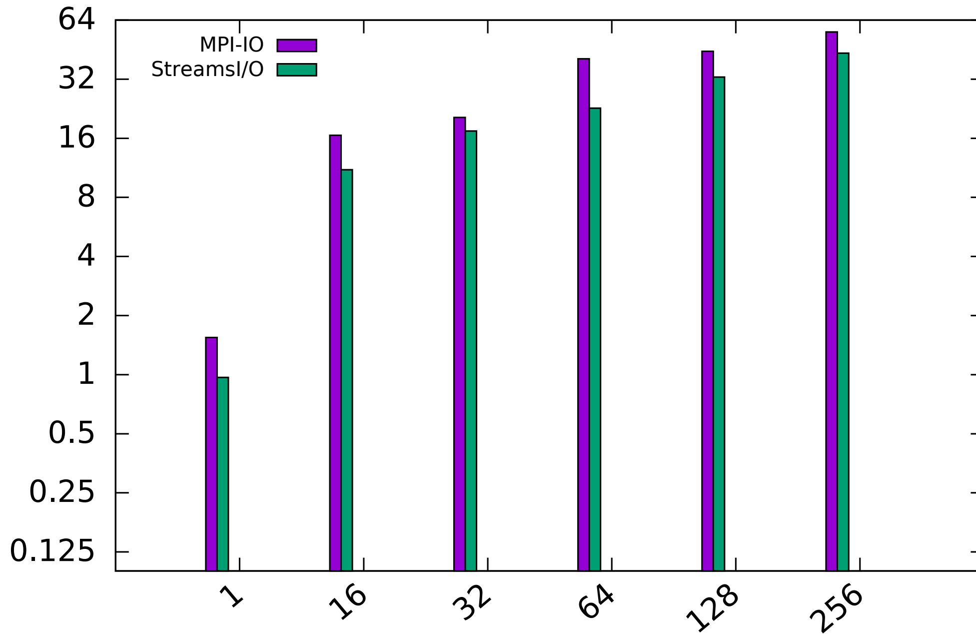

Figure 4.5: Speedups in the I/O phases for the Hansen dataset. Processing elements (cores) vs. speedup over RADMeth

4.4.2 ParRADMeth, pure MPI version

In Table 4.5 execution times of ParRADMeth for different number of cores (C) using only MPI processes are shown. It can be seen that the pure MPI version of the tool manages to significantly reduce the execution time of the algorithm for some datasets. However, the runtime for Akalin dataset stays almost unchanged from 16 processes forwards, even though it is reduced compared with the one from sequential execution. This version does not use threads, just processes, so in each test it will use one process per processing element available.

Dataset	**16C**	**32C**	**64C**	**128C**	**256C**
Test Dataset	218,117	109,643	59,4741	29,8572	16,8326
Akalin 2012	1806,011	1713,63	1676,670	1521,06	1420,55
Heyns 2012	1357,88	700,674	427,978	259,292	178,848
Berman 2012	3312,92	1778,87	1036,47	668,184	492,275
Hansen 2014	3571,35	1978,08	1202,22	823,506	639,506

Table 4.5: Execution time for the pure MPI version (in seconds)

In general terms (forgetting the Akalin dataset), we observe that the pure MPI tool reduces its runtime when incrementing the number of processes. However, this reduction is less significant for the largest datasets. Figure 4.6 shows how the tool scales for the different datasets. Two main conclusions can be extracted from these results. On the one hand, the static workload distribution makes this version of the tool unable to scale well when facing critical scenarios such as Akalin dataset, where workload is by default completely unbalanced. On the other hand, even for non-critical scenarios the static workload distribution makes the tool unable to scale as well as it should.

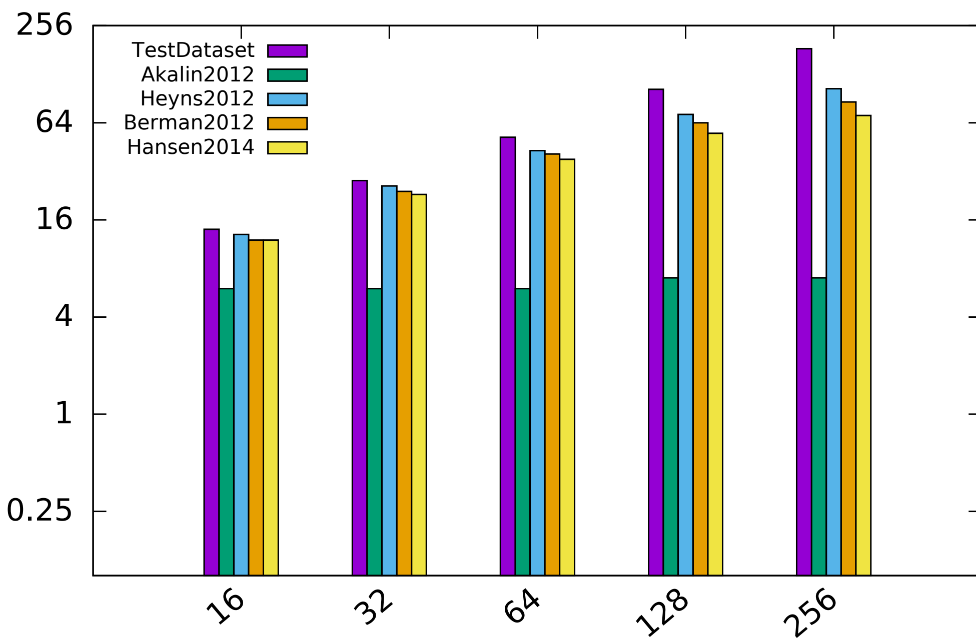

Figure 4.6: Speedups for the pure MPI version. Processing elements (cores) vs. speedup over RADMeth

4.4.3 ParRADMeth, hybrid version without *Hyperthreading*

In Table 4.6 execution times in different number of cores (C) are shown. The hybrid version of the tool highly reduces the execution time of the algorithm for every dataset when increasing the number of processing elements, including Akalin dataset, which, as it was previously explained, supposes the worst case scenario in terms of workload balance. This version uses threads, not only processes, with the configuration that already discussed in Section 4.3, so in each test it uses one process and 16 threads per node.

Figure 4.7 shows how the tool scales for the different datasets. Differently to the pure MPI implementation, this version of the tool is able to scale well in every scenario, and it reaches

Dataset	16C	32C	64C	128C	256C
Test Dataset	178,65	111,30	55,9983	28,6928	14,997
Akalin 2012	635,544	399,291	378,904	251,844	147,719
Heyns 2012	1343,91	690,48	528,959	298,195	150,909
Berman 2012	2408,69	1525,18	974,335	524,031	273,247
Hansen 2014	3226,99	1631,63	1013,7	557,177	297,574

Table 4.6: Execution time from the hybrid version without using *Hyperthreading* (in seconds)

the highest speedups for the largest datasets. Moreover, it avoids the problem of workload imbalance in the Akalin dataset, and the speedup continuously incresases up to 16 nodes (256 processing elements).

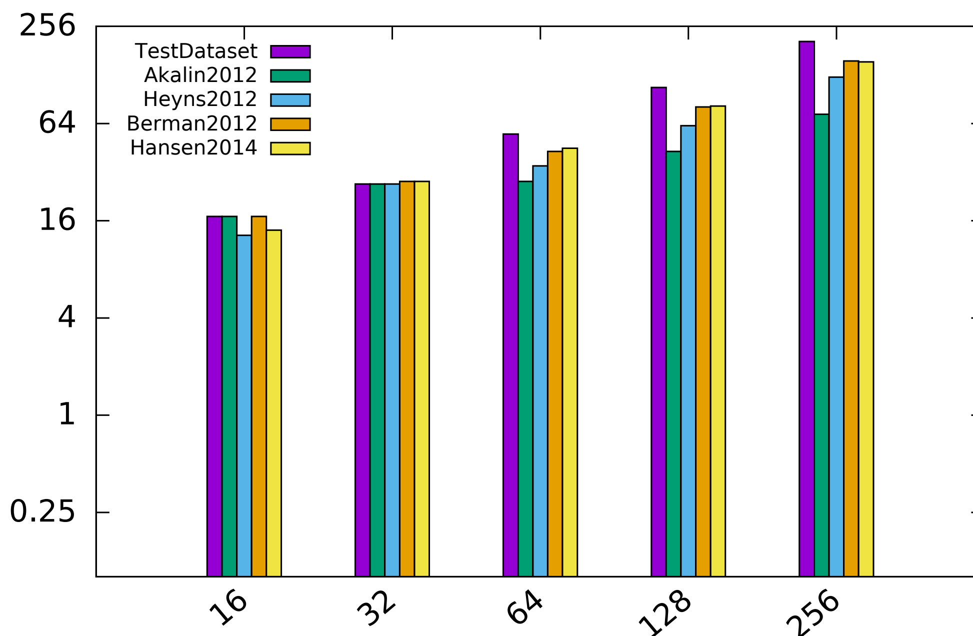

Figure 4.7: Speedups of the hybrid version without using *Hyperthreading*. Processing elements (cores) vs. speedup over RADMeth

4.4.4 ParRADMeth, hybrid version with *Hyperthreading*

As explained in Section 3.1.2, *Hyperthreading* is a technique that allows to execute several threads simultaneously on the CPU. ParRADMeth was tested on Intel Xeon E5-2660 CPUs, that allows the execution of two simultaneous threads at a time. This means that using the same physical processing elements, we can have twice logical threads processing the data. The runtime for all the datasets is significantly reduced compared to the execution without *Hyperthreading*, getting times of less than five minutes in all of them, as it is shown in Table

4.7, where execution times for different number of cores (C) are shown. An exceptional case can be seen for test_dataset using the 16 nodes, where increasing the number of nodes worsen tool's performance, but it just happened because execution times were tiny.

Dataset	16C	32C	64C	128C	256C
Test Dataset	134,634	82,1206	42,0951	21,2393	41,1531
Akalin 2012	498,651	304,764	326,554	215,101	121,354
Heyns 2012	1016,51	514,386	440,915	253,301	136,471
Berman 2012	1829,07	1133,31	780,041	421,23	219,022
Hansen 2014	2424,93	1228	803,705	444,672	242,977

Table 4.7: Execution time from the hybrid version when using *Hyperthreading* (in seconds)

Figure 4.8 shows the speedups for the hybrid version of the tool are shown with *Hyperthreading*. Except for the case of the extremely small testing dataset, this version of the tool proved to have a great scalability in all scenarios. Speedups of up to 195 are reached, and the larger the dataset, the higher the speedup achieved, proving that the tool scales better for bigger datasets.

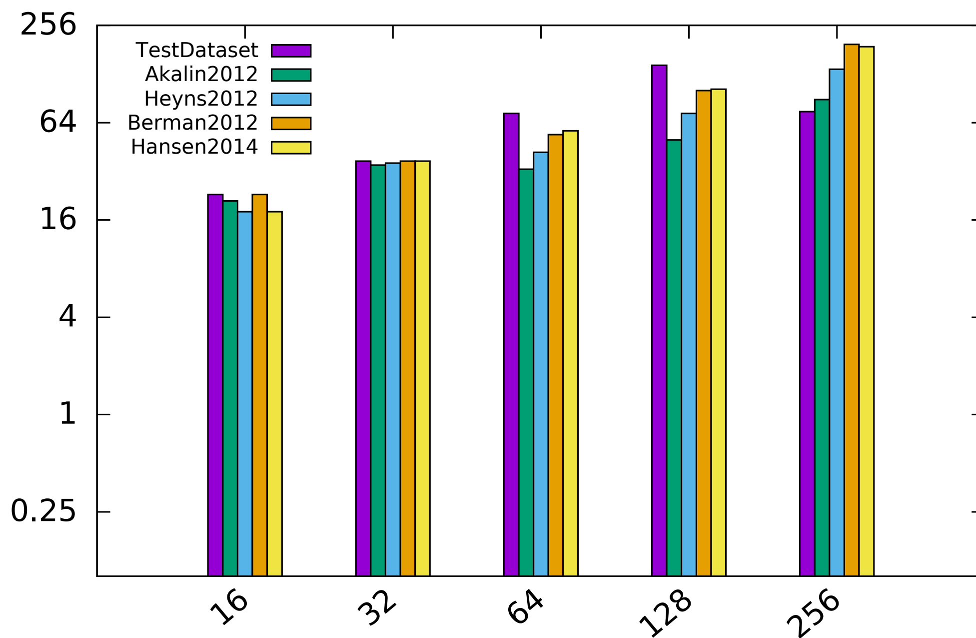

Figure 4.8: Speedups for the hybrid version when using *Hyperthreading*. Processing elements (cores) vs. speedup over RADMeth

4.5 General conclusions

Thanks to this experimental evaluation very powerful conclusions can be extracted from the behaviour of every version of ParRADMeth. Also, because of the use of very different datasets we can get a wide coverage of these behaviours. For instance, the Akalin dataset allows ParRADMeth to be tested in a critical scenario, where getting a good workload balance is extremely difficult. The main idea extracted from this test was that the factor that will determine the scalability and the performance of the tool is how good a version of the tool manages to balance the workload

The first conclusion obtained is that the *dynamic* schedule is the best performer for the hybrid tool, no matter the size of the dataset or the well pre-distributed it is. This proved the importance of allowing ParRADMeth to distribute the workload dynamically, in contrast with the pure MPI tool or the hybrid tool using the *static* schedule, which were never able to reach a speedup value of more than eight for the Akalin dataset, even using up to 256 processing elements. This happens because threads are our main mechanism lo balance the workload during execution time. Even though the *guided* schedule also distributes the workload dynamically it was probed to do this task worse than the *dynamic* one.

Thanks to this workload balance advantage, the hybrid version of the tool proved to reach lower execution times than the pure MPI one, and to scale better in all situations, specially in critical ones, even without using *Hyperthreading*. The hybrid tool also allows us to use *Hyperthreading*, which is able to obtain execution times even ten times lower than the ones we get from the pure MPI approach. Table 4.8 provides a summary of these results for the different versions of the tool.

Dataset	**RADMeth**	**hybrid ParRADMeth**		
	1 core	**1 core**	**256 cores**	**256 cores (Ht)**
Test Dataset	3097,32	2335,78	14,997	41,1531
Akalin 2012	10886,90	8186,47	147,719	121,354
Heyns 2012	18736,50	13968,50	150,909	136,471
Berman 2012	42800,80	31629,60	273,247	219,022
Hansen 2014	45931,20	34224,90	297,574	242,977

Table 4.8: Summary of execution times for different versions of the tool (in seconds)

So, from a general point of view, pretty satisfying results were obtained, especially due to two reasons:

- On the one hand, interesting upgrades were included in the tool, so that it consistently reached superlinear speedups with the hybrid tool in one node, even without using *Hyperthreading*.

- On the other hand, dynamic workload balance was included in the parallel tool, so it is able to scale even in the worst case scenarios. This allows ParRADMeth to reduce execution times to less than five minutes even for datasets that need more than 12 hours to execute with the sequential program. In addition, the parallel tool also gets a better scalability when the larger the dataset is.

Chapter 5
Planning and organization

This chapter gives a detailed overview of the organization of the project. First, the different phases implemented in the project are presented, and afterwards, time and cost of each step are specified.

5.1 Project planning

The project follows an incremental development model, starting with a basic parallel implementation with the whole functionality and then including different optimization techniques in subsequent development steps. Therefore, each of these steps alternate performance evaluation to detect the bottlenecks and the development of optimization techniques that can minimize their impact.

5.1.1 Phase 1: Analysis of the state of the art and understanding of the original tool

First, a general background analysis of the subject was completed. On the one hand, I started reading the original work paper and analyzing the original code of RADMeth [7], as well as the way to execute it. On the other hand, I searched and studied other methods for the identification of differentially methylated regions in genomic analyses. Concretely, I could get a better understanding of the field and the different methods. Finally, I made the decision that the parallel algorithms would be implemented using MPI and OpenMP.

5.1.2 Phase 2: Design and implementation of a basic parallel tool with the whole functionality

Once the code of the original tool was understood, the next step was to design and implement a basic parallel algorithm, only by using MPI. Even though the execution times did not show the desired performance it allowed to start investigating and testing these issues.

After analysing the results of these tests, several bottlenecks were identified and so, different strategies were designed to reduce its impact of even eliminate them.

5.1.3 Phase 3: Addition of optimization techniques

After analysing potential bottlenecks identified during previous tests the following optimization techniques were added :

1. AllInOneGo File Processing, since the whole input file fits in memory and it improves the performance of the tool (see Section 3.3.1).
2. Change of the datatype of the buffers from *size_t* to *unsigned_int* to reduce memory consumption.
3. Design and implementation of a parallel mechanism for input reading (see Section 3.3.2).
4. Design and implementation of a parallel mechanism for output writing (see Section 3.3.2).
5. Design and implementation of a hybrid version of the computational phase using OpenMP.
6. Design and implementation of a concurrent mechanism for input reading (see Section 3.3.2).

5.1.4 Phase 4: Performance evaluation

Once the final pure MPI and hybrid versions were implemented, the next step was to test the tool's performance and scalability. As shown in Chapter 4, these tests consisted of two parts. On the one hand, it was necessary to find the best combination of MPI processes, OpenMP threads and OpenMP schedule policy for one node. On the other hand, it was necessary to measure execution times using that configuration on more nodes.

5.1.5 Phase 5: Documentation and report writing

During all the life-cycle of the project all the steps taken have been documented. During the early stages, brief annotations were made to document objectives, advances, and issues detected. For some particular ideas, such as parallel and concurrent reading phase, more detailed documentation was written. The last step was to collect up all the documentation and performance results into this report.

5.2 Project metrics

In addition to the organization of the project, we have also taken into account the time and resources spent as well as their theoretical costs.

5.2.1 Time

The project duration was nine months, from September 2020 to May 2021. Nevertheless, we must take into account that in some periods the project was suspended due to lack of time because of the regular studies.

Table 5.1 shows a breakdown of the different phases of the project into smaller tasks and its estimated duration. These tasks are then arranged in a Gantt diagram as seen in Figure 5.1.

The most relevant dependencies among tasks are the following, grouped by type:

- **Finish-to-Start dependencies (FS)**. Dependencies indicating that the predecessor task must finish before the successor can start. This is the most common type of dependency in project management and the real world. Dependencies of this type are:
  - **Tasks a, b, c** and **d** $\rightarrow$ **Task e**. Researching about the original tool and the state of the art sets the beginning of the project, that is why, until I understand properly the state of the art and the way RADMeth works, starting the parallel implementation makes no sense.
  - **Tasks e** $\rightarrow$ **Task f**. Bottleneck analysis can not start until the basic implementation is completely implemented.
  - **Tasks g, h, i** and **j** $\rightarrow$ **Task k** and **l**. **Task g, h** and **i** represent the implementation of the different optimization techniques and **Task j** represent the start of the experimentation phase by developing the scripts to execute test in Pluton. We can not start testing until the final versions of the tool are completely implemented and the tests scripts are not developed.
  - **Tasks m** and **n** $\rightarrow$ **Task o**. In this report documentation about code implementation (**Task m**) and experimental results (**Task n**) is gathered, so the report will not be started until all previous documentation is finished.
- Start-to-Start dependencies (SS). Dependencies indicating that a task cannot start before the predecessor task starts. This dependency does not indicate that both tasks have to start at the same time. Dependencies of this type are:
  - **Tasks e** $\rightarrow$ **Task m**. **Task m** creates a document about how the code was implemented, so it can not start before the code implementation itself (**Task e**) starts.

– **Tasks l** → **Task n**. Following the same pattern as before, **Task n** creates a document about the results gathered during tool's experiments, and those results are generated and gathered during **Task n**.

Finally, its important to highlight that **Tasks h** and **i** were not planned on the beginning and they appeared as result of the bottleneck analysis (**Task f**). That is the reason why no relation is created to model when they should start. However, it we knew in advance that this optimization techniques were going to be implemented a **Finish-to-Start dependency** from the task where both of them are detected would be a the suitable way to represent it.

Phase	Task	Duration (h)
1	a. Review original article	6
	b. Analysis of the original code	14
	c. Compiler installation and execution of the original code	10
	d. Information research	15
2	e. Basic parallelization with the whole functionality	60
3	f. Bottlenecks analysis	15
	g. Addition of OpenMP	20
	h. Development of AllInOneGo File Processing technique	5
	i. Development of Parallel Input/Output	60
4	j. Script development to use during performance tests	5
	k. One node configuration research	15
	l. Performance tests and result gathering	40
5	m. Code implementation documentation	10
	n. Analysis results documentation	15
	o. Report elaboration	80
	Total	370

Table 5.1: Total hours inverted in the project detailed by task

5.2.2 Budget

Three people were involved in the development of the project: the student and two tutors. The student was responsible for the software development and result gathering, while the tutors were responsible for the choice of the topic and the supervision of the project. The result analysis was done by all of them.

Table 5.2 shows an approximation of the amount of hours invested in the project by each of the resources. The hours assigned to the tutors are based on the amount of meetings, mail conversations, problem troubleshooting and document reviews.

Finally, Table 5.3 shows estimated cost per hour of each resource, and the total cost of the project.

Resource	**Hours**
Student	370
Tutor 1	35
Tutor 2	25

Table 5.2: Total hours invested in the project by each resource

Resource	**Cost per hour (€/h)**	**Hours**	Cost(€)
Student	40	370	14.800
Tutors	60	60	3.600
Total			**18.400**

Table 5.3: Total costs of the project detailed by resource

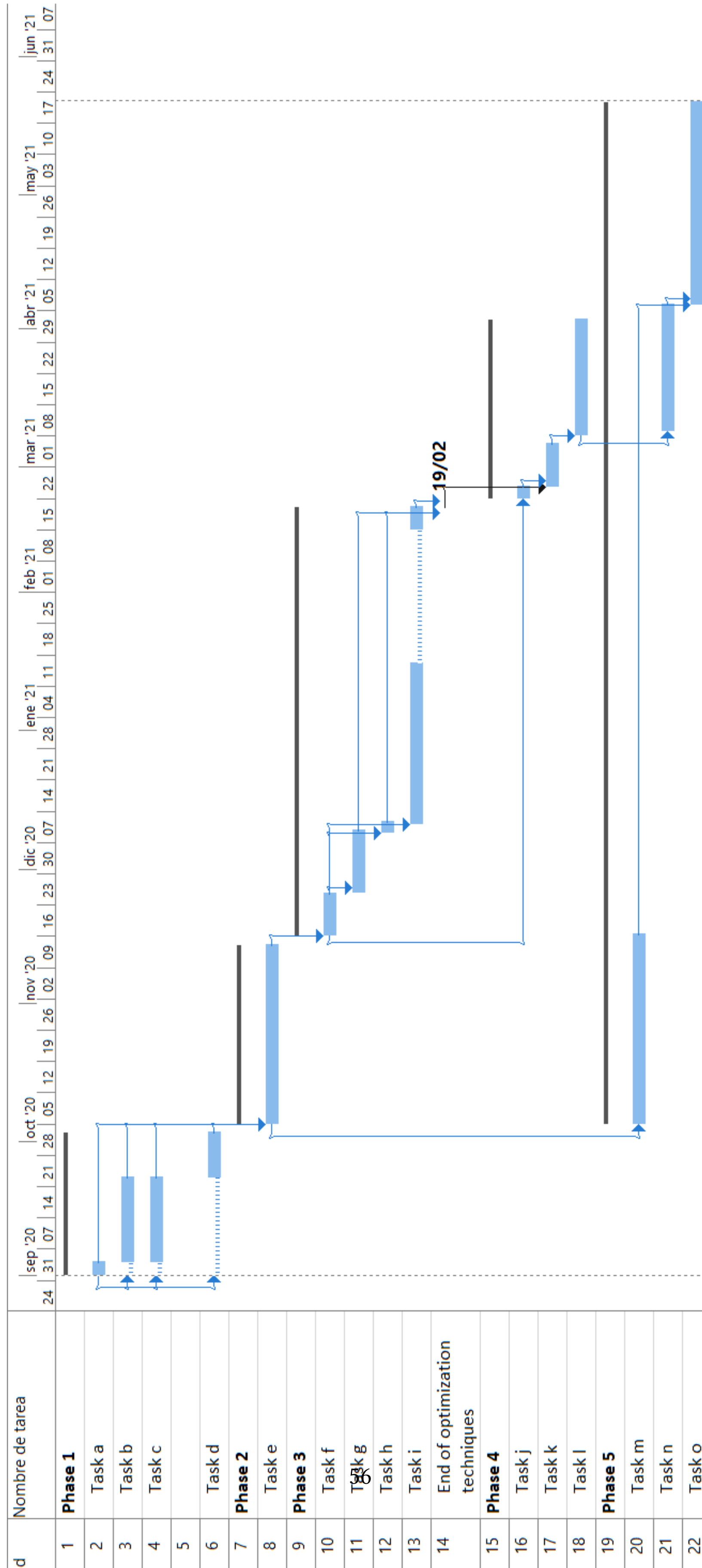

Figure 5.1: Gantt chart showing the arrangement of the different phases of the project

Chapter 6

Conclusions

THIS chapter brings the conclusions of the project, its relation with the bachelor degree and the main future work lines that can be followed after the thesis.

6.1 Conclusions

The main goal of this bachelor's thesis was the development of a parallel tool for the identification of differentially methylated regions in genomic analyses, allowing us to release an open source product that scientific community can take advantage of and dig more deeply into the fields of bioinformatics and parallel programming.

By looking at the experimental results presented in Chapter 4, we can conclude that this goal has been accomplished, since ParRADMeth brings good results in terms of runtime, speedup and scalability. In fact, for every dataset the tool was able to reduce its execution time to less than five minutes, being up to 195 times faster than the execution of the original tool. Another remarkable fact is that, even for a sequential execution of the parallel tool, the optimization techniques make our tool faster than the original one. That is, not only our tool adapts the algorithm to HPC systems in an efficient and scalable way, but it also gets lower execution times when using the same resources as the original tool. Moreover, ParRADMeth is able to reach superlinear speedups even in worst cases scenarios for executions with up to 16 processing elements, and its performance was better the larger the dataset was.

During this project, I was able to learn many interesting topics from fields like bioinformatics and HPC, where initially my knowledge was low. Therefore, it was necessary to invest some time in the study of these topics. This bachelor's thesis also improved my technical skills in different technologies that I already knew, such as C++ or MPI and introduced me into some new ones, such as OpenMP. In general terms, I consider that this project made me grow as a professional and improved my soft and technical skills, as well as it introduced me into the research field.

6.2 Relation to the bachelor's title

The development of this project allowed me to use some of the knowledge obtained during the degree. First of all, I must mention the programming knowledge learned from "Programming I and II" that was improved through the rest of the courses. In addition, when using MPI and multithread technologies some concepts explained on "Concurrency and Parallelism" were applied, as well as some advanced MPI topics learned from "Computer Architecture". Also some concepts about clusters, which are the target architectures of the developed tool, were learned from "Administration of Infrastructures and Information Systems". Finally, the project was planned trying to follow a classic development cycle, learned from "Project Management", even though, since it is a research project, tasks dedication can not be estimated very precisely.

6.3 Future work

On the one hand, one possible future work line could be focusing on the MethPipe pipeline and continue implementing parallel algorithms for other different computational demanding tools that it is composed of, by means of MPI and OpenMP.

On the other hand, another future work line could be focusing on RADMeth and trying to expand its execution to different system architectures. Some examples could be a version for its execution in GPUs or a version for Big Data clusters using MapReduce parallel programming model.

Finally, by releasing the tool as open source, we let other developers contribute with any other improvement or modification result of specific requirements that were not in mind during its design.

Appendices

Appendix A

User Guide

The aim of this guide is to help final users of the application with the installation, configuration and execution of ParRADMeth.

A.1 Prerequisites

The project uses some libraries and programs for compilation, configuration and execution. Before starting the installation, please confirm that the following software is available in your system. Particular versions used during development are shown.

- **GCC** v8.3.0
- **GSL** v2.6
- **make** v3.82
- **MPI** compiler with suppoort for OpenMP:
  - **OpenMPI** v3.1.4
- **Git** (Optional)

Different versions of the software may work but they have not been tested.

A.2 Compilation

The project was designed so a system level installation is not needed, but user level compilation and execution can be done. Compilation of the tool for your system architecture can be done by following these steps:

1. **Download**. First, obtain project files by cloning this git repository.

2. **Compilation**. In this step go to the root folder of the project and use `make`.

3. **Installation**. In this steps, still on the root folder, use `make install`. This will place the executable on the `bin/` folder.

4. **Cleaning** (Optional). Optionally, still on the root folder of the proyect, use `make clean` to delete files generated during tool's compilation. Only unnecessary files are deleted, so executable files generated still work.

A.3 Execution

ParRADMeth must be executed with MPI execution commands (`mpiexec` or `mpirun`). Therefore, the tool can be executed using the following command from the root folder of the project:

```
mpiexec -n <numProcs> ./bin/ParRADMeth regression <options>
                       <design_matrix> <proportion_table>
```
(A.1)

being **numProcs** the number of MPI processes to execute, **design_matrix** the path to the file containing the design matrix, **proportion_table** the path to the file containing the proportion table and **options** a list of the following arguments:

- **-factor <f>** (Compulsory). The factor to test.
- **-o <output_file>** (Optional). Output option.
- **-h** (Optional). Print usage and exit.
- **-v** (Optional). Verbose, print more run information.

List of Acronyms

DM Differential Methylation. 1, 5, 10, 12–15

DMRs Differentially Methylated Regions. 3, 13

DNA Deoxyribonucleic Acid. 3

HMM Hidden Markov models. 13

HPC High Performance Computing. 1, 2, 19, 21, 43

MPI Message Passing Interface. 2, 17, 19, 21

SMT Simultaneous MultiThreading. 22

SPMD Single Program Multiple Data. 17, 19

WGBS Whole-Genome Bisulfite Sequencing. 1, 5, 6, 10, 12, 37

WGS Whole Genome Sequencing. 4

Bibliography

[1] F. Santos, B. Hendrich, W. Reik, and W. Dean, “Dynamic reprogramming of DNA methylation in the early mouse embryo,” *Dev Biol*, vol. 241, no. 1, pp. 172–182, 2002.

[2] M. Moarii, V. Boeva, J.-P. Vert, and F. Reyal, “Changes in correlation between promoter methylation and gene expression in cancer,” *BMC Genomics*, vol. 16, no. 1, pp. 1471–2164, 2015. [Online]. Available: https://doi.org/10.1186/s12864-015-1994-2

[3] S. Everhard, J. Tost, H. El Abdalaoui, E. Crinière, F. Busato, Y. Marie, I. G. Gut, M. Sanson, K. Mokhtari, F. Laigle-Donadey, K. Hoang-Xuan, J.-Y. Delattre, and J. Thillet, “Identification of regions correlating MGMT promoter methylation and gene expression in glioblastomas,” *Neuro-Oncology*, vol. 11, no. 4, pp. 348–356, 08 2009. [Online]. Available: https://doi.org/10.1215/15228517-2009-001

[4] M. Schultz, R. Schmitz, and J. Ecker, “‘Leveling’ the playing field for analyses of single-base resolution DNA methylomes,” *Trends Genet: TIG*, vol. 28, no. 12, pp. 583–585, 2012.

[5] M. Ehrlich, “DNA methylation in cancer: too much, but also too little,” *Oncogene*, vol. 21, no. 35, pp. 5400–5413, 2002.

[6] P. Lopez-Serra and M. Esteller, “DNA methylation-associated silencing of tumor-suppressor microRNAs in cancer,” *Oncogene*, vol. 31, pp. 1609–22, 2012. [Online]. Available: https://www.nature.com/articles/onc2011354

[7] Dolzhenko and Smith, “Using beta-binomial regression for high-precision differential methylation analysis in multifactor whole-genome bisulfite sequencing experiments,” *BMC Bioinformatics*, vol. 15, p. 215, 2014. [Online]. Available: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-15-215

[8] H.-U. Klein and K. Hebestreit, “An evaluation of methods to test predefined genomic regions for differential methylation in bisulfite sequencing data,” *Briefings*

in Bioinformatics, vol. 17, no. 5, pp. 796–807, 10 2015. [Online]. Available: https://doi.org/10.1093/bib/bbv095

[9] Q. Song, B. Decato, E. Hong, M. Zhou, F. Fang, J. Qu, T. Garvin, M. Kessler, J. Zhou, and A. Smith, "A reference methylome database and analysis pipeline to facilitate integrative and comparative epigenomics," *PLOS ONE*, vol. 8, no. 12, p. e81148, 2013. [Online]. Available: http://smithlabresearch.org/software/methpipe/

[10] M. Crowder, "Beta-binomial anova for proportions," *Appl Stat*, vol. 27, pp. 34–37, 1978.

[11] R. Lister, M. Pelizzola, R. Dowen, R. Hawkins, G. Hon, J. Tonti-Filippini, J. Nery, L. Lee, Z. Ye, Q. Ngo, L. Edsall, J. Antosiewicz-Bourget, R. Stewart, V. Ruotti, A. Millar, J. Thomson, B. Ren, and J. Ecker, "Human DNA methylomes at base resolution show widespread epigenomic differences," *Nature*, vol. 462, no. 7271, pp. 315–322, 2009.

[12] Y. Li, J. Zhu, G. Tian, N. Li, Q. Li, M. Ye, H. Zheng, J. Yu, H. Wu, J. Sun, H. Zhang, Q. Chen, R. Luo, M. Chen, Y. He, X. Jin, Q. Zhang, C. Yu, G. Zhou, J. Sun, Y. Huang, H. Zheng, H. Cao, X. Zhou, S. Guo, X. Hu, X. Li, K. Kristiansen, L. Bolund, and J. Xu, "The DNA methylome of human peripheral blood mononuclear cells," *PLoS Biol*, vol. 8, no. 11, p. 1000533, 2010.

[13] C. Becker, J. Hagmann, J. Müller, D. Koenig, O. Stegle, K. Borgwardt, and D. Weigel, "Spontaneous epigenetic variation in the Arabidopsis thaliana methylome," *Nature*, vol. 480, pp. 245–249, 2011.

[14] G. Challen, D. Sun, M. Jeong, M. Luo, J. Jelinek, J. Berg, C. Bock, A. Vasanthakumar, H. Gu, Y. Xi, S. Liang, Y. Lu, G. Darlington, A. Meissner, J. Issa, L. Godley, W. Li, and M. Goodell, "Dnmt3a is essential for hematopoietic stem cell differentiation," *Nat Genet*, vol. 44, no. 1, pp. 23–31, 2011.

[15] E. Hodges, A. Molaro, C. Dos Santos, P. Thekkat, Q. Song, P. Uren, J. Park, J. Butler, S. Rafii, W. McCombie, A. Smith, and G. Hannon, "Directional DNA methylation changes and complex intermediate states accompany lineage specificity in the adult hematopoietic compartment," *Mol Cell*, vol. 44, no. 1, pp. 17–28, 2011.

[16] Q. Song, B. Decato, E. Hong, M. Zhou, F. Fang, J. Qu, T. Garvin, M. Kessler, J. Zhou, and A. Smith, "A reference methylome database and analysis pipeline to facilitate integrative and comparative epigenomics," *PloS One*, vol. 8, no. 12, p. 81148, 2013.

[17] K. Hansen, B. Langmead, and R. Irizarry, "BSmooth: from whole genome bisulfite sequencing reads to differentially methylated regions," *Genome Biol*, vol. 13, no. 10, p. 83, 2012.

[18] K. Hebestreit, M. Dugas, and H.-U. Klein, "Detection of significantly differentially methylated regions in targeted bisulfite sequencing data," *Bioinformatics*, vol. 29, no. 13, pp. 1647–1653, 2013.

[19] H. Feng, K. Conneely, and H. Wu, "A bayesian hierarchical model to detect differentially methylated loci from single nucleotide resolution sequencing data," *Nucleic Acids Res*, vol. 42, no. 8, p. e69, 2014.

[20] D. Sun, Y. Xi, B. Rodriguez, H. Park, P. Tong, M. Meong, M. Goodell, and W. Li, "MOABS: model based analysis of bisulfite sequencing data," *Genome Biol*, vol. 15, no. 2, p. 38, 2014.

[21] G. Smyth, "Linear models and empirical bayes methods for assessing differential expression in microarray experiments," *Stat Appl Genet Mol Biol*, vol. 3, 2024, article3.

[22] C. Warden, H. Lee, and J. Tompkins, "COHCAP: an integrative genomic pipeline for single-nucleotide resolution DNA methylation analysis," *Nucleic Acids Res*, vol. 41, p. e117, 2013.

[23] A. Akalin, M. Kormaksson, S. Li, F. Garrett-Bakelman, M. Figueroa, A. Melnick, and C. Mason, "methylKit: a comprehensive r package for the analysis of genome-wide dna methylation profiles," *Genome Biol*, vol. 13, no. 10, p. 87, 2012.

[24] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org/

[25] Hildebrand, Dean, Nisar, Arifa, and R. Haskin, "pNFS, POSIX, and MPI-IO: a tale of three semantics," 01 2009.

[26] T. Rajeev, G. William, and L. Ewing, "Optimizing noncontiguous accesses in MPI–IO," *Parallel Computing*, vol. 28, no. 1, pp. 83–105, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819101001296

[27] A. Akalin, F. Garrett-Bakelman, M. Kormaksson, J. Busuttil, L. Zhang, I. Khrebtukova, T. Milne, Y. Huang, D. Biswas, J. Hess, C. Allis, R. Roeder, P. Valk, B. Löwenberg, R. Delwel, H. Fernandez, E. Paietta, M. Tallman, G. Schroth, C. Mason, A. Melnick, and M. Figueroa, "Base-pair resolution DNA methylation sequencing reveals profoundly divergent epigenetic landscapes in acute myeloid leukemia," *PLoS Genet.*, vol. 8, no. 6, 2012.

[28] H. Heyn, N. Li, H. Ferreira, S. Moran, D. Pisano, A. Gomez, J. Diez, J. Sanchez-Mut, F. Setien, F. Carmona, A. Puca, S. Sayols, M. Pujana, J. Serra-Musach, I. Iglesias-Platas, F. Formiga, A. Fernandez, M. Fraga, S. Heath, A. Valencia, I. Gut, J. Wang, and M. Esteller, "Distinct DNA methylomes of newborns and centenarians," *Proc Natl Acad Sci U S A*, vol. 109, no. 26, pp. 10 522–7, 2012.

[29] B. Berman, D. Weisenberger, J. Aman, T. Hinoue, Z. Ramjan, Y. Liu, H. Noushmehr, C. Lange, C. Dijk, R. Tollenaar, D. Van Den Berg, and P. Laird, "Regions of focal DNA hypermethylation and long-range hypomethylation in colorectal cancer coincide with nuclear lamina-associated domains," *Nature genetics*, vol. 44, pp. 40–6, 11 2011.

[30] K. Hansen, S. Sabunciyan, B. Langmead, N. Nagy, R. Curley, G. Klein, E. Klein, D. Salamon, and A. Feinberg, "Large-scale hypomethylated blocks associated with Epstein-Barr virus-induced B-cell immortalization," *Genome research*, vol. 24, 09 2013.