Università degli Studi di Torino

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

The road towards a cloud-based high-performance solution for genomic data analysis

Candidate:
Fabio Tordini

Supervisor:
Prof. Marco Aldinucci

Coordinator:
Prof. Mariangiola Dezani

April 1, 2016

Abstract

Nowadays, molecular biology laboratories are delivering more and more data about DNA organisation, at increasing resolution and in a large number of samples. So much that genomic research is now facing many of the scale-out issues that high-performance computing has been addressing for years: they require powerful infrastructures with fast computing and storage capabilities, with substantial challenges in terms of data processing, statistical analysis and data representation.

With this thesis we propose a high-performance pipeline for the analysis and interpretation of heterogeneous genomic information: beside *performance*, *usability* and *availability* are two essential requirements that novel Bioinformatics tools should satisfy. In this perspective, we propose and discuss our efforts towards a solid infrastructure for data processing and storage, where software that operates over data is exposed as a service, and is accessible by users through the Internet.

We begin by presenting NuChart-II, a tool for the analysis and interpretation of spatial genomic information. With NuChart-II we propose a graph-based representation of genomic data, which can provide insights on the disposition of genomic elements in the DNA. We also discuss our approach for the normalisation of biases that affect raw sequenced data.

We believe that many currently available tools for genomic data analysis are perceived as tricky and troublesome applications, that require highly specialised skills to obtain the desired outcomes. Concerning usability, we want to rise the level of abstraction perceived by the user, but maintain high performance and correctness while providing an exhaustive solution for data visualisation.

We also intend to foster the availability of novel tools: in this work we also discuss a cloud solution that delivers computation and storage as dynamically allocated virtual resources via the Internet, while needed software is provided as a service. In this way, the computational demand of genomic research can be satisfied more economically by using lab-scale and enterprise-oriented technologies. Here we discuss our idea of a task farm for the integration of heterogeneous data resulting from different sequencing experiments: we believe that the integration of multi-omic features on a nuclear map can be a valuable mean for studying the interactions among genetic elements. This can reveal insights on biological mechanisms, such as genes regulation, translocations and epigenetic patterns.

Contents

List of Figures

Glossary

This short glossary contains some definitions of biology-related terms used throughout this thesis. It is not intended to give a thorough explanation of the related concepts.

base refers to one of the *nucleobases* that build DNA and RNA, and that give the helical structure of DNA and RNA. The primary, or canonical, nucleobases are *cytosine*, *guanine*, *adenine*, *thymine* and *uracil*, abbreviated as C, G, A, T, and U.

basepairs (unit: **bp**) are the building blocks of the DNA double helix and contribute to the folded structure of both DNA and RNA. The size of an individual gene or an organism's entire genome is often measured in base pairs: the haploid human genome (23 chromosomes) is estimated to be about 3.2 billion bases long and to contain 20,000–25,000 distinct protein-coding genes. **kb** and **Mb** are often used to refer to thousands of basepairs (kilo base) and millions of basepairs (mega base).

binding site is a region on a protein, or piece of DNA or RNA, to which ligands (specific molecules) may form a chemical bond. Binding sites are often an important component of the functional characterization of biomolecules, as they are responsible for the chemical reaction that transforms a substrate into a product.

chromatin is a complex of macromolecules found in cells, consisting of DNA, protein and RNA, and plays an active role in maintaining DNA integrity and controlling DNA replication and gene expression. It is only found in eukaryotic cells (cells with defined nuclei).

chromosome is a packaged and organized structure containing most of the DNA of a living organism. It is complexed with many structural proteins called histones, with associated transcription factors and several other macromolecules.

contact maps record the interactions between different sections of the chromosome in the form of a weighted adjacency matrix A, where two fragments i and j interact with frequency $A_{i,j}$.

CTCF is a transcription factor that in humans is encoded by the CTCF gene. It is involved in many cellular processes, including transcriptional regulation, insulator activity and regulation of chromatin architecture.

DNA (deoxyribonucleic acid) is a molecule that carries most of the genetic instructions used in the development, functioning and reproduction of all known living organisms and many viruses. Alongside proteins and carbohydrates, nucleic acids compose the three major macromolecules essential for all known forms of life.

DNA polymerases are enzymes that create DNA molecules by assembling nucleotides, and usually work in pairs to create two identical DNA strands from a single original DNA molecule.

DNA replication is the process of producing two identical replicas from one original DNA molecule. This biological process occurs in all living organisms and is the basis for biological inheritance.

DNase refers to any enzyme capable of establishing or cleavage any chemical bonds that link nucleotides.

exon is any part of a gene that codes for a part of the final mature RNA product of that gene after introns have been removed by RNA splicing.

fasta is a text-based format for representing either nucleotide sequences or peptide sequences, in which nucleotides or amino acids are represented using single-letter codes. The format originates from the FASTA software package, but has now become a standard file format in bioinformatics.

fastq is a text-based format for storing both a biological sequence (usually nucleotide sequence) and its corresponding quality scores, which are each encoded with a single ASCII character for brevity. It has recently become the de facto standard for storing the output of high-throughput sequencing instruments.

GC-content is the percentage of nitrogenous bases on a DNA molecule, that are either *Guanine* or *Cytosine*. GC ratios within a genome is found to be markedly variable: such variations within the genomes of complex organisms result in a mosaic-like formation, with isolated regions called *isochores*. GC-rich isochores contain many proteins coding genes, thus supporting the discovery of gene-rich regions in the genome.

gene is a locus (or region) of DNA that encodes a functional RNA or protein product, and is the molecular unit of heredity. Most biological traits are under the influence of many different genes, as well as the gene–environment interactions. Some genetic traits are instantly visible, such as eye colour, while some are not, such as blood type or the thousands of basic biochemical processes that comprise life.

histones are highly alkaline proteins found in eukaryotic cells nuclei that package and order the DNA into structural units called nucleosomes. They are the chief protein components of chromatin and play a role in gene regulation mechanism.

immunoprecipitation is a technique used to isolate and concentrate a particular protein from a sample containing many thousands of different proteins, using an antibody that specifically binds to that particular protein.

intron is a nucleotide sequence within a gene that is removed by RNA splicing during maturation of the final RNA product. Sequences that are joined together in the final mature RNA after RNA splicing are exons.

isochores are large regions of DNA (greater than 300 kb) with a high degree uniformity in guanine (G) and cytosine (C): G-C and C-G (collectively GC content).

isoforms are mRNAs that are produced from the same locus but are different in their transcription start sites (TSSs), protein coding DNA sequences (CDSs) and/or untranslated regions (UTRs), potentially altering gene function.

locus (plural loci) is the specific location or position of a gene or DNA sequence, on a chromosome, in the field of genetics.

mappability score of a DNA sequence is the inverse of the number of times that a read, originating from any position in the reference genome, maps to the genome itself. It permits to identify those regions that produce reads which map back unambiguously (and uniquely) to themselves.

mapping generally refers to the alignment of each read — resulting from a DNA sequencing process — to the reference genome, in order to computationally identify where it originated. The primary complication of such process is that a read may map equally well (or *nearly* equally well) to multiple positions, because of repetitive sequences in the genome. Short read mapping algorithms try to identify the best mapping position for each read, that minimizes the number of differences between the read and the genome, and compute a mapping quality score that estimates the probability that the assigned location is the correct position.

methylation denotes the addition of a methyl group to a substrate or the substitution of an atom or group by a methyl group. In DNA sequencing, it modifies the function of the DNA, typically acting in various contexts, from repressing gene expression, X-chromosome inactivation and carcinogenesis.

omics is an English-language neologism that informally refers to a field of study in biology ending in *-omics*, such as genomics. The related suffix *-ome* is used to address the objects of study of such fields, such as the genome.

PCR (polymerase chain reaction) is a technology in molecular biology used to amplify a single copy or a few copies of a piece of DNA across several orders of magnitude, generating thousands to millions of copies of a particular DNA sequence. Quantitative PCR is an established tool for DNA quantification that measures the accumulation of DNA product after each round of PCR amplification.

PET (paired-end tags or di-Tags) are the short sequences of a DNA fragment which are unique enough that they (theoretically) exist together only once in a genome, therefore making the sequence of the DNA in between them available upon search or upon further sequencing.

primer refers to a strand of short nucleic acid sequences (generally about 10 base pairs) that serves as a starting point for DNA synthesis.

reads refer to short sequences of DNA, typically 25–400 bp long. Basically, reads are raw sequences that come off a sequencing machine.

RNA (ribonucleic acid) is a polymeric molecule implicated in various biological roles in coding, decoding, regulation, and expression of genes. Like DNA, it is one of the major macromolecules essential for all known forms of life.

RNA polymerases are enzymes that produce primary transcript RNA, and are necessary for constructing RNA chains using DNA genes as templates, during the transcription process.

mRNA (Messenger RNA) is a large family of RNA molecules that convey genetic information from DNA to the ribosome, where they specify the amino acid sequence of the protein products of gene expression.

ribosome is a large and complex molecular machine found within all living cells, that serves as the site of biological protein synthesis (translation). Ribosomes link amino acids together in the order specified by messenger RNA (mRNA) molecules.

RSSs (recombination signal sequences) are composed of three elements: a heptamer of seven conserved nucleotides, a spacer region of 12 or 23 basepairs in length, and a nonamer of nine conserved nucleotides. They play an active role in the mechanism of genetic recombination that occurs only in developing lymphocytes during the early stages of T and B cell maturation. The process is a defining feature of the adaptive immune system.

sam (bam) is a text format for storing sequence data in a series of tab delimited ASCII columns. Most often it is generated as a human readable version of its sister BAM format, which stores the same data in a compressed, indexed, binary form. SAM format data is output from aligners that read FASTQ files and assign the sequences to a position with respect to a known reference genome.

sonication is the act of applying sound energy to agitate particles in a sample, for various purposes. Ultrasonic frequencies (>20 kHz) are usually employed, leading to the process also being known as ultra-sonication.

SRA (sequence read archive) is a "raw data" archive, and also a common format (**.sra**) used by the NCBI, EBI, and others for storing reads and read alignments.

TADs (topologically associated domains) can be defined as dense, contiguous regions of chromatin (*bins*) that present a high frequency of intra-domain, rather than inter-domain, chromatin interactions. They are delimited by sharp boundaries that clearly separate more spatially compact regions with respect to their surrounding regions.

transcription is the first step of gene expression, in which a particular segment of DNA is copied into RNA (mRNA) by the enzyme RNA polymerase.

transcription factor is a protein that binds to specific DNA sequences, thereby controlling the rate of transcription of genetic information from DNA to mRNA. Transcription factors may attach to specific sequences of DNA adjacent to the genes that they regulate (DNA binding site), playing a crucial role in the regulation of gene expression.

transcriptome is the set of all RNA molecules, including mRNA, rRNA, tRNA, and other non-coding RNA transcribed in one cell or a population of cells.

translocation refers to a chromosome abnormality caused by rearrangement of parts between non-homologous chromosomes.

Chapter 1

Introduction

High-Performance Computing describes a set of hardware and software techniques developed for building computer systems capable of quickly execute large amounts of computation. These techniques have generally relied on harnessing the computing power of large numbers of processors working in parallel, either in tightly-coupled shared-memory multiprocessors or loosely-coupled clusters of PCs. Experience has shown that a great deal of software support is necessary to promote the development and tuning of applications on parallel architectures.

Bioinformatics can be broadly defined as the field of science in which biology, computer science and information technology merge together into a single discipline. More precisely, it comprises the set of computing techniques employed to manage and extract useful information from the DNA/RNA/Protein sequence data, being generated (at very high volumes) by automated techniques (e.g., DNA sequencing) and stored in large public databases (e.g., EBI[1], NCBI[2]). Certain methods for analysing genetic data have been found to be extremely computationally intensive, providing motivation for the use of powerful computers.

The work discussed in this thesis has its *raison d'être* in these two definitions: there is an increasing demand for powerful applications coming from scientific domains, such as Biology and Genomics. In particular, recent advances

[1] `http://www.ebi.ac.uk/`
[2] `http://www.ncbi.nlm.nih.gov/`

in Molecular Biology and Bioinformatics techniques brought to an explosion of the information about the spatial organisation of the DNA in the nucleus: DNA sequencing has been driving unprecedented discoveries in the life sciences since the emergence of next-generation sequencing (NGS) technologies. The sequenced genome is the product of an extensive computational process that transforms hundreds of Gigabytes of raw sequenced output into aligned genomes, and the sequencing industry has been demanding for increasingly larger compute and storage resources.

From the hardware perspective, multi-core processors have become the *de facto* standard in microprocessors production [14]. For several decades, processors' clock speed kept increasing of about 30% per year, until reaching a physical limit: when clock frequencies exceed 5GHz, the power a microprocessor could dissipate dramatically rises, causing the chips to melt. This has forced microprocessor vendors to increase performance by producing chip multiprocessors with multiple computing cores: the number of processing units per chip is doubling with each new generation processor [98], as transistors get smaller and smaller. Current processors are composed of 4 or more (up to 12) cores on the same chip, and this number is continuously increasing every year, up to the point that the term *many-cores* has been introduced to emphasize the large amount of cores per chip that some novel devices contain. One such example is the *Many Integrated Core* architecture by Intel, better known as Intel Xeon Phi. Also GPUs fall in this classification: apart from their 3D-graphics rendering capabilities, GPUs have became more and more "programmable" and are currently used for general-purpose computing.

These technologies are often combined together in high-end heterogeneous systems, equipped with several multi-core processors on the same board plus one or more many-core devices, and these computing nodes are tied together using interconnection networks that link processors and shared memory, resulting in larger and more complex HPC infrastructures.

In a scenario with an exponential expansion of biological information, inevitable difficulties arise for data storage and analysis: with the amount of data

growing continuously, it is becoming increasingly daunting for small laboratories or even large institutions to establish and maintain computational infrastructures for data processing. Cloud computing is nowadays a consolidated technology that exploits the full potential of *virtualised* resources to deliver computation and storage via the Internet [133]: virtualisation techniques enable a larger resource usage by sharing a given hardware among several users, thus reducing the required amount of instances of that particular device.

Albeit relatively new, cloud computing holds great promises in effectively addressing data storage and analysis problems that arise in many scientific disciplines. For instance, cloud computing is wide-spreading in Bioinformatics, just because it is a discipline heavily dependent on data and, even more, on space-consuming and time-consuming data processing tasks.

1.1 High-Performance Computing overview

Parallel programming has long been related to HPC environments, and since its early introduction it has not embraced much more than low-level communication and synchronisation libraries. Despite providing a complete control over the parallel application, in the hierarchy of abstractions it is only slightly above toggling absolute binary into the front panel of the machine. With the emergence of cluster and grid computing, and parallel architectures becoming dynamic and heterogeneous, the possibilities for ad-hoc optimizations are limited and this programming methodology begun to suffer for the lack of proper tools to easily implement parallel applications.

By definition, high-performance computing main purpose is "high performance", but FLOPS and speed-up should not be the only measures: human productivity, total cost and time-to-solution are equally — if not more — important [131]. While a big methodological change is required to build applications that are likely to be designed as parallel programs, existing applications should be easily ported to multi-cores with moderate effort. In this regard, *portability* turns out to be one of the mayor problems to be addressed by a parallel remains an outstanding problem that a parallel programming model should address:

the ability to compile and execute different architectures with none to minimum changes, while making the best possible use of the underlying (parallel) architecture.

All in all, programs that worsen their performance when ported on different architectures (i.e., bad scaling, limited speed-up, etc.) must be rewritten, thus nullifying most of the benefits of code portability. Follows a problem commonly referred to as *performance portability*, or the ability to write a single software implementation that can be targeted to either a CPU or GPU (or a combination of them) with high performance. This problem remains partially unaddressed, because it requires an agreed-on functional language specification, while many of the currently available programming languages adhere to a programming model that is innately unsuitable for certain architectures.

A high-level approach to parallel programming is perceived by many as the best solution that can foster performance portability: general-purpose patterns that allow a featured compiler to produce efficient code for most of the available computer architectures.

1.1.1 HPC architectures

The growing need of improved performance and the continuing constraints on power and scalability in multi-core processors, led system designers to look at heterogeneous solutions that incorporate specialized processing facilities, designed to handle particular tasks. As an example, a heterogeneous design brings together CPUs and GPUs processing: GPUs, apart from their 3D-graphics rendering capabilities, have become more and more "programmable", up to the point that they are currently used for general-purpose computing and are now exploited for intensive computation over large sets of data. Large HPC systems are equipped with these relatively inexpensive and powerful devices as a way of accelerating parts of the applications they are running.

Distributed systems vary widely but share a common characteristic: they require a communication network to connect inter-processor memory. In a distributed system, each computing node is usually a shared-memory computer with its own private memory, which needs to explicitly communicate with other

connected processing elements through a messaging infrastructure. A well recognised example of distributed system is called *cluster*, which is made up of a collection of commodity platforms connected by a commodity interconnection network — such as Ethernet or Infiniband.

Advances in biochemistry and biotechnology are placing unprecedented demands on data storage and analysis: as the amount of available data grows continuously, it is becoming increasingly daunting for small laboratories — or even large institutions — to establish and maintain computational infrastructures for data processing. Luckily enough, nowadays the *Cloud* paradigm has become a consolidated technology, that exploits the full potential of *virtualised* resources to deliver computation and storage via the Internet, enabling a larger resource usage by sharing a given hardware among several users. Buyya et al. gave a more formal and widely accepted definition of it [32], and they emphasize that a Cloud is "a type of parallel and distributed system consisting of a collection of inter-connected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resource(s)".

The picture of a cloud recalls the idea of a pool of resources with some kind of "undefined borders", that are made available on-demand to the final user upon payment. Albeit relatively new, cloud computing holds great promises in effectively addressing data storage and analysis problems that arise in many scientific disciplines. For instance, cloud computing is wide-spreading in Bioinformatics, just because it is a discipline heavily dependent on data and, even more, on space-consuming and time-consuming data processing tasks [133].

1.1.2 Structured parallel programming

Many well known sequential languages, such as C or Fortran, use specific statements to define and coordinate multiple execution flows. Libraries like MPI [122],

Pthreads [31] and OpenMP [124] provide APIs that help programmers to develop parallel applications by exposing constructs that control the programming model (message-passing, shared-memory, etc.) that the underlying computer systems express. However, these methods are considered "low-level" approaches, because a programmer needs to orchestrate every aspect of the concurrent execution of programs, including threads (or processes) creation, communication and synchronisation, which also requires a thorough understanding of the underlying hardware architecture.

Efficient parallel applications normally use a small number of common patterns that model parallelism exploitation forms. Structured parallel programming springs out from this evidence and has been developing into two main research branches: the *algorithmic skeletons* community and the *parallel design patterns* community.The latter has a software engineering perspective, while the former pursues an abstraction of the generic patterns of computation and interaction used in parallel applications. These two communities often reach to similar results that go for the benefits of both areas.

Structured parallel programming started with the concept of algorithmic skeletons defined by Cole [38], and has been successfully applied basically to any possible parallel environment, starting from shared-memory machines [7, 100], to grid and clusters [3, 5], and recently to cloud computing [34, 33]. A pattern (or *skeleton*) focuses on the parallel behaviour of the application — rather than on its implementation — which can be expressed in terms of a graph of computational nodes where data dependencies among nodes represent the parallel activity of a given application.

Parallel programming patterns permit to automatically create different parallel implementations starting from the high-level description that they encompass, hiding implementation details The parametric nature of the produced code is able to cope with different parallelism degrees. These points are the basic building blocks that ensure performance portability on various architectures.

1.2 HPC and Bioinformatics

It is nowadays acknowledged that the human genome is a complex machine built on a four-letters (AGCT) language: everything happening within cells derives from this language. Genomic material is folded inside each micrometer-scale cell nucleus, and genome organization is guided by genes interaction in and between chromosomes. These interactions reflect the transcription regulation process and guide genes activity: inferring genome organization is crucial for many cellular processes related to gene expression regulation, including DNA accessibility, epigenetic patterns and chromosome translocation.

More and more data about genome organisation springs out from laboratories, and this data must be processed and analysed in order to extrapolate useful information and obtain a significant interpretation of genomic data. However, no data analysis pipeline anywhere in any scientific field works as one monolithic process: different stages of the data analysis process are just fundamentally different, and have different parallelism, memory access and data access requirements. Also, it often makes sense to run the same stage of an analysis in a number of different ways, to demonstrate the robustness of novel results (which are not unusual in fields like Genomics and Bioinformatics), or to tackle different sorts of data, for example one in which a reference genome is available, compared to one where it is not. Here High-Performance Computing comes into play: from modelling scientific processes to the use of computers to obtain quantitative results from these models, it turns a domain science into a computational activity.

Some examples are worth to mention, where HPC actually led to concrete benefits. During the early assembly of the human genome, a Bioinformatics post-analysis of whole sets of shotgun sequencing runs supported by a large and powerful HPC system permitted to get very close to an unpredictable victory [156].

Crossbow [96] is software pipeline for whole genome resequencing analysis that exploits the cloud — and the Hadoop implementation of the MapReduce

pattern — to launch many copies of the short-read aligner Bowtie [97] in parallel. After Bowtie has aligned the reads (in the order of billions for a human resequencing project) to the reference genome, Hadoop[3] automatically sorts and aggregates the alignments by chromosomal region, and launches many parallel instances of the SOAPsnp[4] resequence utility.

Some other efforts have been dedicated to foster usability and availability of Bioinformatics tools and pipelines: Grendel [80] is a platform and language independent Web Service-based system for distributed resource management, and acts as a single access point to HPC infrastructures. Unipro UGENE [120] was designed as a multi-platform open-source software that integrates widely used Bioinformatics tools within a common user interface.

More recently, HPC solutions have been massively employed in the study of the genome architecture: Lieberman-Aiden's team used GPUs very aggressively while working on the 3-D maps of entire folded genomes [128], and found a structural basis for gene regulation — which they called a "genomic origami" — that allows the same genome to produce different types of cells: as genes specialize in functions, they are actually folding enabling new configurations. Their discovery revealed thousands of hidden switches that enable a cell to be in one state versus another. The big-data tools created for the study included parallelised pipelines for HPC clusters, dynamic programming algorithms and custom data structures.

1.2.1 Next-generation sequencing

DNA sequencing permits to determine the exact order of nucleotides present in a given DNA or RNA molecule. In the past decade, progresses in computing technology and chemical research have dramatically increased the ability to sequence DNA and gather information, bearing numerous biological and medical discoveries. The first major effort into DNA sequencing was the "Human Genome Project", completed in 2003 using "first-generation" sequencing methods, known as Sanger sequencing, developed in 1975 by Edward Sanger.

[3] http://hadoop.apache.org/
[4] http://soap.genomics.org.cn/soapsnp.html

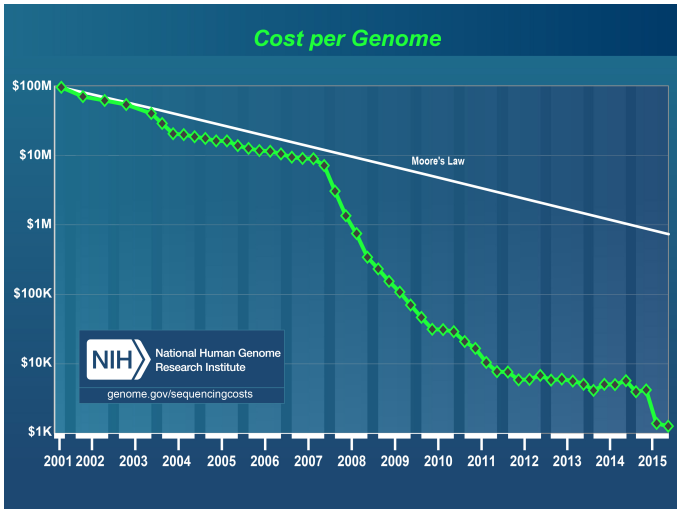


Figure 1.1: *Trends in DNA sequencing costs for a whole human-sized genome, compared against Moore's law. Picture publicly available, taken from* `http://www.genome.gov/sequencingcosts/`. *Updated at January 2016*

The high demand for low-cost genomic data has driven the development of high-throughput technologies, normally referred to as next-generation sequencing (NGS) techniques. These methods parallelise the sequencing process producing thousands (or millions) of biological data. DNA sequencing technology is outpacing Moore's law — in terms of cost decrease over years (see Figure 1.1) — and the cost for a whole genome is likely to soon reach the unprecedented lower-bound of $1000 USD[5]. Consequently, each new generation of sequencers will require more, not simply faster, computing and storage resources.

Although sequencing mainly refers to DNA, NGS techniques are used in many related genomic researches, such as transcriptome profiling, DNA-Protein interactions, epigenome characterization, and for the analysis of the chromosome conformation inside the nucleus of the cell.

1.2.2 Capturing chromosome conformation

The study of chromosome organization in the nucleus is extremely relevant to gain insight on biological function at the gene level, as well as the global nuclear level, and it will further enable the investigation of pathologies related to

[5] `http://www.genome.gov/sequencingcosts/`

genome instability or nuclear morphology [24]. Since 1980s, the predominant method for determining nuclear organization and chromatin conformation was FISH, a procedure that locates the positions of specific DNA sequences on chromosomes by using fluorescent probes which permit to identify (and quantify) the counterpart of the sequence in a biological sample.

Many techniques are now available to evaluate the physical proximity in cell populations *in vivo*: Chromosome Conformation Capture (**3C**) and its derivatives measure the frequency at which two chromosome fragments physically associate in a three-dimensional (3D) space, both within and between chromosomes, based on the propensity for those two locations to become cross-linked together [46].

Novel approaches, such as *4C*, *5C*, and *Hi-C*, combine 3C methodology with high-throughput sequencing to interrogate the 3C ligation product library more comprehensively [49, 102], and focus on the analysis of a set of predetermined loci: according to their coverage (i.e., the genome locations they are able to examine), 3C-based analyses provide information over increasing domains, reaching a genome-wide scope with Hi-C technique.

Quantifying the number of ligation events between two chromatin fragments gives their "contact frequency" in the examined population of nuclei, providing a pairwise estimator for their distance. Hi-C methods detect contacts between groups of loci, per chromosome or genome-wide, which are typically summarised in a non-negative matrix called "contact map".

The increasing volume of Hi-C data calls for the development of algorithms and data structures that can summarize, compress and integrate data sets while considering issues of time and space complexity. Such a computational approach can complement physics simulations and provide the necessary framework for large scale studies of the 3D genome architectures and its function.

Data representation is the foundation of our collective scientific knowledge, as readers' understanding of a dataset is generally limited to what is presented in a publication. Noteworthy, scientists customarily process the findings they have collected and the conclusions they draw into various kinds of visuals, in

order to represent the data. How to represent knowledge depends largely on the way reasoning can be done with that knowledge.

Moving to *omic* sciences, they are data-rich disciplines where knowledge is stored in a large number of different data sources, which need to be mined and integrated in such way that high-dimensional data is represented in a lower dimensional space, preserving the proximity relationship. For example, the output of a Hi-C process is a list of pairs of locations along all chromosomes, which can be represented as a square matrix where each element (i, j) of the matrix indicates the sum of read pairs matching in positions i and j. This matrix-based representation, called *contact matrix*, is reliable while looking at the interactions between two chromosomes, but becomes unsuitable to describe long-range chromatin interactions or to model a contact based metric of gene distances.

Another representation that is gaining a lot of success relies instead on graphs, which are used to provide a better description of the social behaviour of genes: a network (or graph) has a high level of expressiveness, insofar as nodes represent the actors of a process while edges identify relationships among the actors. Structural properties of a network can reveal significant information on how the actors of the represented process interact, while parallel algorithms can be employed to operate over a network. Moreover, graph-based representations of genes inside the nucleus can be very useful to map and re-interpret omics data, in order to characterize different spatially-associated domains [162] while providing an interpretation of genomic information by mean of complex, dynamical structures that organize items in an integrated way.

Indeed, sequencing results from expression profiles, methylation patterns and chromatin domains are analysed only from the 1D genome sequence viewpoint. Nonetheless, an increasing number of experiments highlights the importance of co-localization and co-expression of genes: there is an undeniable need for software solutions that permit the integration and the interpretation of multi-omics features on a nuclear map, capable of representing the effective disposition of genes in the three-dimensional space.

1.3 Contributions of this thesis

As genome sequencing technology is outpacing Moore's law, molecular biology laboratories are delivering more and more data about DNA organisation, at high resolution and in a large number of samples. So much that genomic research is now facing many of the scale-out issues that high-performance computing has been addressing for years: they require powerful infrastructures with fast computing and storage capabilities, with substantial challenges in terms of data processing, statistical analysis and data representation.

From these premises arises our main research question: *how can Computer Science and the HPC community support Bioinformatics on its increasing and forthcoming computing requirements?*

With this broad commitment in mind, we shrank our investigations to a concrete use case, and we outlined a number of criteria that we believe a Bioinformatics tool should meet: *performance*, *usability* and *availability*. Given the interdisciplinary nature of the problem, we could better formulate specific research questions that encouraged our investigations:

- how can we address performance needs in Bioinformatics applications, whose execution is strongly affected by memory requirements due to the considerable size of the data they work on.
- how can we address usability needs in Bioinformatics applications, considering that they are going to be used by a wide and heterogeneous range of scientists, with extremely varied backgrounds?
- how can we make our solutions quickly available to the scientific community, considering the increasing need for storage and computational power required to perform genomic data analysis?

In this Thesis we propose a high-performance solution for the analysis and interpretation of heterogeneous genomic information: moving from a novel tool for Hi-C data analysis, we will deploy a cloud solution that provides a solid hardware infrastructure for data processing and storage, while software tools

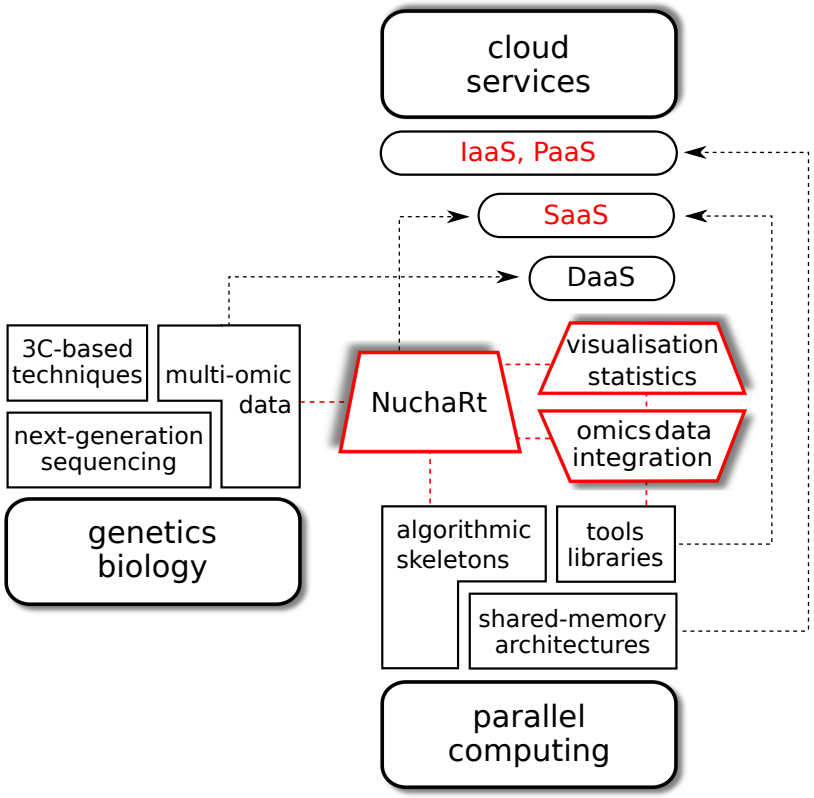


Figure 1.2: *A high-level view of our approach for heterogeneous genomic information processing and interpretation*

employed for different Bioinformatics pipelines are exposed as services and accessible by users through the Internet. Figure 7.1 depicts the overall schema of our workflow: Genetics and Biology laboratories churn out more and more data every day, which is available by means of web repositories that can easily be accessed by everyone. For this whole lot of data to be useful, it must be processed and analysed in order to extrapolate significant information. Here Computer Science comes in handy, providing hardware, software tools and methods for making these analyses fast and accurate. The Cloud, with its *on-demand* paradigm, has the potentials for being an effective solution to host computing power, storage resources and software services and make them available to the scientific community.

The background Chapters give a broad account on both HPC technologies and those genomic and biological aspects of interest, from which this work has

emerged. Inevitable is the emphasis on shared-memory multi-core architectures and structured parallel programming, because they are the natural ground upon which this thesis has grown. These Chapters also contain a literature review, that serves as a state of the art of the research field where this work applies.

In Chapter 4 we focus on the re-engineering of NuChart, a prototype written in the R language, originally designed for the analysis of genomic data resulting from Hi-C experiments. The presented tool had limits and drawbacks, mostly concerning poor performance, no parallelism exploitation, poor memory management and reduced analysis coverage. [148]. We re-worked the whole application and produced NuChart-II, written in C++ and built using high-level parallel programming patterns and *non-blocking* algorithms, so that it can fully exploit the computing capabilities exposed by modern shared-memory multi-core architectures, with attention to the memory optimisation required to handle a memory bound algorithm [149, 50]. We also discuss our solution for the normalisation of biases [108] that affect raw sequenced data (Section 4.3). Here we leverage the approach in the original NuChart prototype and provide a mean for assessing the probability of spatial proximity between genomic elements. Nevertheless, performance is of utmost importance in our work, and we discuss our achievements in Section 4.5.

In Chapter 5 we address the usability requirement: our idea is to rise the "level of abstraction" perceived by the user, but maintain high performance and correctness. Here we discuss the *re*-integration of NuChart-II into the R environment, that led to the Nucha*R*t package [150] R facilitates the integration of exploratory data analysis because it permits to easily move through the steps of model development, from data analysis to implementation and visualisation. Moreover, since the tool is mostly targeted to biologists — which are not expected to be expert C++ programmers — the R environment will make users more comfortable. Here we also discuss our efforts and ideas for an exhaustive data visualisation.

In Chapter 6 we present our solution aimed at fostering the availability of Bioinformatics applications, proposing Nucha*R*t as the key for the integration of multi-omic data: we describe a Cloud infrastructure that can be used to

seamlessly run Bioinformatics pipelines on virtual resources, so that the computational demand of genomic research can be satisfied more economically, by using lab-scale and enterprise-oriented technologies. In Section 6.3 we discuss our idea of a *task farm* for the analysis and integration of heterogeneous data: by mapping data resulting from RNA-Seq and ChIP-Seq experiments into Hi-C graphs, we can study the interactions among genetic elements that can reveal insights on biological mechanisms, such as genes regulation, translocations and epigenetic patterns. We tested our solution by comparing it against the work of Shen et al. [142], which presents associated RNA-Seq, ChIP-Seq and Hi-C experiments on the same samples of laboratory mouse (Section 6.5). Here we also discuss performance and benefits resulting from our cloud infrastructure.

Organisation of the Thesis

The work described in this thesis has the following organisation:

Chapter 2 provides a background on parallel computing, describing relevant architectural features of shared-memory multiprocessor systems. After introducing hardware concepts of parallel systems, some related topics are discussed, such as synchronisation techniques and memory management. Finally, a brief overview of current tools for parallel programming is given, with emphasis on structured parallel programming.

Chapter 3 presents an overview of methods used in molecular biology and computational biology to explore and analyse the DNA. High-throughput sequencing technologies for DNA exploration are introduced and discussed, together with 3C-based methods for capturing the organization of chromosomes with emphasis on the Hi-C technique and analytical approaches to detect and correct biases resulting from such experiments.

Chapter 4 describes our approach for developing a fast and scalable tool aimed at facilitating the analysis of chromosome conformation using a graph-based representation of genomic data. Here we propose NuChart-II, a software that allows to visualize and integrate genomic features involved

in the chromosome spatial organization, built on top of the FastFlow structured parallel programming framework.

Chapter 5 discusses the integration of our C++ application into the R environment, an important step toward our objective of augmenting the usability of Bioinformatics tools by rising the level of abstraction while maintaining performance and scalability. Here we also compare our approach to other available parallel facilities in R, motivating our choices, and briefly examine memory management in R. A discussion about issues in graph visualisation and proposed solutions is presented.

Chapter 6 gathers all previous efforts in building a fast and usable tool for genomic data analysis and presents a cloud infrastructure where tools for genomic data analysis are provided as services. Here we describe a task farm designed to integrate heterogeneous genomic data into a chromosomal map, by mean of NuChart-II. The whole infrastructure is characterised by a coarse grain parallelism among services and a finer grain that exploits the computing power of shared-memory multi-core architectures.

Chapter 7 concludes the thesis and discusses possible steps forward into the use of high-performance computing solutions for Bioinformatics.

List of relevant publications by the author

- F. Tordini, M. Drocco, C. Misale, P. Liò, I. Merelli, M. Aldinucci: "NuChart-II: the road to a fast and scalable tool for Hi-C data analysis," in *International Journal of High Performance Computing Application*, 2015. (*submitted*)
- F. Tordini, I. Merelli, P. Liò, M. Aldinucci and L. Milanesi: "Nucha*R*t: embedding High-Performance Computing in R for augmented DNA Exploration," in *Post-Conference proc. of the 12th Intl. meeting on Computational Intelligence methods for Bionformatics and Biostatistics (CIBB)*, LNBI, Springer, 2016. (*To appear*)
- F. Tordini, M. Drocco, C. Misale, L. Milanesi, P. Liò, I. Merelli, and M. Aldinucci: "Parallel Exploration of the Nuclear Chromosome Conformation with NuChart-II," in *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*, Turku, Finland, 2015
- M. Drocco, C. Misale, G. P. Pezzi, F. Tordini, and M. Aldinucci: "Memory-Optimised Parallel Processing of Hi-C Data," in *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*, Turku, Finland, 2015
- I. Merelli, F. Tordini, M. Drocco, M. Aldinucci, P. Liò, L. Milanesi: "Integrating Multi-omic features exploiting Chromosome Conformation Capture data," in *FRONTIERS IN GENETICS*, 6(40), February, 2015
- F. Tordini, M. Drocco, I. Merelli, P. Liò, L. Milanesi, and M. Aldinucci: "NuChart-II: a graph-based approach for the analysis and interpretation of Hi-C Data," in *Post-Conference proc. of the 11th Intl. meeting on Computational Intelligence methods for Bionformatics and Biostatistics (CIBB)*, LNBI, Springer, Cambridge, UK, 2014.
- F. Tordini: "Memory Management on Heterogeneous Architecture," in *Proceedings of the Work in Progress Session* held in connection with *Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, 2014

- M. Aldinucci, F. Tordini, M. Drocco, M. Torquati and M. Coppo: "Parallel stochastic simulators in systems biology: the evolution of the species", in *Proc. of Intl. Euromicro PDP 2013: Parallel Distributed and network-based Processing*, 2013
- F. Tordini, M. Aldinucci and M. Torquati: "High-level lock-less programming for multicore," in *Proceedings of the Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES) — Poster Abstracts*, Fiuggi, Italy, 2012.

Chapter 2

Background on parallel computing

In recent years, parallel and distributed architectures have become more and more available, thanks to a substantial improvement in computer and networking technology. This revolution not only affected large-scale, high-end systems, but also personal computing facilities and small clusters benefit from these hardware advances. Moreover, the fast and wide spread of multi-core processors among microprocessors designers strongly contributed to this change, that inevitably impacted also the programming model scenario: parallel computing needs to be constantly supported by efficient, reliable though handy parallel/distributed programming paradigms.

In this chapter, we provide a background on parallel computing, describing relevant architectural features of shared-memory multiprocessor systems. Then walking through some related topics, such as synchronisation mechanisms and memory management, a brief overview of the current tools for parallel programming is given, with emphasis on structured parallel programming.

Parallel computing systems

Parallel systems are categorized according to the number of instruction streams and data streams they can simultaneously manage: SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instructions Multiple Data). SIMD architectures are specifically oriented to optimal data-parallelism support and mostly

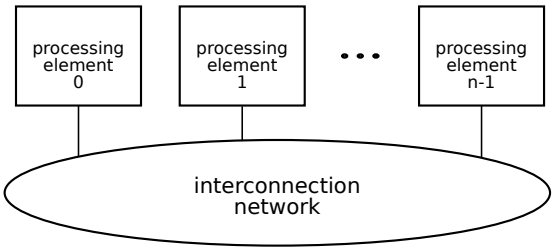


Figure 2.1: *MIMD system: processing elements (commodity computers or multiprocessors) are connected through an interconnection network, that serves as a physical mean for communication.*

target fine-grained arithmetic operations. Common SIMD architectural implementations are array processors, vectorization facilities for CPUs and Graph Processing Units (GPUs). MIMD architectures are the most widely adopted high-performance machines, and range from low-to-medium parallelism (i.e., shared-memory servers, PC/workstation clusters) to high-end massively parallel platforms. In these systems, parallel programming methodologies may be applied to the process (or thread) level, in order to exploit parallelism.

SIMD architectures have always been used in computer architectures as special purpose interfaces (such as network devices or audio/video interfaces). The last decade saw the strong emergence of accelerators explicitly developed to increase the performance of some classes of applications, namely data parallel programs where data are split among processors and the same sequence of instructions is performed over each data item. Only in the last decade Graphical Processing Units (GPUs) have attracted more attention for being used in general purpose computing (GPGPU) and have become more and more "programmable". Standard APIs for GPU programming have quickly emerged — NVIDIA CUDA, OpenCL and OpenACC among the others — thus qualifying GPUs as a specific class of many-core architectures.

Porting applications from CPU to GPUs does not necessarily bring remarkable performance results, and usually requires substantial rewriting of the application's kernel code. Plus, GPUs do not allow general interaction patterns

among control flows (threads), so only specific parallel patterns can be instantiated. We will not go into greater detail about GPUs, as it lies outside of the scopes of this thesis.

MIMD systems have a number of independent processing elements which can be general-purpose commercial computers or CPUs with a local memory and some I/O resources (see Figure 2.1). In this simplified view, two main classes of MIMD systems can be distinguished: *shared-memory* systems (or multiprocessors) and *distributed-memory* systems (or multicomputers). Multiprocessor systems are normally composed of two or more processing elements (PEs) in a single machine, where each PE features two or more (up to 12) processing units (PUs) on the same chip, with each PU implementing a 2-way or 4-way simultaneous multi-threading, allowing the concurrent execution of up to 4 threads. Each PU has access to its own memory hierarchy, from private registers and caches to a shared physical memory space. Multicomputers are composed of many complete computers that can cooperate via I/O.

An *interconnection* network connects all processing elements and permits inter-process communication: in a shared-memory system the interconnection network implements shared-memory accesses and memory hierarchy management, while it provides message passing among processing elements in a multicomputer system. Current most common architectures in high-performance computing are multicomputer systems whose processing elements are "multi-core" machines, arguably couples with a number of SIMD accelerators. [6] (see Figure 2.2).

[6] In the rest of this work, we will use the term *core* to refer to a *processing unit* within a processor's circuit die. In this regard, a multi-core implements multiprocessing in a single physical processor package.

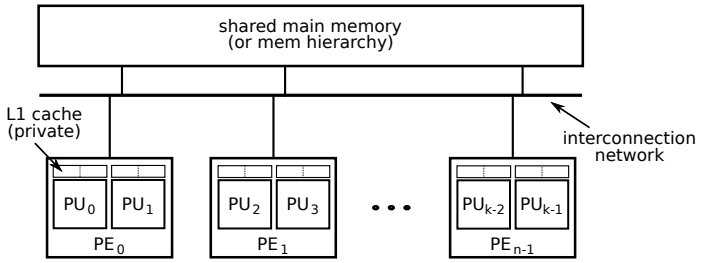


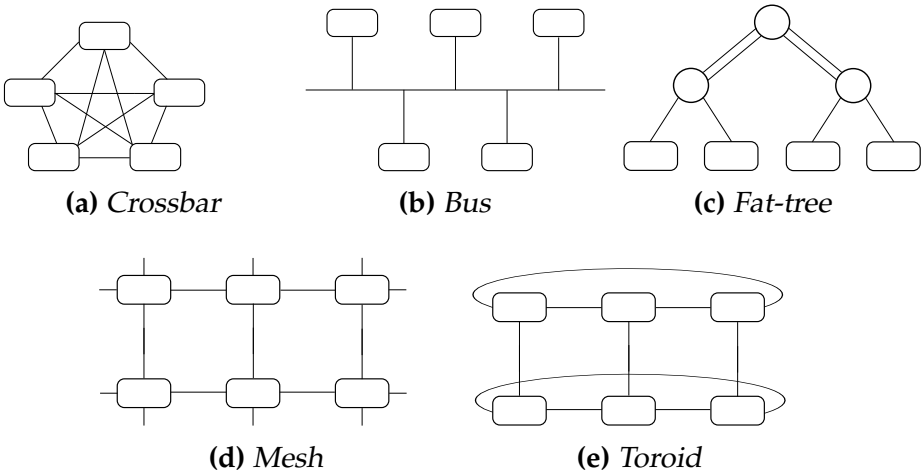
Figure 2.2: *Shared-memory system: processors address every part of the main memory and might physically share parts of the memory hierarchy (e.g, L2 caches). The interconnection network enables inter-process communication*

2.1 Shared-memory architectures

In a shared-memory multiprocessor system, any processor is able to address every part of the main memory, with the possibility of physically share information and cooperate with other processors. Some important characteristics, like interconnection networks, cache management, synchronization mechanisms, memory organization and memory consistency mechanisms, have to be examined when discussing shared-memory systems.

Interconnection networks

Processors and shared physical memory are linked through an interconnection network that enables data exchange among processing elements and consequently influences latency and speed of inter-process communication. Two well-known interconnection network topologies are *crossbar* and *bus*. A crossbar (Figure 2.3a) represents an *all-to-all* connection and has the advantage of keeping the latency fixed, but becomes impractical with a large number of processing elements, because the number of links is equal to n^2, with n being the number of nodes. The bus (Figure 2.3b) is generally pictured as a single link to which each node is connected. Buses are not suitable for highly parallel systems, due to their limited scalability and significant latency, proportional to the number of linked nodes. These two solutions can be seen as two extremes for interconnection network topologies: highly parallel architectures normally find a compromise on the number of connections each processing element has, so that



(a) *Crossbar* **(b)** *Bus* **(c)** *Fat-tree*

(d) *Mesh* **(e)** *Toroid*

Figure 2.3: *Common interconnection networks*

each processing element is directly connected to only a small subset of neighbour nodes. A *fat-tree* (Figure 2.3c) is widely used in large-scale systems, because it minimizes conflicts in the network but provides very high bandwidth: a key feature in fat-tree interconnections is that the link bandwidth increases from leafs to root, while each layer has the same aggregate bandwidth. *Mesh* and *Torus* (Figure 2.3d, 2.3e) are topologies with nodes arranged in a rectilinear array of $n = 2, 3, \ldots$ dimensions, with processors connected to their nearest neighbours and to corresponding processors on opposite edges of the array.

While bus and crossbar are not suitable for connecting large numbers of processing elements, the other topologies become interesting as the number of processing elements increases, because higher bandwidth and fault tolerance are better preserved.

Caches and locality of reference

Caches play a role of utmost importance in shared-memory systems: they are small, high-speed buffer memories that temporarily hold those portions of the contents of main memory which are (believed to be) currently (or shortly) in use, and permit the execution rate of the machine to be substantially increased.

A multiprocessor structure is normally composed of multiple processing units (or cores) on a single chip. Every multi-core processor includes a local

cache hierarchy, and according to vendors' choices some cache levels might be shared among processing elements: a quite common configuration sees separated private L1 caches for data and instruction, while L2 is unified and could be shared among cores or kept private for every processing unit. Some architectures provide also a shared L3 cache. Cache sizes at each level are different for each processor, and mostly depend on manufacturer design.

Cache memories are pervasive in today's computer systems, so that it is difficult to imagine processors without them. Cache memories, together with virtual memory and processor registers, form the so called *memory hierarchy*, that embroils the principle of locality of reference. Most applications exhibit *temporal* and *spatial* localities among instructions and data. Spatial locality refers to those situations in which memory locations that are "near" to the currently referenced address (i.e., near in terms of address), will likely be referenced in the near future. *Temporal* locality supposes that the currently referenced address will likely be referenced again in the near future. Most likely referenced items are supposed to be kept in the fastest devices – the smaller the memory size, the shorter the access time.

Cache coherence

Memory hierarchy implies that shared resource data is stored in multiple copies in local devices. This improves performance but has a counterpart: consistency of shared resources must be maintained and caches must be kept updated. This is usually accomplished implementing some automatic *coherence* mechanism, which guarantees that when a memory location is read it returns the latest value written in that location. *Snooping* and *directory-based* cache coherence are the most widely used protocols [129, 99]. Snoopy protocols exploit the interconnection network to maintain coherence of data: each cache line is associated with a state, and every memory access operation is notified to each processor in the system by broadcasting anyone any change of states associated with cache lines. This approach, despite being simple and efficient with a bus interconnection, has a limit in the number of processing elements that can be connected to the bus, making this solution better suit for small-scale multiprocessor systems.

Large-scale multiprocessor and distributed systems use more complex interconnection mechanisms, and normally adopt a directory-based cache coherence protocol: a directory with main memory units maintains the coherence between caches. To each cache block corresponds a directory entry, and the entry keeps track of shared copies or the identification of the processor that contains modified data. When a processor wants to load an entry from the primary memory to its cache, it sort of asks for permission to the directory: if the entry has not been modified, a copy is sent to the requesting processor's cache, and the directory entry is updated to reflect the existence of a shared copy. When an entry is changed, the directory either updates or invalidates other caches with a copy of that entry. This approach minimizes traffic in the interconnection network, but has an increased memory access latency and its performance is affected by the number of shared blocks and the number of copies of individual shared blocks that must be kept in memory.

Synchronisation

In a shared-memory architecture, multiple processing elements often operate on modifiable shared data. Such operations must guarantee that shared data is kept in a consistent state, which is achievable only if no concurrent manipulation is allowed and a supposed *read-modify-write* sequence of operations on a shared variable is an atomic action. This synchronisation problem is called *mutual exclusion*, and is a basic requirement to prevent race conditions in concurrent environments. It is recognised at the application level, as well as at run-time support and at firmware architecture level.

Atomic operations on a shared variable are often enforced using well known primitive operations named `lock` and `unlock`, whose semantic reflects that of a semaphore and imposes a wait condition on a shared variable (the semaphore) when it assumes a supposed *red* value, while it allows to proceed the execution when this value changes to *green*. At the run-time level, the use of atomic instructions (such as `test-and-set` or `compare-and-swap`) on the shared variable ensures the mutual exclusion.

Lock-based synchronisation mechanisms have a considerable impact on parallel applications performance, inasmuch they have a limiting effect on scalability and bring a consistent overhead in terms of memory accesses and memory congestion. Some techniques have been studied to address this problem [90]. For example, *lock-free* data structures permit to solve the mutual exclusion problem without locking mechanisms, while transactional memory has been proposed as an architectural mechanism that exploit cache coherence support to execute memory operations in an atomic way and obtain efficient lock-free synchronisation [72]. These techniques will be examined in Section 2.3

2.1.1 Memory organisation

A traditional classification of memory organisation takes into account the relative distance between processing elements and shared main memory, thus considering the shared-memory access time. When the shared-memory access time is equal for every processing element in the system, we have a *Uniform Memory Access* (**UMA**) system. Often referred to as SMP (Symmetric MultiProcessor) organisation, a UMA system normally has a low number of cores and a single main memory interface interleaved among processing elements. Figure 2.2 shows a simple shared-memory architecture with a (limited and small) number of processing elements, each equipped with two processing units. All processing elements are directly connected to the shared main memory through the interconnection network, all having equal access time.

When the shared-memory access time is not equal for every processing element, but depends on the relative main memory location distance, we have a *Non Uniform Memory Access* (**NUMA**) system. The main memory interface is thus the union of all local memory modules and every processor can address both its own local memory and any other (remote) module memory (see Figure 2.4): access time to local memory will be much lower than access time to remote memory modules. This is because accesses to remote memory utilize the interconnection network, which is clearly slower than the dedicated links inside a processor. With NUMA, maintaining cache coherence across shared memory has a significant overhead, and trying to keep a consistent memory

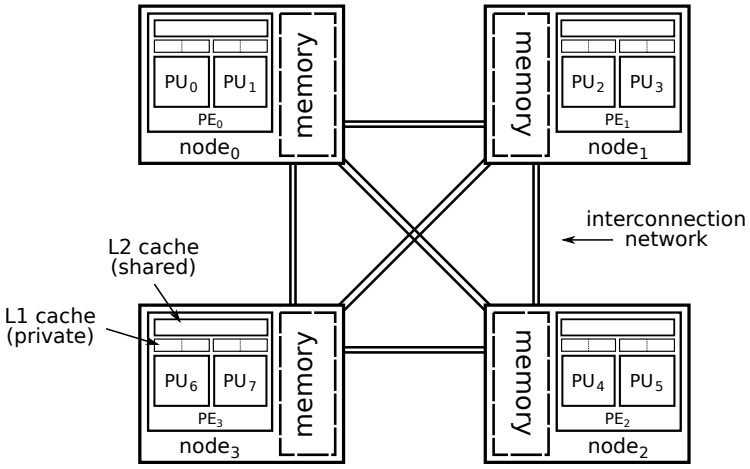


Figure 2.4: *A NUMA system organized with four computing nodes, each equipped with a processing element featuring two computing units. Each node has its own memory hierarchy: private L1 caches (L1*d*, L1*i*), shared L2 cache and local memory. Through the interconnection network, any processor can address any other node's memory (remote memory), with different latency and access times*

image when more than one cache stores the same memory location may exacerbate the inter-processor communication infrastructure.

In this respect, cache coherency is not for free: it largely affects synchronisation costs and may require expensive performance tuning that largely increases programming complexity. Assuming a cache-coherent NUMA architecture (cc-NUMA), data structures allocation in memory should depend on how they are placed and moved across the system.

Memory ordering and memory consistency models

Memory ordering describes the order in which *Load* and *Store* memory operations are performed. For instance, with an *in-order* behaviour a processing element computes instructions in the same order they are specified in a program (program order). A system that reflects this behaviour is said to have a *sequential consistency* memory model. If data dependencies are respected, memory ordering might be changed, either as a result of compiler optimisations, or as a consequence of dynamic optimisations performed at run-time by the

processor. An *out-of-order* behaviour caused by compile-time or run-time optimisations may not be an issue on programs running on single-core machines, and it allows to better exploit CPU resources, with a notable improvement of the computation's service time. A system that reflects this behaviour is said to have a *relaxed consistency* memory model, which enables the use of optimizations to eliminate some of the constraints that sequential consistency places on the overlap and reordering of memory operations.

On some concurrent executions, the memory ordering must be guaranteed: such condition can be achieved by forcing every running process to wait for all running memory operations to complete, before executing a `Store` operation. A *memory barrier* is a special instruction that enforces an ordering constraint on memory operations, and guarantees that instructions issued prior to the barrier are performed before instructions issued after the barrier.

A *consistent* memory wants the entire shared memory accessed by all processors to be identical. For this requirement to be effective, a memory system should always return the *last* value written in each location, including caches or local data copies. This clearly defines an *order* in which operations performed on shared memory should be made visible to individual processors in a multiprocessor system. At the same time, a sequential semantic for memory operations allows the programmer to assume that all memory operations will occur one at a time, in the sequential order specified by the program (i.e., *program order*). In simpler terms, consistency defines the temporal order in which two memory operations, either from a single thread or from different threads, will have their effect on memory locations.

A memory model not only influences many aspects of system design, but also bridges the gap between the behaviour expected by the programmer and the actual behaviour of the program. Furthermore, a memory model affects the design (and complexity) of programming languages, compilers and the underlying hardware.

In cache-based systems, a coherency protocol ensures the serialization of write operations and subsequent read or write accesses to update data. These

updates are propagated to remote sites, and the coherency protocol is responsible of keeping multiple copies of cache's blocks consistent to all processors. With this view of a cache coherence protocol, a memory consistency model decides when a new value can be propagated to any given processor [2].

Sequential consistency provides a simple view of the system to programmers, while imposing two strict requirements: memory operations executed by each individual processor must follow the order specified by its program, and a single sequential order among operations from all processors is also enforced. This concept extends the uniprocessor behaviour to shared-memory multiprocessor models, and was first formalised by Lamport in 1979 [95].

For example, sequential consistency could be implemented on a multitasking uniprocessor, where a single sequential processing element allows one process at a time (not in any specific order) to execute its instructions. Thus a process p_i runs until a context switch to another process occurs. On every context switch, any outstanding memory operation must be completed before switching to the new thread.

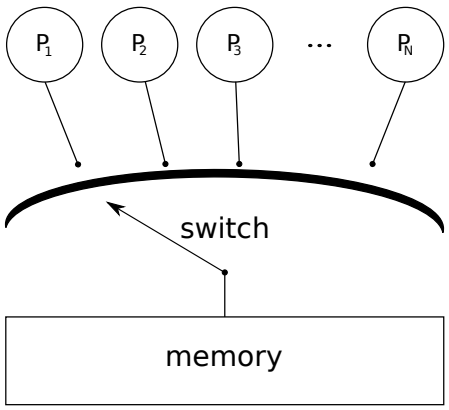


Figure 2.5: *Sequential consistency implementation: each processor P_i performs its next memory operation in its program order; the switch selects one processor at a time and allows it to complete one memory access.*

Figure 2.5 shows an abstraction of the sequential consistency model, as it could be implemented on a set of processors P_i: every processor performs memory accesses following its program order. The switch randomly selects one processor and allows it to complete its memory access, then repeats this process as

long as requests exist.

Although appealing conceptually, sequential consistency can result in very poor performance on modern processing systems. If program order must be ensured, an acknowledgement message from memory must be generated every time a write operation has completed. When cache-coherency is also enforced, invalidate or update messages for all cached copies will be generated for each write to memory. The second requirement for sequential consistency, conveniently defined as *write atomicity*, enforces a single sequential order among all operations. In a cache-based system this means that writes to the same memory location must be serialized, and the written value can only be made available after it has become visible to all processors, forcing all other operations to wait for the acknowledgement message before continuing. In both cases, the performance of these implementations does not scale up with increasing core count: a large amount of communication is needed to maintain consistency, while write operations ideally orchestrated by a single switch (Figure 2.5) become a real bottleneck.

If performance and correctness are not to be touched, some optimizations enable a reordering of read and write operations without violating sequential consistency. *Optimisation* techniques, either by hardware or compiler, are applicable to sequentially consistent systems with hardware support for cache coherency. These techniques allow the overlapping of memory operations to improve performance and tolerate memory access latencies [145].

One common hardware technique that potentially boosts performance combines prefetching of values that would otherwise be delayed due to consistency model constraints, and a speculative execution that allows a processor to schedule `read` instructions that appear later in program order (e.g., a branch prediction), overlapping their execution with operations preceding them. Values of later references are loaded into cache and are kept coherent until the processor actually performs the instructions [62]. In order to be effective, this technique requires a memory system with coherent caches, so that the results returned by the speculation is guaranteed to be correct. Sequential consistency is maintained insofar as reads and writes are performed in program order: in this regard, the prediction performed by the processor has to be correct. In case a

cache line gets invalidated or updated before the operation has been issued, the instruction is rolled back and reissued. Also, compilers can detect memory operations that can be reordered and perform simple optimisations, such as register allocation, sub-expression elimination, loop unrolling or software pipelining, without violating sequential consistency.

These techniques permit to hide memory access latencies by buffering and pipelining instructions, without violating sequential consistency constraints, but actual gains in terms of performance are to be assessed for each case, and might not reach performance gains obtainable using relaxed consistency models. For a detailed description of these techniques we refer to the referenced literature.

Relaxed consistency models

Relaxed consistency models enable latency tolerant memory optimizations by reducing ordering constraints, that is, relaxing the ordering among reads (R) and writes (W) performed by a single processor to the same address. We recall that the order enforced by a consistency model can be presented in terms of when an operation is allowed to execute: a `read` (or `load`) operation is considered performed when the return value is bound (i.e., *assigned*) and can not be modified by other `write` operations. Similarly, a `write` (or `store`) is considered performed when the written value is visible to all processors.

Two widely implemented memory consistency models are *total store order* (TSO) and *processor consistency* (PC): they both allow a read to be reordered with respect to previous writes to the same address, but differ in their "strictness" because TSO lets a `read` to *only* return the value of its own processor's `write`, while PC lets a `read` to return the value of *any* `write` to the same address. Relaxing the program order from a `write` followed by a `read` can substantially improve performance at the hardware level by effectively hiding the latency of a write operation. The *partial store ordering* model (PSO) further relaxes the program order requirements, by eliminating ordering constraints between writes to different locations. Just like in TSO, PSO allows writes not yet serialized to

be read in advance by the same processor, but prohibits to read the value of another processor's write before the written value is visible to all other processors, in order to maintain `write` atomicity.

More *relaxed* memory consistency models seek to preserve only the orders that programmers "require", but facilitate higher performance by allowing more hardware and software optimizations — performed by the compiler and the runtime system, respectively —, with the drawback of adding complexity and a higher risk of coding errors. All program order requirements can be relaxed (i.e. between all operations to different locations), so that memory operations following a `read` may be overlapped or reordered with respect to the `read` itself, hiding the consequent latency by implementing non-blocking reads in the context of *in-order* or *out-of-order* processors.

Weak Ordering (WO) is the seminal relaxed consistency model; Release Consistency (RC) is an extension of WO. In a WO model, program order between two operations can be maintained using synchronization operations (i.e. *fences* or memory barriers) [51]. Between synchronization operations, the order of accesses does not have to be preserved, but the synchronisation mechanism guarantees that instructions issued prior to the barrier are performed before instructions issued after the barrier. In this mechanism, operations that are not labelled as a synchronization can be arbitrarily reordered between synchronization operations. The RC model further relaxes WO model and splits synchronization operations into *acquires* and *releases* [63]. An acquire is a `read` operation that is performed to gain access to a set of operations or variables. A release is a `write` operation that grants permission to another processor to gain access to some operations or variables.

Relaxed consistency models provide considerable reordering freedom to hardware and compiler: memory requests ordering are related to synchronisation points in the program that guarantee correctness and consistency. The majority of modern processors support at least one of the relaxed memory consistency model.

2.2 Memory allocation

Memory allocation has a significant influence on data cache miss rates of applications, substantially affecting overall performances. This problem widens with multi-core architectures, that often exhibit a *cache-coherent* Non-Uniform Memory Access (**cc-NUMA**) and weak memory consistency models (e.g. Total Store Ordering or Weak Ordering) [2], where many threads request memory chunks and free used memory. The growing need of improved performance and the continuing constraints on power and scalability in multi-cores, led system designers to look to heterogeneous architectures, that incorporate specialized processing capabilities to handle particular tasks, bringing together CPUs and hardware accelerators such as GPUs. When several different co-processors are put together, memory management becomes a dramatic bottleneck, that severely limits program scalability, robustness and overall performance. The non-optimized access to memory space can become a serious issue, especially in fine-grained parallelism where data structures allocation and their mapping onto physical memory is a key issue for offloading computations onto hardware accelerators.

Memory affinity in shared-memory architectures

UMA and NUMA architectures basically differ on the way shared memory is organised and distributed among processing units. Also, the "size" of the parallel architecture characterises both systems: parallel machines with a small number of processors have a simple interconnection network (such as bus) that links processors and shared memory, typical of a UMA architecture; more complex networks are required to efficiently connect a large number of processors, often resulting in decentralized organisations typical of NUMA configurations (as in Figure 2.4). In the latter case, the distance between a processor and the shared memory strongly affects the memory access time, thus having a significant impact on applications' performance, so much that they would not be comparable against the same application run on a UMA architecture. For this reason, when

dealing with NUMA architectures the concept of *memory affinity* assumes a fundamental role: each running process (or thread) should favour the use of the nearest (local) memory and limit the traffic towards other (remote) memories.

For memory affinity to be effective, two main aspects should be properly managed: where memory for an application is allocated (i.e., data structures allocation), and how processes (or threads) are assigned (*pinned*) to computing units. In UMA architectures memory affinity is not required to obtain good performance results: the physical address space is interleaved among processing elements, that can exploit the aggregate bandwidth of the whole memory.

Traditional data mapping strategies used by most operating systems to allocate memory in NUMA architectures are *first-touch* and *next-touch*. A first-touch strategy allocates memory on the NUMA node closest to the processor that generated a page fault exception (i.e., the first processor that "touches" the page). This strategy would be optimal in a NUMA system, if it was safe from data races and thread migrations. When these events happen, the first-touch allocation policy provides non-optimal results: in many parallel applications one thread initializes the data and forces page allocation on a single NUMA node, leading to an increased number of memory accesses (either local or remote) to that node. This strongly affects also the performance of other threads, because memory pages are not migrated during execution. Another drawback with this behaviour is that performance gain is valuable only when the applications have a regular data access pattern. When using next-touch, pages are migrated between nodes according to memory accesses to them. However, if the same page is accessed from different nodes, next-touch leads to excessive data migrations. In case of thread migrations between nodes this issue is even amplified.

Operating Systems now provide special kernel-level system calls specific for managing and customising memory allocation and thread placement on NUMA architectures. Additional support is given by libraries that wrap kernel's system calls and separately manages placement of threads to CPUs and placement of memory [12, 29, 89]. These approaches permit to define specific *memory policies* that combine allocation strategies and thread scheduling to meet

the different access patterns during the applications computation stages: *libnuma* and *hwloc* allow developers to apply memory policies to specific memory ranges, whereas *numactl* applies one memory policy to the whole application. Memory management policies include a strict allocation (*bind*) into a given node, round-robin memory allocation among and non-strict (*preferred*) binding to a node.

Several works try to address this problem, often proposing profile-guided page allocation for parallel programs [44, 105] (i.e. obtain an approximate memory access trace by executing and analysing an application's memory access pattern), or by proposing page migration techniques aimed at obtaining a trade-off between remote access latency and the migration overhead [83, 114]. Minas [126] is framework for cc-NUMA platforms that provides either automatic or explicit memory affinity management. The explicit tuning is based on an API called *MAi*, that exposes simple functions to manage allocation and data placement, extending those functionalities proposed by *libnuma* and other O.S. specific system calls. Minas ANSI C functions describe a set of custom memory policies designed for optimised array and matrix allocation on cc-NUMA platforms. Each set of policies affect the way memory pages are placed on physical memory blocks.

Experiments conducted with Minas showed modest performance gains with respect to first-touch policy or against other implementations based on the above discussed libraries: when applying different data mapping policies, allocated data structures are sliced among memory interfaces, leading to unbalanced memory accesses among interfaces (i.e., all memory controllers handle different numbers of memory accesses). In this context, it gets difficult to exploit the aggregate memory bandwidth. However, no mapping policy solution is suitable to maximise performance of all applications in all NUMA architectures, but memory affinity and memory access locality remain the most important metrics for performance improvements [47].

Thread pinning means to "attach" a thread to a specific processing unit (or core), so that the O.S. scheduler is guaranteed to maintain the mapping during the execution. In presence of a parallel program executed on a NUMA system, the thread-to-core pinning policy may affect performance because of the

shared level of cache present in multi-core chips: sharing a cache may reduce the performance if the working set of all the threads does not fit into the shared cache, but may also increase the performance in case of frequent data exchange among threads, as it can better exploit automatic cache coherence. Pinning also affects memory performance and plays an important role when combined with memory allocation policies.

Issues in memory management

Heterogeneous systems present several memory issues, because each diverse hardware co-processor in use has different memory requirements. Many of these problems are also present in multi-threaded environments, where a dynamic memory allocator should avoid or limit harmful events that would corrupt applications' performance.

False sharing occurs when threads on different processors modify variables that reside on the same cache line without actually sharing data (Figure 2.6). This invalidates the cache line and forces an update, which hurts performance in parallel applications: the frequent coordination required between processors when cache lines are marked invalidated requires cache lines to be written to memory and subsequently loaded. False sharing increases this coordination and can significantly degrade application performance.

False sharing may be *actively* or *passively* induced by the allocator: actively induced false sharing happens when a single-heap allocator can give many threads parts of the same cache line; passively induced false-sharing happens when a number of objects is allocated within a cache line, and then these objects are passed to different threads, one by one. To avoid false-sharing an allocator should pad out every memory request to the size of the cache line, but this could cause a dramatic increase in memory consumption and degrade space locality and cache utilisation (ex: 8 bytes objects padded to 32/64 bytes of cache line).

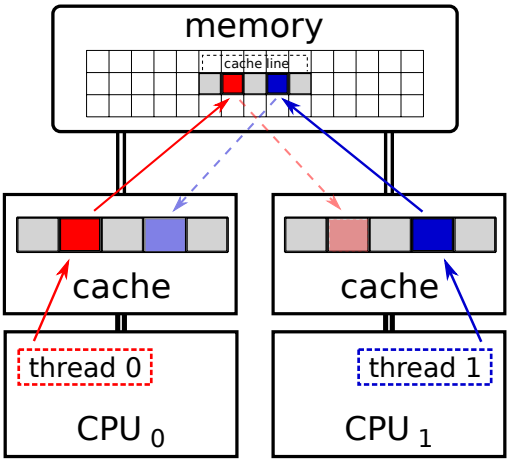


Figure 2.6: *False sharing occurs when threads on different processors modify variables that reside on the same cache line, without actually sharing data*

Fragmentation happens when storage space is used inefficiently: free memory is no longer contiguous but becomes *fragmented* into sparse blocks of memory of smaller and smaller size, leaving a good percentage of the total memory unallocated but unusable for most typical scenarios (see Figure 2.7). This reduces capacity and leads to allocation errors. Fragmentation can be distinguished in *internal*, *external* and *data* fragmentation. Internal fragmentation occurs when the unusable memory is contained within the allocated region: allocated memory may be slightly larger than requested memory; this size difference is internal to a partition, but not being used. Internal fragmentation is difficult to cope with: usually the best way to remove it is with a design change. For example, memory pools drastically cut internal fragmentation by spreading the space overhead over a larger number of objects. External fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. Data fragmentation occurs when a collection of data in memory is broken up into many pieces that are not close together. It is typically the result of attempting to insert a large object into storage that has already suffered external fragmentation.

Blow-up can be seen as a particular kind of fragmentation, where the quantity of unavailable memory dramatically increases. It can be either unbounded

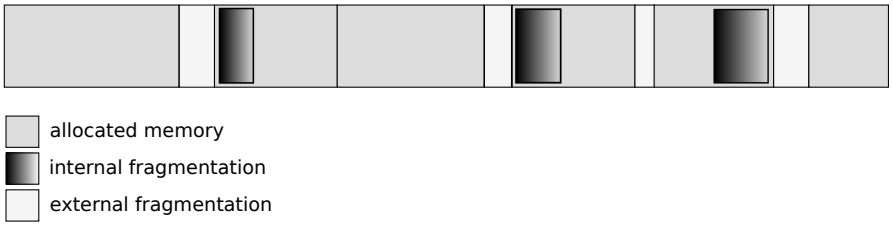


Figure 2.7: *Memory fragmentation: allocated regions are spaced out by small blocks of memory unsuitable for being reused*

or can grow linearly with the number of processors. On systems with virtual memory, large allocations only need to be contiguous in virtual address space, not in physical address space. Thus the problem of fragmentation is reduced.

2.2.1 Memory allocators - Literature review

Many relevant works address the problem of dynamic memory allocation for multi-threaded systems, but none of them are designed to manage dynamic memory in heterogeneous systems, nor they specifically address memory issues in NUMA architectures.

The *Hoard* memory allocator [25] is designed to meet the requirements for a scalable concurrent memory allocator. The allocation is done using multiple per-processor heaps, which avoids false-sharing and reduces the synchronization overhead. Each heap contains one or more *superblocks*, where each superblock handles blocks of one size class. Memory requests are mapped to the closest matching size in a fixed set of size-classes, which bounds internal fragmentation. When a processor heap is found to have too much available space, one of its superblocks is moved to the global heap. To avoid heap blowup, freed blocks are returned to the heap they were allocated from and empty superblocks may be reused in other heaps.

Several solutions are loosely based on the Hoard architecture. Michael presented a completely lock-free allocator [110] that adds immunity to deadlock and starvation: here each per-processor heap contains one active and at most one inactive, partially filled superblock per size-class, plus an unlimited number of full superblocks. Partially filled superblocks are stored globally in per-size-class FIFO queues. The results showed in [110] demonstrate the efficiency

of the proposed solution. Tsigas et al. also developed a lock-free allocator inspired by Hoard [65]. The results of the experiments conducted show that Hoard-inspired allocators induce very little false-sharing and yield very good performances in terms of scalability. *Streamflow* is a locality-conscious, multi-threaded memory allocator proposed by Nikolopoulos et al. [135], which favours the exploitation of temporal and spatial object access locality.

The Slab allocator [27] is an object-caching kernel memory allocator designed as a "loose confederation of independent caches". An allocation request would retrieve memory in a best-fit manner; *malloc/free* operations are fast, constant-time operations, as they consist of moving a cache object (i.e. a *buffer*) to or from a list of caches of different, predefined sizes. It primarily optimizes dynamic allocation of small objects while using cache-conscious page-block layout. These optimisations are accomplished using scalable synchronisation mechanisms and segregated heaps with header-less objects, pretty much like the above-mentioned Hoard's and Michael's allocator.

Intel provides a scalable allocator [93] as part of the TBB framework [82]. This allocator has common roots with Hoard, Streamflow, and other state-of-the-art concurrent and sequential allocators. But mostly, it is derived from the scalable transactional memory allocator developed as part of the McRT research program at Intel [79]. Memory is requested in 1MB chunks, using operating system primitives like `mmap` (in Unix-like systems). It ensures that memory is reused, so that new blocks are requested only when a thread can't find any free object in the blocks of its own heap and there are no available blocks in the global heap. TBB allocator uses thread-private heaps to reduce false-sharing, provide better scalability and lower the complexity of synchronisations. Like in many other solutions, heaps use different storage bins (i.e. buffers) to allocate objects of different sizes, so that memory requests are rounded up to the nearest object size and internal fragmentation is minimal.

The most popular among GPUs optimized allocators are *XMalloc* and *ScatterAlloc*. XMalloc [77] uses an approach similar to Hoard, introducing superblocks from which memory requests are satisfied, and uses atomic CAS operation for reducing synchronisation costs, as suggested by Michael [110]. XMalloc stores freed blocks in lock-free FIFOs, and tries to serve allocation requests from FIFO

queues before asking for new memory. XMalloc introduces a SIMD optimization, where memory requests are grouped together within a warp, thus eliminating the SIMD-level memory contention. ScatterAlloc [146] expands XMalloc by reducing simultaneous accesses from different threads to the same memory region: using a multiplicative hash function, it scatters memory requests to multiple pages. Concurrent accesses to the same memory region are drastically reduced and the allocation process is faster. FDGMalloc [161] further ameliorates XMalloc and ScatterAlloc. It also takes inspiration from Hoard, as it uses a superblock shared by all threads in a warp, but it does not use any header data for the superblock, except for one pointer to the next free chunk within the superblock. A voting algorithm is used to determine which thread has to be declared the worker thread, that is the one committed to execute the allocation request.

2.3 Blocking and non-blocking algorithms

In modern shared-memory multi-core architectures the efficiency of synchronization mechanisms is the cornerstone of performance and speedup of fine-grained parallel applications. The traditional way to implement concurrent data structures in multi-threaded applications makes use of mutual exclusion (`lock/unlock`) as a synchronization mechanism to ensure the correctness of concurrent updates. This approach has a number of disadvantages and typically involves various sources of overhead, which have a significant impact on performance as the parallelism degree increases [121]:

- mutual-exclusion can cause the whole system to come to a halt if circular dependencies exist. In such a case, every job is stalled waiting for another job to complete: this situation is called *deadlock*;
- mutual-exclusion can cause lower priority jobs to take precedence over higher priority jobs: this situation is referred to as *priority inversion*;
- mutual exclusion can needlessly restrict parallelism by serialising non-conflicting updates. Even using fine-grained locks to mitigate such effect,

cache performance may become a problem, along with the extra cost of acquiring and releasing these locks;

- lock contention limits scalability and adds complexity.

Non-blocking synchronization techniques are a valid alternative approach to lock-based synchronisation mechanisms, which have been designed to limit or avoid lock disadvantages: a non-blocking synchronization guarantees that even if a process (or thread) that is using a shared object stalls, fails, or even dies, *some* other process can use that object to make progress in a finite amount of time. This property is normally referred to as **lock-freedom**. This is a very different approach to that taken by lock-based mechanisms, in which a process will either spin or block until the contending operation is completed.

A stronger non-blocking guarantee of progress is called **wait-freedom**, which ensures that *every* process will make progress within some finite time [70]. This condition guarantees that no operation can experience permanent livelock. In practice, algorithms obeying the strict definition of non-blocking are not always necessary, considering also that obtaining efficient wait-free algorithms has a high cost in code complexity.

An alternative, weaker non-blocking condition is called **obstruction-freedom**, that guarantees progress for any thread that eventually executes in isolation [71]. A thread is considered to execute in isolation if it can run long enough without encountering a synchronization conflict from a concurrent thread. Like wait-freedom and lock-freedom, obstruction-freedom synchronization ensures that no thread can be blocked by delays or failures of other threads.

These techniques have been designed using a hardware-implemented class of atomic *read-modify-write* synchronization primitives provided by most modern architectures. Among them, `compare-and-swap` (CAS) takes three arguments in input: a memory location, a value that is expected to be read from the memory location, and a new value to write in the given memory location if the expected value is found. When using a CAS for synchronization, some computation is performed on the read value, and then a call to CAS writes the modified value while ensuring that the value read has not been changed in the meantime.

An early paper by Herlihy demonstrates that these atomic primitives are powerful enough to achieve wait-free (or lock-free) implementations of any linearizable data object [73, 74]. Unfortunately, these operations are not inexpensive, since they might fail to swap operands when executed and may be re-executed many times, thus introducing other sources of potential overhead, especially under high contention [121].

The most effective use of non-blocking synchronization has been the direct implementation of data-structure-specific algorithms. Michael and Scott [109] reviewed a number of such implementations for several common data structures. They concluded that where direct non-blocking implementations exist, they generally out-perform locking implementations in all cases. Lock-based implementations out-perform non-blocking constructions when processes do not experience long delays such as page-faults, context switches, or failures. Of course, the non-blocking constructions enjoy the advantages not related to performance, such as fault tolerance and freedom from deadlock.

2.4 Structured parallel programming

Parallel programming has always been related to HPC environments, where programmers write parallel code by mean of low-level libraries that give complete control over the parallel application, allowing them to manually optimize the code in order to exploit at best the parallel architecture. This programming methodology has become unsuitable with the fast move to heterogeneous architectures, that encompass hardware accelerators, distributed shared-memory systems and cloud infrastructures, highlighting the need for proper tools to easily implement parallel applications.

For this reason novel parallel programming tools have been introduced in the last decade, both by the academic and by the industrial world. To the current state of the art, a proper rewrite of the program is necessary if we want to exploit parallelism on heterogeneous architectures. In this case, a proper mix of ease of use and performance is still the main concern of researchers. It is widely

acknowledged that the main problem to be addressed by a parallel programming model is *portability*: the ability to compile and execute the same code on different architectures and obtain the same top performance [43]. A high-level approach to parallel programming is the right way to go if we want to address this problem, so that programmers can build parallel applications and be sure that it will perform reasonably well on the wide choice of parallel architectures available today [144].

Attempts to raise the level of abstraction and reduce the programming effort date back to at least three decades. Notable results have been achieved by the *skeletal* approach [37, 38] (a.k.a. *pattern-based* parallel programming), that has gained popularity after being revamped by several successful parallel programming frameworks. Despite some criticisms – mostly related to the limited amount of patterns that might not be sufficient to allow a decent parallelization of most algorithms – algorithmic skeletons success has been determined by the several advantages it has against traditional parallel programming frameworks. For instance, correctness and efficiency of parallel applications are guaranteed by the skeleton framework designer, as well as portability on different target architectures. In general, skeletons' higher abstraction level and parallel programming simplification have attracted more and more.

We briefly discuss here how parallel programming is commonly addressed by general-purpose programming languages, at a rather low-level, where the programmer has to actually deal with synchronisation and communication challenges. Following sections will focus on high-level parallel programming patterns, with an analysis of structured parallelism and a literature review.

2.4.1 Low-level parallel programming

Parallel programming languages are usually extensions to well-established sequential languages, such as C, Java or Fortran, where the coordination of multiple execution flows is either obtained by means of external libraries, linked at compile time to the application source code (e.g., *Pthreads*, *OpenMP*, *MPI*), or enriched with specific constructs useful to orchestrate the parallel computation (as in Java).

POSIX Threads (or Pthreads) [31] are one of the most famous low-level parallel programming APIs for shared-memory environments. They can be found in every Unix-like operating system (Linux, Solaris, Mac OS X, etc.) and other POSIX systems, giving access to OS-level threads and synchronization mechanisms. Since Pthreads is a C library, it can be used in C++ programs as well, though there have been much improvements in the C++11 standard aimed at facilitating shared-memory parallel programming. Considering that C++ is constantly being updated and maintained, novel C++ releases provide better parallelism facilities, with solutions clearly inspired by the structured parallel programming paradigm (e.g., execution policies and data parallel patterns in C++17[7]).

Message Passing Interface (MPI) [122] is a language-independent communication protocol used for programming parallel computers, as well as a message-passing API that supports point-to-point and collective communication by mean of directly callable routines. Many general-purpose programming languages have bindings to MPI's functionalities, among which: C, C++, Fortran, Java and Python.

Mainly targeted at distributed architectures, it offers specific implementations for almost any high-performance interconnection network. At the same time, shared-memory implementations exist, that allow the use of MPI even on NUMA and multi-processors systems.

MPI permits to manage synchronisation and communication functionalities among a set of processes, and provides mechanisms to deploy a virtual topology of the system upon which the program is executing. These features, supported by a rich set of abilities and functions, clearly require high programming and networking skills: MPI has long been the *lingua franca* of HPC, supporting a substantial majority of all supercomputing work scientists and engineers have relied upon for the past two decades. Nonetheless, MPI is at a very low level of abstraction for application writers: MPI means programming at the *transport* layer, where every exchange of data has to be implemented through *sends* and *receives*, data structures must be manually decomposed across processors and

[7] `http://en.cppreference.com/w/cpp/experimental/parallelism`

every update of the data structure needs to be recast into a flurry of messages, synchronizations, and data exchange.

OpenMP [124] is considered by many the *de-facto* standard API for shared-memory parallel programming. It is an extension that can be supported by C, C++ and Fortran compilers and defines an "accelerator-style" programming, where the main program is run sequentially while code is accelerated in specific points, running in parallel, using special preprocessor instructions known as `pragma`s. Compilers that do not support `pragma`s can ignore them, making an OpenMP program compilable and runnable on every system with a generic sequential compiler.

While Pthreads are very low-level and require the programmer to specify every detail of the behaviour of each thread, OpenMP allows to simply state which block of code should be executed in parallel, leaving to compiler and run-time system the responsibility to determine the details of the thread behaviour. This feature makes OpenMP programs simpler to code, with a risk of unpredictable performance that strongly depend on compiler implementations and optimizations.

OpenCL [88] is an API designed to write parallel programs that execute across heterogeneous architectures, and allows the users to exploit GPUs for general purpose tasks that can be parallelised. It is implemented by different hardware vendors such as Intel, AMD and NVIDIA, making it highly portable and allowing the code written in OpenCL to be run on different hardware accelerators. OpenCL represent an extension to C/C++ but must be considered a low-level language, focusing on low-level features management rather than high-level parallelism exploitation patterns. It has the capability to revert to the CPU for execution when there is no GPU in the system, and its portability makes it suitable for hybrid (CPU/GPU) or cloud based environments.

Java provides multi-threading and RPC support that can be used to write parallel applications for both shared memory and distributed memory architectures [119]. However, while being a high-level sequential language, parallel support is given in a very low-level fashion, possibly lower than OpenMP. Furtheromore, its sequential performance is usually lower than languages like C or Fortran and is therefore rarely used in high performance parallel programming.

Berkeley Unified Parallel C (UPC) [53] uses a partitioned global address space to extend the shared-memory model to a distributed memory setting: any processor can directly read and write variables on the partitioned address space, while each variable is physically associated with a single processor. Threads partition is subdivided into a *local* portion and a *shared* portion. Local data can be accessed only by the thread that owns the partition, while data in the shared portion are accessible by all threads. As in a shared-memory model, threads access shared memory addresses concurrently through standard read and write instructions, rather than through message passing. This programming model is still a low-level shared-memory environment, and uses barriers and locks to synchronize the execution flow. UPC extends the C language by adding synchronization primitives and a memory consistency model to operate over a shared address space.

2.4.2 Algorithmic Skeletons

A parallel *paradigm* can be thought of as a well-known pattern of parallelism exploitation. A single paradigm can be used to describe just a small part of an application, while multiple paradigms can be merged together to describe more complex constructs. This idea is at the core of structured parallel programming: expressing the parallel code as a composition of simple "building blocks". Following this approach, the application programmer defines the application, while a parallel programmer builds the parallel implementation of each building block (or their composition) according to the parallelism expressed by the application. Parallel paradigms can thus be divided into two conceptually different classes, depending on the way parallelism is exploited inside the pattern: *i) stream-parallel* paradigms and *ii) data-parallel* paradigms.

Algorithmic skeletons capture common parallel programming paradigms (e.g. Map+Reduce, ForAll, Divide&Conquer, etc.) and make them available to the programmer as high-level programming constructs equipped with well-defined

functional and extra-functional semantics [6]. Ideally, algorithmic skeletons address the difficulties of parallel programming (i.e., concurrency exploitation, orchestration, mapping, tuning) moving them from the application design to development tools, by capturing and abstracting common paradigms of parallel programming and providing them with efficient implementations: an example is a toolkit of code generation techniques and a pre-optimized run-time support. Differences between algorithmic skeletons and parallel design patterns lie mainly in the motivations leading to these two apparently distinct concepts and in the research environments where they have been developed: the parallel programming community for algorithmic skeletons and the software engineering community for parallel design patterns. Despite working disjointly, these two communities often reach to similar results that go for the benefits of both researches.

2.4.2.1 Stream parallelism

Patterns in this class exploit the presence of independent data items available as a stream (a sequence) of input elements to be processed concurrently, using a series of sequential or parallel stages. Streams processed in a stream parallel application are usually generated and consumed outside the application itself, but there are some cases where streams to be processed may also be generated inside the application. Input elements in a stream parallel program are said to be *consecutive*, meaning that they might not all available at the beginning of the computation (as it would be the case with traditional containers such as vectors). Rather, they appear at different moments in time, with item x_i appearing at time t_i, item x_{i+1} appearing at time t_{i+1}, and so on, with $t_i < t_{i-1}$. Clearly, the rate at which streaming items can be computed cannot be higher than the rate they arrive with. This characteristic could limit the parallelisation achievable and would obfuscate the benefits of a concurrent execution if the set of input elements is small.

In terms of performance, a stream-parallel pattern is effective in improving the throughput of the application, but results ineffective when the time needed to execute each single element of the stream (the latency) strongly dominates

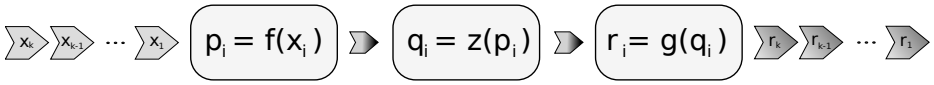


Figure 2.8: *A general* Pipeline *pattern with three stages*

the application performance. Stream parallel applications are present in many computing environments, from video processing (streams of frames) to network filtering (streams of packets), signal processing (streams of signal samples) and so on.

Pipeline

A pipeline represents the canonical stream. The *pipeline* pattern represents a functional *partitioning* of the sequential code, that is divided into multiple pieces executed *concurrently*[8] (see Figure 2.8). It is typically used to model computations expressed in *stages*, and in the general case a pipeline has at least two stages. Given a sequence $x_1, \ldots, x_k$ of input tasks and the simplest form of a pipeline with three stages, the computation on each single task x_i is expressed as the composition of three functions f, z and g, where the second stage (function z) works on the results of the application of the first stage, $z(f(x_i))$, and third stage applies the function z on the output of the second stage: $g(z(f(x_i)))$.

The parallelisation is obtained by concurrently executing all the stages onto different consecutive items of the input stream. In the general form, a pipeline with stages $s_1, \ldots, s_m$ computes the output stream

$$s_m(s_{m-1}(\ldots s_2(s_1(x_k))\ldots)), \ldots, s_m(s_{m-1}(\ldots s_2(s_1(x_1))\ldots))$$

The sequential code has to be described in terms of a composition of functions, where the output of each stage is sent to the next one, respecting the function ordering. The number of functions limits the maximum degree of parallelism of the application, meaning that with a 3-stages pipeline will exploit at most three processors. The semantic of the pipeline paradigm ensures that all stages will execute in parallel.

[8]Tasks can start, run, and complete in overlapping time periods. This does not mean they will necessarily be running at the very same instant.

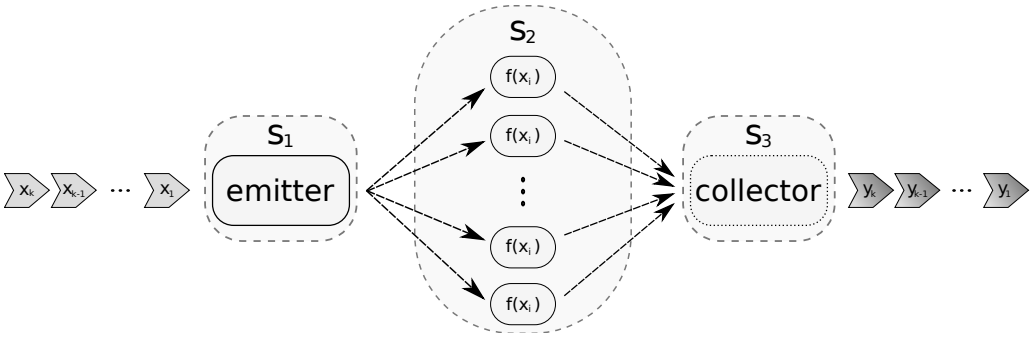


Figure 2.9: *A simple* Farm *pattern with an* optional *collector stage, whose stroke is dashed. Background light-gray boxes show that a farm pattern can be embedded into a three-stages pipeline*

Farm

The farm pattern represents a functional *replication* of a code segment, to be intended as a replication of the business logic of the application (see Figure 2.9). A farm is often used to model *embarrassingly* parallel computations, and is generally considered "simple" to exploit, as it does not require to rewrite the original code and allows for optimal task concurrency. The application programmer that uses a farm pattern only has to take care of a proper load balancing among the function replications, and should select an appropriate *degree of parallelism*[9] that does not lead to inefficiencies but fully exploits the available hardware parallelism.

Given a function f that represents an application's business logic, and given a sequence $x_1, \ldots, x_k$ of input tasks, the farm computes the output stream

$$f(x_k), f(x_{k-1}), \ldots, f(x_1)$$

The basic idea is similar to the pipeline: computing more input elements at the same time. For instance, a basic farm pattern can be embedded into a two (or three) stages pipeline, where the first stage applies some function to the input streams items and then *emits* tasks to a pool of *workers*, that would compose the second stage. A possible third stage (a *collector*) gathers workers' output and rebuilds a final result. More complex combinations of both patterns are

[9]The number of active threads (or processes) that actually execute at the very same time

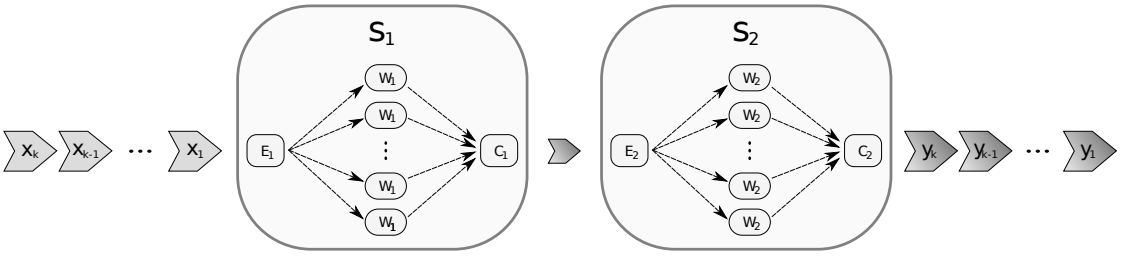


Figure 2.10: *A combination of pipeline and farm patterns, where each stage of a two-stage pipeline is a farm. In this figure, subscripts in farms components are used to distinguish the two farms*

possible, such as a *pipeline of farms*, where each stage of the pipeline is, in fact, a farm (as in Figure 2.10).

Task parallelism consists of running the same or different tasks on different executors (i.e., cores, machines, etc.). Tasks are concretely performed by threads or processes, which may communicate with one another as they execute. Communication usually takes place to pass data from one thread/process to one or many others. Task parallelism does not necessarily concern stream parallelism, but there might be the case in which the computation of each single item in a input stream embeds a parallel nature, that can efficiently be exploited to speedup the application. The *farm* pattern is a typical representation of such class of patterns.

2.4.2.2 Data parallelism

These paradigms extract parallelism inside a computation by partitioning the data structures and exploiting parallelism in the computation of different subtasks resulting from the partition of the input task. In other words, a data-parallel computation performs the same operation on different items of a given data structure at the same time. Opposite to task parallelism, which emphasizes the parallel nature of the computation, data parallelism stresses the parallel nature of the data[10].

[10]We denote this difference by using a zero-based numbering when indexing data structures in data parallel patterns.

Formally, a data parallel computation is characterized by partitioning data structures and function replication: a common partitioning approach divides input data between the available processors. This partitioning step is often followed by the replication and execution of some, mostly independent, operations across these partitions. Depending on the algorithm, there might be the case in which data dependencies exist among partitioned subtasks: many data-parallel programs may suffer from bad performance and poor scalability, because of a high number of data dependencies or a low amount of inherent parallelism.

Data parallelism also supports *loop-level* parallelism, where successive iterations of a cycle working on independent or read-only data are concurrently executed on different flows (i.e., threads or processes). In terms of performance, the more processors available, the finer the data partitioning, thus the higher the degree of parallelism and possibly the performance obtainable with a data parallel pattern. In reality, these conditions might not always be applicable: there exists a lower bound on the number of partitions, determined by the items that compose the data structures. Moreover, the smaller the partition the smaller the computation time per worker: this approach becomes feasible only up to a certain point.

Here we present two widely used instances of data parallel patterns, namely the *map* and the *reduce* patterns. Other data parallel pattern exists, that basically permit to apply higher-order functions to all the elements of a data structure. Among them we can mention the *fold* pattern and the *scan* (or prefix sum) pattern. The *stencil* pattern is a generalisation of the *map* pattern, and under a functional perspective both patterns are similar, but the stencil encompasses all those situations that require data exchange among workers.

Map

The map pattern is a straightforward case of data parallel paradigm: given a function f that expresses an application's behaviour, and a data structure X of known size (e.g., a vector with n elements), a map pattern will apply the function f, to all the elements $x_i \in X$:

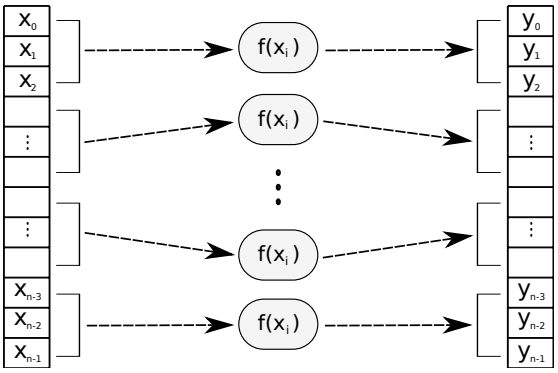


Figure 2.11: *A general representation of a* Map *data parallel pattern. Input data structure is partitioned according to the number of workers. Business logic function is replicated among each worker*

$$y_i = f(x_i) \qquad \forall i = 0, \ldots, n-1$$

Each element y_i of the resulting output vector Y is the result of a parallel computation.

The degree of parallelism of a map pattern is a critical parameter, and should carefully be chosen depending on the input data structure's size and the granularity of the computation: when it is too small, it might hinder the full exploitation of all computing power, resulting in a limited speedup, while a excessive degree of parallelism may lead to inefficiencies as some parallel workers are likely to remain idle.

Notable examples that naturally fit this paradigm include some vector operations (*scalar-vector* multiplication, vector sum, etc), *matrix-matrix* multiplication (in which the result matrix is partitioned, the input matrices replicated), the *Mandelbrot Set* calculation, and many others.

Reduce

A reduce pattern applies an *associative* function ($\oplus$) to all the elements of a data structure. In Figure 2.12, given a vector X of length n the reduce pattern computes

$$x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}$$

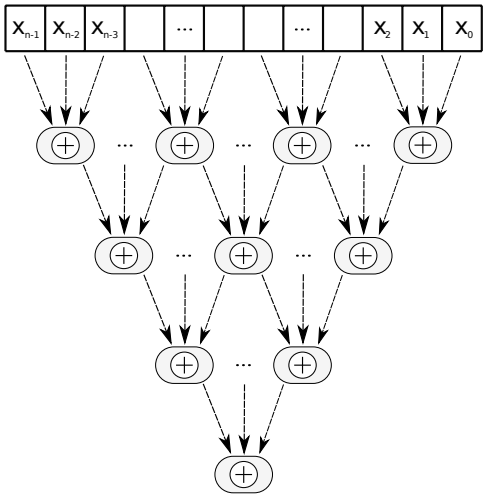


Figure 2.12: *A general* Reduce *pattern: the reduction is performed in parallel organising workers in a tree-like structure: each worker computes the function* $\oplus$ *on the results communicated by the son worker. Root worker delivers the reduce result*

Each worker locally computes a reduce over the assigned partition of the vector, in parallel, and propagates its result that is used to compute a global reduce. In a tree-like organization, leaf nodes compute their local reduce and then propagate results to parent nodes; the root node delivers the reduce result.

The composition of a *map* step and a *reduce* step generates the *Map+Reduce* pattern, in which a function is first applied, in parallel, to all elements of a data structure, and then the results from the map phase are merged using some reduction function (see Figure 2.13). This is an important example of the *composability* allowed by parallel patterns, which permits to define a whole class of algorithms for data-parallel computation.

The functional composition of these two parallel patterns is at the base of Google's *MapReduce* distributed programming model and framework, which exploits *key-value* pairs to compute problems that can be parallelised by mapping a function over a given dataset and then combining the results. Likely, it is the largest pattern framework in use, and spun off different open-source implementations, such as Hadoop [159] and Phoenix [127].

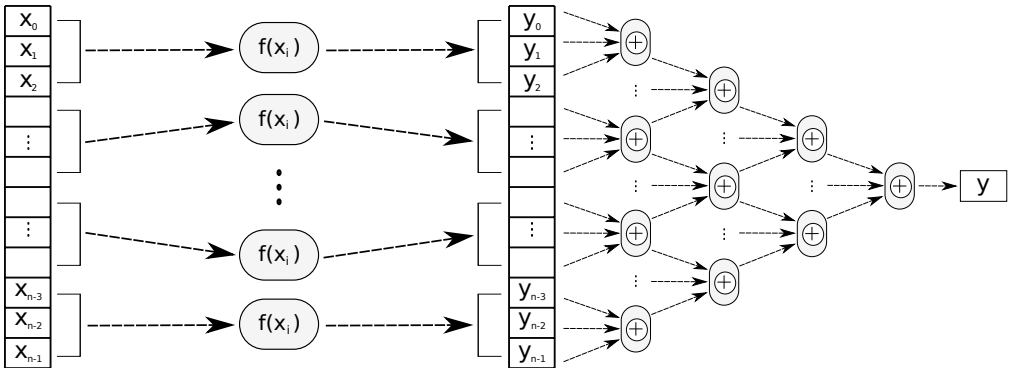


Figure 2.13: *A* Map+Reduce *pattern*

2.4.3 Literature review

The algorithmic skeletons defined by Cole [38] represent the first approach to structured parallel programming. He proposed 4 skeletons (Fixed Degree Divide&Conquer, Iterative Combination, Cluster and Task Queue) obtained identifying particular algorithmic techniques through an analysis of patterns that could perform well on the specific target machine. From his idea, many researchers focused on finding general and effective patterns that could be promoted to skeleton.

Here we discuss some skeleton libraries that target C/C++ as their execution language, and mostly focus on parallel programming for multi-core architectures. This choice, because the work presented in this thesis has been mostly conducted using C++ on shared-memory multi-core machines. For a broader survey of algorithmic skeletons, we refer to the survey from González-Vélez and Leyton [67].

One of the earliest proposals of pattern-based parallel programming was **P^{3}L** [40], a skeleton-based coordination language that manages the parallel or sequential execution of C code. P^3L comes with a proper compiler for the language, and uses implementation templates to compile the code into a target architecture. P^3L provided patterns for both stream parallelism and data parallelism.

SKELib [41] builds upon the contributions of P^3L by inheriting, among

other features, the template system. It differs from them because a coordination language is no longer used, and skeletons are provided as a C library. It only offered only stream-based skeletons (namely farm and pipe patterns).

SkeTo [106] is a C++ library based on MPI that provides skeletons for distributed data structures, such as arrays, matrices, and trees. The current version is based on C++ expression templates, used to represent part of an expression where the template represents the operation and parameters represent the operands to which the operation applies.

SkePU [54] is an open-source skeleton programming framework for multi-core CPUs and multi-GPU systems. It is a C++ template library with data-parallel and task-parallel skeletons (map, reduce, map-reduce, farm) that also provides generic containers type and support for execution on multi-GPU systems, both with CUDA and OpenCL. Smart containers automatically optimize communication, perform memory management and synchronize asynchronous calls using operand data flow. Preprocessor macros have been implemented to simplify the definition of skeleton's business logic. Eventually, by overloading `operator()` it is possible use skeleton functions as functor objects.

SkelCL [147] is a skeleton library targeting OpenCL. It allows the declaration of skeleton-based applications hiding all the low-level details of OpenCL. The set of skeletons is currently limited to data-parallel patterns: map, zip, reduce and scan, and it is unclear whether skeleton nesting is allowed. A limitation might come from the library's target, which is restricted to the OpenCL language: it likely benefits from the possibility to run OpenCL code both on multi-core and on many-core architectures, but the window for tunings and optimisations is thus quite restricted.

The Muenster Skeleton Library (**Muesli**) [36] is a C++ template library that supports shared-memory multi-processor and distributed architectures using MPI and OpenMP as underlying parallel engines. It provides data parallel patterns such as map, fold (i.e., reduce), scan (i.e., prefix sum), and distributed data structures like distributed arrays, matrices and sparse matrices. Skeleton functions are passed to distributed objects as pointers, since each distributed object has skeleton functions as internal member of the class itself. In Muesli, the programmer must explicitly indicate whether GPUs are to be used for data parallel

skeletons, if available.

Intel Threading Building Blocks (**TBB**) [82] defines a set of high-level parallel patterns that permit to exploit parallelism independently from the underlying platform details and threading mechanisms. It targets shared-memory multi-core architectures, and exposes parallel patterns for exploiting both stream parallelism and data parallelism. Among them, the *parallel_for* and *parallel_foreach* methods may be used to parallelise independent invocation of the function body of a for loop, whose number of iterations is known in advance. C++11 *lambda* functions can be used as arguments to these calls, so that the loop body function can be described as part of the call, rather than being separately declared. The parallel_for splits the range $[0, num\_iter)$ into sub-ranges and processes each sub-range r as a separate task using a serial for loop in the code.

FastFlow [8] is a parallel programming environment originally designed to support streaming applications on cache-coherent multicore platforms. It is realised as a C++ pattern-based parallel programming framework aimed at simplifying the development of applications for (shared-memory) multi-core and GPGPU platforms. The key vision of FastFlow is that ease-of-development and runtime efficiency can both be achieved by raising the abstraction level of the design phase. It provides a set of algorithmic skeletons addressing both stream parallelism — with the farm and pipeline patterns — and data parallelism — providing stencil, map, reduce pattern, and their arbitrary nesting and composition [9]. Map, reduce and stencil patterns can be run both on multi-cores and offloaded onto GPGPUs. In the latter case, the business code can include GPGPU-specific statements (i.e., CUDA or OpenCL statements). Leveraging the farm skeleton, FastFlow exposes a *ParallelFor* pattern [42], where chunks of a loop iterations having the form `for(idx=start;idx<stop;idx+=step)` are executed by the farm workers. Just like TBB, FastFlow's ParallelFor pattern uses C++11 *lambda* functions as a concise and elegant way to create a function object: lambdas can "capture" the state of non-local variables by value or by reference and allow functions to be syntactically defined when needed.

Libray	Programming Language	Programming Model	Skeleton nesting	Skeletons
P^3L	C	MPI	limited	map, reduce, seq, comp, pipe, farm, scan, loop
SkeLib	C	MPI	no	farm, pipe
SkeTo	C++	MPI	no	map, zip, gather, reduce, scan, distributed data structures
SkePU	C++	shared-memory / many-cores	-	map, reduce, map-reduce, farm
SkelCL	C / OpenCL	shared-memory / many-cores	limited	map, zip, reduce and scan
Muesli	C++	shared-memory / many-cores	limited	map, fold, scan,
TBB	C++	shared-memory	yes	parallel_for, parallel_while, parallel_do, parallel_pipeline, parallel_sort
FastFlow	C++	shared-memory	yes	pipe, farm, map, reduce, ParallelFor, stencil

Table 2.1: *Algorithmic Skeletons: comparison table*

2.5 HPC and Cloud computing

Cloud computing has widespread in the HPC community over the last decade, simply because it brings clear advantages: it is *scalable*, *on-demand*, *fast* and (supposedly) *inexpensive*. Such features (or benefits) of a cloud solution should be evaluated against those of Grid computing, which arose when the academic community needed to crunch large data sets (such as satellite data, genomic, nuclear physics, etc.), and it became easier for academic research institutions to move from large, mainframe-style supercomputers, towards a more scalable model, using lots of "relatively" inexpensive hardware in large clusters. In fact, a Grid is a hardware and software infrastructure that has the ability to coordinate resources and services distributed across several control domains, using standard, open, general-purpose protocols, in order to supply high quality of services [57, 58].

The picture of a Cloud system is not much far away from that of the Grid, because it can be described as a pool of *virtualized* computer resources that hosts a variety of different workloads, which can be deployed and scaled-out through the rapid provisioning of additional virtual resources [28]. A key point

here, that clearly distinguishes Clouds from Grids, is that a Cloud abstracts the underlying compute components, which range from hardware infrastructures to operating systems, to software packages. Computing nodes are virtualized through hypervisor technologies such as VMs, dynamically provisioned *on-demand*, as a personalized resource collection [32].

Cloud components are made available based on their usage: the so-called "*pay-as-you-go*" billing system, that clearly brings many potential advantages, especially for scientists who do not require 24/7 accessibility.

In their famous work "A View of Cloud Computing" [13], Armbrust et al. identified three key features that differentiates a Cloud system from existing supercomputing facilities:

- hardware provisioning appears to be infinite, available on-demand;
- no up-front commitment by cloud users is required, so that they can increase hardware resources only when there is an increase in their needs;
- pay-for-use of computing resources on a short-term basis, as needed, with the freedom of release them, as needed.

To these aspects, we want to add that in a Cloud system, users and developers have sweeping control of their clusters: this is useful when scientists have applications that require particular pieces of software to be installed at the system level. Conversely, when a Cloud provides a full software stack *as a service*, scientists never have to install a thing. The Cloud provider installs, maintains, and optimizes the application while scientists merely conform to a specific API.

2.5.1 Cloud service models

Depending on which service it delivers, a Cloud solution can be classified according to one of four categories [39, 153]:

IaaS At the lower level, the Cloud can be seen as a cluster of interconnected processing elements (either physical or, more often, virtual) that execute user applications: this includes hardware (e.g., CPUs), data storage and

networking facilities. In this case we speak of *Infrastructure as a Service* (IaaS). A big challenge introduced by HPC applications regards the enhancement of flexibility of Cloud platforms for resource management, in order to satisfy user needs. The most appropriate approach to ensure such flexibility is via virtualisation, that mainly involves either the generation of multiple virtual machine instances to partition the physical resources, or multi-tenancy techniques, which enable users to share application instances and treat them as independent ones.

PaaS At the platform level, Cloud solutions offer development environments in the form of full computing platforms. This model includes operating systems, execution runtime environments, databases and development tools. User applications are rooted in the software tools provided by the platform: in this case we speak of *Platform as a Service* (PaaS).

SaaS At the application level, the Cloud can be perceived as a set of services that can be accessed to compute results. We speak of *Software as a Service* (SaaS) in this case. For SaaS there is no client-side software requirement for the user: the services are reachable through an access point, like a web portal or a visualization tool, eliminating the need for local installation of a large variety of software tools and periodical updates.

Beside the canonical service models listed above, a further category of Cloud services can be described, called *Data as a Service* (**DaaS**): that fosters data delivery and accessibility, and concentrates in distributing data on-demand instead of software applications or hardware resources. It is reasonable to see DaaS as integrated into a Cloud-based application.

Amazon Web Services (AWS) which provides a centralized repository of public data sets, including archives of GenBank, Ensembl, 1000 Genomes, Model Organism Encyclopedia of DNA Elements, Unigene, Influenza Virus, etc. As a matter of fact, AWS contains multiple public datasets for a variety of scientific fields, from biology, astronomy, chemistry, etc.[11]. These datasets are delivered as services and can be seamlessly integrated into cloud-based applications [61].

[11] `http://aws.amazon.com/publicdatasets`

2.5.2 Cloud implementation models

The service models presented above can be delivered using four different implementation models: *private* cloud, *community* cloud, *public* cloud, and *hybrid* cloud. When building a cloud solution for scientific purposes, they have to be taken into account.

A **private cloud** is basically the same as owning and maintaining a traditional cluster, where the user has total control over the infrastructure and can configure the machines according to his needs. One big issue in such scenario is the absence of instant scalability, as the capacity of execution is limited to the physical hardware available. Moreover, the user is responsible for the energy consumption of the system and for hardware maintenance.

In a **community cloud** users are members of one organization, which has a set of resources possibly connected to resources owned by organizations. A user from one of the involved organizations arguably has access to the whole batch. The pay-per-use model may not be applicable to this type of cloud. One disadvantage here is the amount of available resources: they are limited to the number of machines that are part of the community cloud.

In a **public cloud**, the infrastructure is usually provided by a company tha "owns" the infrastructure: the *provider*. The advantage in this case is the access to an (arguably) unlimited number of computational resources, which can be allocated and deallocated on demand. The pay-per-use billing model permits to spend money only while using the resources. Access to up-to-date hardware is fairly guaranteed. There might be some concerns about data privacy, that can be addressed by a contract regarding data access.

A **hybrid cloud** can be used to extend the computational power already available with a connection to an external provider. This model permits to increase computing resources almost instantly, without up-front costs. In certain scenarios, it is also possible to configure the system to automatically allocate cloud resources. Data transfer might become a real bottleneck in this model, because the local cloud is connected to the public cloud through an Internet connection, where the bandwidth could be limited.

2.5.3 Performance

A typical public cloud computing environment is a hosted service available on the Internet. A hypothetical user and the providers cloud manager agree upon bandwidth and transfer ratio for both data movement to and from the cloud resources. The Internet connection speed and availability are an issue even for performance and reliability with a cloud computing service. Virtualisation and network interconnection play an important role in determining the overall performance of a cloud infrastructure. If the hypervisor does not have good resource management, it is possible that the physical resources are under- or overused. Also, the network interconnection of the VM is a concern: as the network resources are pooled among all the users, the network performance is not guaranteed.

To evaluate the performance of the cloud solution, it is necessary to compare it with a traditional system, whose performance must be well known and will be used as the basis for comparison. For a fair comparison, both the base and cloud systems need to present similar characteristics, mainly the number of cores (or PUs) of each system.

Other aspects play a part when evaluating of a cloud computing environment designed for science services:

- the deployment capability of providers to build solid environments in the cloud and the capability to execute the workload;
- the economic evaluation, performed to determine if it is better to use a cloud or to buy regular machines.

2.5.4 Existing Cloud platforms

Several cloud services are available in the market, which offer their services according to the service models listed in Section 2.5.1. Among them, Amazon Web Services, Google App Engine, Microsoft Windows Azure, and Open Stack are some of the most known cloud service providers.

Amazon Web Services (AWS)[12] has been operating since 2006; it is one of the most widely known cloud providers and runs a global web platform serving millions of customers every year. Many different kinds of services are offered, including storage, platform, and hosting services. Two of the most-used AWS services are the Amazon Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (S3). Amazon EC2 is an IaaS model, and may be considered the central part of Amazons cloud platform. It was designed to make web scaling easier for users. The interaction with the user is done through a web interface that permits to obtain and configure any desired computing capacity with little difficulty.

Amazon S3 is a storage solution for the Internet. It provides storage through web service interfaces, such as REST and SOAP. Inside the provider, the stored objects are organized into buckets, which are an Amazon proprietary method. The names of these buckets are chosen by the user, and they are accessible using a URL with a regular web browser. This means that Amazon S3 can be easily used to replace static web hosting infrastructure. One example of an Amazon S3 user is the Dropbox service, provided as SaaS for the final user, with the user having a certain amount of storage in the cloud to store any desired file.

Google provides its own Cloud platform[13], as a set of modular cloud-based services that cover compute, storage and networking facilities. Google Compute Engine is an IaaS model, because the system provides a robust computing infrastructure, while the user must choose and configure the platform components, and is responsible for administer and monitor the system. Google will ensure that resources are available, reliable, and ready to use.

Google App Engine (GAE) is Google's PaaS model, that enables users to build and deploy their web applications on Google's infrastructure, taking care of most of the management of the resources for you. Users need to develop their application using the supported programming languages, such as Python, Java and php.

[12] `http://aws.amazon.com/`
[13] `https://cloud.google.com/`

Microsoft started its initiative in cloud computing with the release of Windows Azure[14] in 2008, which initially was a PaaS to develop and run applications written in the programming languages supported by the .NET framework. Currently, the company owns products that cover all types of service models. Online Services is a set of products that are provided as SaaS, while Windows Azure provides both PaaS and IaaS.

Windows Azure PaaS is a platform developed to provide the user the capability to develop and deploy a complete application into Microsoft's infrastructure. To have access to this service, the user needs to develop an application following the provided framework. The Azure framework supports for a wide range of programming languages, including all .NET languages, Python, Java, and PHP. A generic framework is provided, in which the user can develop in any programming language that is supported by the Windows OS. Windows Azure IaaS is a service developed to provide the user access to VMs running on Microsofts infrastructure.

OpenStack[15] is a free and open-source cloud Operating System that controls large pools of compute, storage, and networking resources, providing a so called "Infrastructure as a Service" (IaaS) remote environment for end users, where the actual software runs as a service on reliable and scalable servers rather than on each end-user computer. OpenStack has a modular architecture and is made up of many different parts, most of which where progressively added to the platform's core over the years since its first release in 2010, while new modules and capabilities are continuously under development. Nevertheless, three modules can be identified as the key components of the platform: compute, storage and networking.

The primary computing engine behind OpenStack, called *Nova*, is used for deploying and managing large numbers of virtual resources that handle computing tasks. The storage system for objects and files is called *Swift*. With this storage system files are written to multiple disk drives spread throughout servers in the data center, providing a scalable redundant storage system where

[14] `https://azure.microsoft.com/`
[15] `https://www.openstack.org/`

data replication and integrity across the cluster is maintained via software. *neutron* is the module responsible for networking capabilities. It helps to ensure that each of the components of an OpenStack deployment can communicate with one another quickly and efficiently.

OpenStack APIs are open-source Python clients, and can run on most existing operating systems, including Linux, Mac OS and Windows. A command-line interface enables to access the platform's API through easy-to-use commands that can be included in scripts to automate tasks. Internally, each command uses cURL command-line tools, which embed RESTful APIs, and use the HTTP protocol. They include methods, URIs, media types, and response codes.

2.6 Discussion

In this Chapter we discussed some relevant aspects concerning shared-memory architectures and current trends on parallel programming facilities for such class of architectures. Given the level of details of certain aspects, it was not really feasible to introduce and thoroughly describe all of them, also because it is far from the scope of this thesis. We therefore decided, for the sake of readability, to keep this chapter at an introductory level, referring to cited references for more detailed analyses.

While an introductory discussion on distributed systems has been omitted, because again was deemed as out of the scope of this work, in Section 2.5 we provide an overview on the Cloud computing paradigm and its relations and use in HPC, presenting also some existing Cloud providers. References provided throughout the section will help in acquiring a more in-depth analysis of the field.

We also presented a brief literature review on the skeleton libraries of interest for our purposes, focusing mostly on solutions that target C/C++ language and the shared-memory computing paradigm (see Table 2.1). This choice was made because most of the work conducted and reported in this thesis revolves around the C++ language, on top of shared memory architectures (as discussed in Chapters 4 and 5).

2.6.1 Measuring Performance

There are two important objectives pursued when using high-performance computing systems: to have access to more memory and to obtain higher performance. It is easy to characterize the gain in memory, as the total memory is the sum of the individual memories. The speed of a parallel computer is harder to characterize: in terms of performance, what is generally expected from a parallel program is that its execution time when using p processing elements is about $1/p$ of its sequential execution time. This optimal case is generally known as *linear scalability*. Unfortunately, this is not always true.

A simple approach to define *speedup* is to run a program on a single processor and on a parallel machine with p processors, and to compare execution times. With T_s being the execution time on a single processor and T_p the time on p processors, the speedup S_p is defined as

$$S_p = \frac{T_s}{T_p}, \quad S_p \in (0, p]$$

An ideal case wants the speedup to grow linearly with the growing number of processors to use, but in practice we do not expect to attain that. To measure how far we are from the ideal speedup, the *efficiency* can be defined as

$$E_p = \frac{S_p}{p} \qquad \text{with } 0 < E_p \leq 1$$

Several factors impair the possibility of achieving an ideal behaviour: first of all, using more processors might necessitate more communication, whose overhead was not part of the original sequential computation. Also, unbalanced workloads cause some processors to be idle for some time, again lowering the actual speedup. Finally, not all problems are suitable to be parallelized. Some problems have been recognized as inherently sequential and will never benefit from a parallel implementation. In other cases, a problem may contain a substantial fraction of work which is not parallelizable, thus nullifying

the performance gain obtainable by parallelising the rest of the work. The *Amdahl's law* [10] states that the amount of non-parallelizable work in an application determines the maximum speedup achievable by parallelizing the application [10]. In other words, in an application where f percent of the total work is inherently sequential, the execution time for that part can not be reduced, no matter how many processors are available. So if T_s is the sequential running time, fT_s will be spent in the sequential fraction and $(1-f)T_s$ in the parallel fraction. Using p processing elements, in the ideal case the parallel fraction becomes

$$\frac{(1-f)T_s}{p}$$

and the achievable speedup equals to

$$S_p = \frac{T_s}{fT_s + (1-f)\frac{T_s}{p}}$$

If we imagine infinite processing resources ($p \to \infty$), the parallel fraction tends to zero, and the asymptotic speedup can be written as

$$\lim_{p\to\infty} S_p = \frac{T_s}{fT_s} = \frac{1}{f} \tag{2.1}$$

Equation 2.1 is the Amdahl's law and explains that an application having a certain percentage f of sequential fraction cannot achieve a speedup higher than $1/f$.

Amdahl's law, could even be considered optimistic. Parallelizing a code will give a certain speedup, but it also introduces *overheads* – a time loss with respect to the computation of the parallel application results – that will lower the speedup attained: the time needed to set up parallel activities is necessary to implement parallelism. When splitting a loop in a number of parts, each processor has to deal with a loop overhead related to the calculation of bounds and the test for completion. And this overhead is replicated as many times as there are processors performing such loop, so that the said overhead can be treated as a sequential part of the code.

Amdahl's law can be refined in such a way that overhead is taken into account: the time spent to compute the parallel fraction becomes $(T_s/p)+T_c$, where T_c is a constant that takes into account the overhead time. The Amdahl's law can thus be rewritten as:

$$\lim_{p\to\infty} S_p = \lim_{p\to\infty} \frac{T_s}{fT_s + (1-f)(\frac{T_s}{p} + T_c(p))} = \frac{1}{f + (1-f)(\frac{T_c(p)}{T_s} + \frac{1}{p})} \tag{2.2}$$

For this to be close to the ideal speedup, we need $T_c \ll (T_s/p)$ or $p \ll (T_s/T_c)$. In other words, the number of processors should not grow beyond the ratio of sequential execution time and overhead. If the overhead is negligible this reduces to equation 2.1, otherwise the asymptotic speedup results smaller than the one obtainable taking into account only the load of the sequential fraction.

Amdahl's law shows that a larger and larger numbers of processors would not pay off. In other words, it states that if there is a fixed computation requirement, which gets executed on more and more processors, this limits the expected speedup that parallelisation can provide, given a fixed data set size. In practice, normally this is not the case: *Gustafson's law* provides a more realistic assumption [68], saying that the sequential fraction is independent of the problem size, thus an increase of problem size can retain scalability with respect to the number of processors.

Let F_s be the sequential fraction time (the fT_s above) and F_p be the parallel fraction time (the $(1-f)T_s$ above): we can define the execution time of the parallel program as

$$T_p = F_s + F_p$$

The corresponding time for the sequential program is

$$T_s = F_s + p \cdot F_p$$

This gives a speedup of

$$
\begin{aligned}
S_p = \frac{T_s}{T_p} &= \frac{F_s + p \cdot F_p}{F_s + F_p} \\
&= F_s + p \cdot F_p \\
&= p - F_s \cdot (p - 1)
\end{aligned}
\tag{2.3}
$$

That is, speedup is now a function that varies linearly with the number of processors. If F_s is small, the speedup is approximately p. It may even be the case that F_s diminishes as p increases, together with the problem size. When this is true, the speedup approaches p monotonically with the growth of p.

Gustafson's law (equation 2.3) is based on the idea that if the problem size is allowed to grow monotonically with p, then the sequential fraction of the workload would not dominate. That is to say, the true parallel power of a large multiprocessor system is only achievable when a large parallel problem is applied.

Many applications cannot scale up to meet the time bound constraint due to physical constraints. In practice, the physical constraint is often the memory limitation arising with the increasing disparity between CPU speed and memory data access latency, making applications' execution time depend on the memory speed of the system [163].

2.6.2 Research niche

With this Chapter we introduced some of the aspects related to HPC technologies, namely hardware architectures and parallel programming facilities. In the core Chapters of this thesis we will extensively work on shared-memory architectures exploiting algorithmic skeleton libraries for obtaining best possible performance on such architectures, while performance portability is guaranteed by the skeleton libraries.

Most of the work on shared-memory, thread-level parallelism has been performed using the FastFlow library, despite many others provide support for share-memory multi-core architectures — which is the type of architecture we have been using for this work. The main reason behind the choice of FastFlow

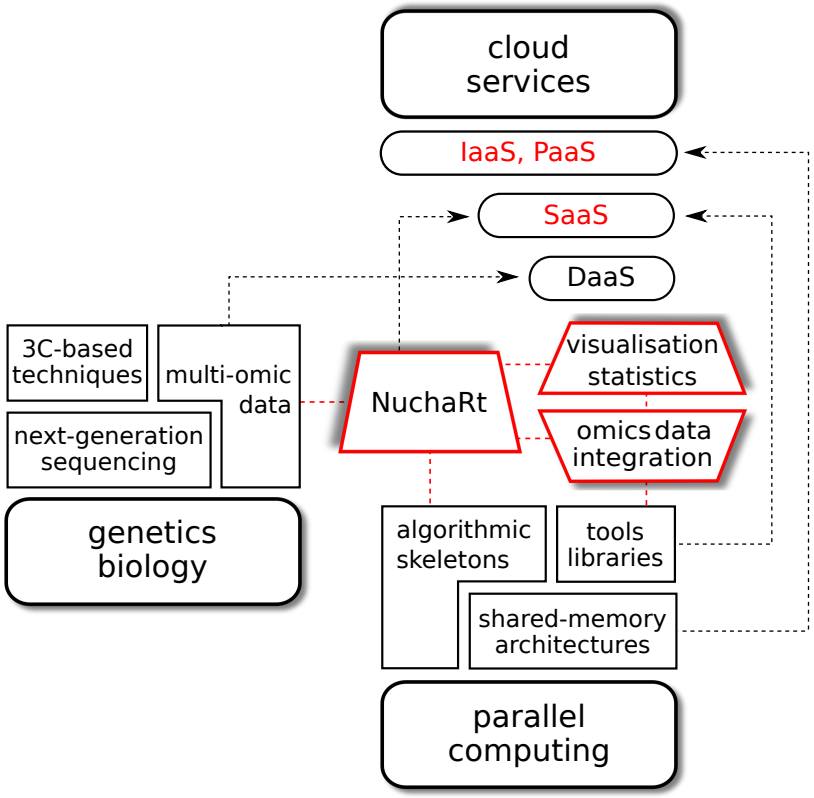


Figure 2.14: *A high-level view of our approach for heterogeneous genomic information processing and interpretation*

was we were already acquainted and experienced with FastFlow, and largely enjoyed the possibility to seamlessly parallelise `for` loops by means of lambda functions. We also evaluated our solution against TBB- and OpenMP-based implementations, so that we could have a comparison with parallel programming frameworks created by major industry powerhouses, as opposed to other algorithmic skeleton frameworks developed by academic institutions.

As a matter of fact, we will show that the performance achieved with the two alternatives we evaluated reached comparable results to those obtained with FastFlow. Main differences where found mostly in terms of ease of coding: OpenMP permits to easily achieve the desired result, by simply using the pre-compiler directives that permit to exploit the loop parallelism. On the other hand, Intel TBB implementation was not as much straightforward as the above: the `parallel_for` pattern needs more tuning and fixes with respect to the

FastFlow's one, and thread-local storage is not as easy to use.

The performance achievements on shared-memory architectures remain effective when we embed our C++ solution into the R environment, where we aim at making an application for genomic data analysis, usable by an heterogeneous range of scientist with different backgrounds, thus not necessarily familiar with the C++ language and parallel programming principles.

As Figure 7.1 shows, our R/C++ application will then be the key service for the integration of heterogeneous genomic information, and will be used in a cloud-based solution (Chapter 6), where the orchestration of various Bioinformatics software employed in genomic data processing permits to efficiently exploit a virtualised computing infrastructure: the benefits of this approach mostly come from the concurrency kept up among software services.

Chapter 3

Background on high-throughput genome sequencing and DNA exploration

These years witness an incredible explosion of the available molecular biology information. In particular, the integration and the interpretation of omics data in a systems biology view is complex, because actual representations rely on the genomic coordinates, discarding at first gene functional cooperation and renouncing to exploit the real conformation of the DNA[16] in the nucleus. In this view, the organization of the chromosome in the nucleus is extremely relevant to understand biological functioning — at the gene level as well as at the global nuclear level. The study of the organization of chromatin in the nucleus has shed some light on the spatial aspects of gene regulation, and will further enable the investigation of pathologies related to genome instability or nuclear morphology [24].

In this chapter we will provide a broad account of molecular biology techniques employed to gather information about chromosomal organisation, with literature review of the current state of the art on computational methods for genomic data analysis.

[16] A glossary is present in the preface, and is intended to give a definition of most of the biology-related terms used throughout this thesis.

3.1 DNA exploration overview

Nuclear cell's organisation was traditionally studied by microscopy, bringing early knowledge about nuclear conformation, such as discovering large chromosomes at the nuclear periphery and small chromosomes located more interiorly, or revealing gene-poor and gene-rich regions within the chromosome territory. Fluorescence in situ hybridization (FISH) observations of DNA also significantly contributed in revealing valuable information on physical distances between genomic *loci*, and suggested a correlation between gene proximity and expression status [56]. However, FISH data and other microscopy methods are usually limited to single-cell analyses of gene positioning: on a genome-wide or cell-population scale they are limited in throughput and resolution.

Over the past ten years, advances in high-throughput molecular biology techniques have provided insights into chromatin interactions on a larger scale: in 2002 Dekker et al. developed a strategy called "Chromosome Conformation Capture" (**3C**) [46], a novel approach that puts the basis for a large number of methods that continuously improved the analysis of nuclear organization. The 3C method was the first to explore the organization of the chromosome in the cell, by detecting the frequencies of interaction between any two genomic loci in order to reveal their spatial disposition. New approaches followed, such as 4C, 5C, Hi-C and ChIA-PET: these methods utilize *next-generation sequencing* (**NGS**) techniques to interrogate the 3C ligation library more comprehensively [167, 49, 102] and with an increased throughput.

Most 3C-based techniques focus on the analysis of a set of predetermined loci, enabling the exploration of the conformation of the chromosomal regions of interest. On the other hand, Hi-C allows the characterization of long-range chromosomal interactions genome-wide [102]: it gives information about coupled DNA fragments that are cross-linked together due to spatial proximity, that would not otherwise be observable in shorter-ranges interactions, thus providing data about the chromosomal arrangement in the 3D space of the nucleus.

These techniques allow the description of the nucleus organization at unprecedented resolution: this is of critical importance for understanding and

evaluating the regulation of gene expression, DNA replication, repair, and recombination. Moreover, using the Hi-C approach, the possibility of comparing the three-dimensional organization of the DNA in physiological and pathological conditions is in hand. The capability of describing how diseases reorganize the chromatin conformation to originate novel co-localized gene clusters of co-expression would be of primary importance.

3.2 Next-Generation Sequencing

DNA stores biological information used in the development, functioning and reproduction of all known living organisms. DNA sequencing is thus fundamental for basic biological research and in numerous applied fields, from medical diagnosis to biotechnology: it permits to gain accurate knowledge of the nucleic acids composition (i.e. the building blocks of DNA) within a DNA molecule, and includes any method or technology employed to determine the order of the four bases — *Adenine*, *Guanine*, *Cytosine*, and *Thymine* — in a strand of DNA. The advent of novel rapid sequencing methods has greatly accelerated biological and medical research: genomes of numerous types and species — including the human genome and other complete DNA sequences of many animals and plants — have been discovered since the early 1970s [125].

In 1977 Frederick Sanger developed a DNA sequencing strategy based on the "chain-termination" method. Walter Gilbert developed another sequencing technology based on chemical modification of DNA and subsequent split at specific bases. Sanger's method used fewer toxic chemicals and lower amounts of radioactivity than Gilbert's one, thus becoming the gold standard for nucleic acid sequencing during the subsequent two and a half decades. It was adopted as the primary technology in the "first-generation" of laboratory and commercial sequencing applications. The first automatic sequencing machine based on Sanger's method was introduced only in 1987 by Applied Biosystems, with the name AB370 [103].

Since completion of the first human genome sequence in 2003[17] [156], the demand for cheaper and faster sequencing methods has greatly increased. This demand has driven the development of high-throughput sequencing strategies, also called *next-generation* sequencing (NGS) techniques and usually classified as second and third generation sequencing technologies. This classification could result quite ambiguous, because there is no consistent definition for each of these techniques and it is difficult to assign all different instruments to one or the other category [91]. *Second* generation techniques generally refer to those methods that apply a *Polymerase Chain Reaction* step (PCR) for signal intensification prior to sequencing, while *third* generation techniques mostly identify single molecule sequencing.

Commercially available high-throughput sequencing systems first appeared in 2005. Among them, the most used are the *Roche 454*, the *Illumina/Solexa HiSeq 2000* and the *AB SOLiD* systems. Of the three mentioned NGS sequencing systems, the Illumina HiSeq 2000 provided the biggest output and lowest reagent cost; the SOLiD system has the highest accuracy and the Roche 454 system has the longest read length [103].

Each of these approaches uses a different technology to accomplish nucleic acid sequencing, involving specialised control software and expensive computational infrastructures. However, the sequencing costs have dramatically reduced since the shift from Sanger-based to "second-generation" DNA sequencing technologies: the National Human Genome Research Institute (NHGRI) tracks costs associated with DNA sequencing performed at the sequencing centres [158], and shows that costs reduction, either in terms of "costs per genome" or in terms of "costs per Megabase of DNA Sequence", are profoundly outpacing Moore's Law (Figure 1.1).

With more and more organisms being sequenced, a flood of genetic data is becoming a reality. Consequently, each new generation of sequencers will require more, not simply faster, compute and storage support. For instance, the first computational effort is usually the alignment of sequenced reads against a

[17] As it is the result of a collaborative work of many researchers, we omit all author's names in the article's reference. The full authors list can be found at: `http://www.sciencemag.org/content/291/5507/1304`, together with the full article.

reference genome. This is useful to highlight genomic regions involved in expression and regulation. Alignments are computed using complex algorithms, normally employing the Burrow-Wheeler Transformation [30], which makes NGS data analysis a time-consuming, resource-demanding task that requires parallel, distributed and cloud infrastructures for conducting data analysis and interpretation.

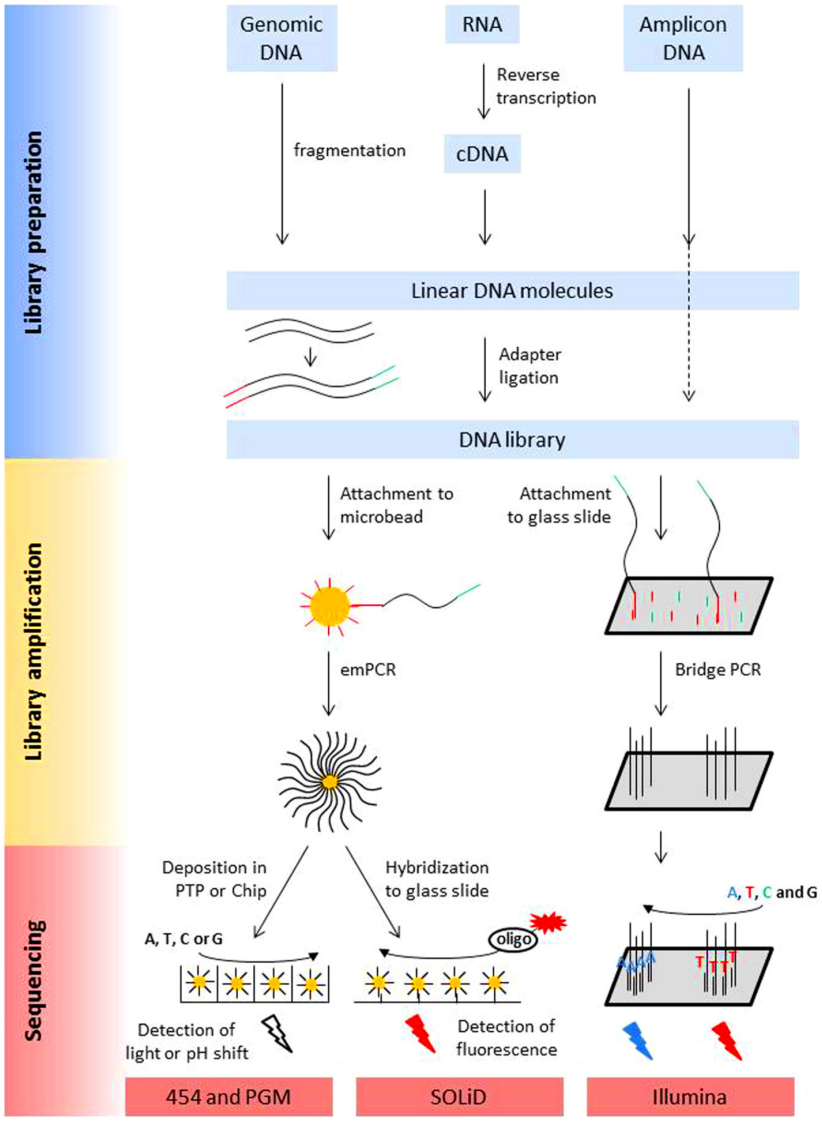


Figure 3.1: *A generic Next Generation Sequencing workflow, adapted from Knief C., 2014 [91]*

General methodology

NGS platforms can perform massively parallel sequencing, during which millions of fragments of DNA from a single sample are sequenced in a single experiment: we discuss here a general methodology used by second generation sequencing platforms. This choice is driven by the fact that the experiments analysed in this thesis come from data obtained using such "generation" of sequencers, which have different characteristics but also share several aspects of library preparation, library amplification and sequencing process (see Figure 3.1). A detailed description of next generation sequencing platforms and principles can be found in dedicated reviews [123, 103, 112] and references therein.

DNA sequencing methods differ from platform to platform in how sequencing is actually performed; they anyway rely on a common — or, to some extent similar — pattern of execution. The cutting-edge idea was to amplify each piece of genome in a local fashion, and automatically analyse the signal corresponding to each piece of genome in parallel, generating millions of sequencing reads. Operatively, the addition of one (or more) nucleotide(s) at each sequencing step results in a local reaction that generates a signal, recorded by the instrument. Some methods use light to mediate the signal, so a CCD camera is used for the identification of wells, while in some others a PH variation or an electronic variation is captured through a semiconductor technology.

An initial library of nucleic acids (DNA or complementary DNA (cDNA)) is first built by fragmenting the DNA (or cDNA) sample and ligating adapter sequences (synthetic oligonucleotides of a known sequence) onto the ends of the DNA fragments. Once constructed, libraries are *amplified* (i.e., replicated) using PCR or similar techniques, in order to increase signal intensity for the sequencing process. To obtain a nucleic acid sequence from the amplified libraries, sequencing must be applied. Sequencing is performed in a massively parallel manner for thousands to billions of library fragments, which act as a template off of which a new DNA fragment is synthesized. The sequencing occurs

through repeated cycles of nucleotide addition by a DNA polymerase or ligase (SOLiD), detection of incorporated nucleotides and washing steps. As nucleotides incorporate into the growing DNA strand, they are digitally recorded as sequence. Due to this iterative procedure, including extensive washing and flooding steps, sequencing lasts several hours to days.

Most sequencers allow sequencing of library fragments from both ends of a chromosome fragment: a corresponding reverse read can be assigned to each individual forward read. Paired-end reads can be used to improve sequence quality generated reads and can be done for library fragments of up to approximately 800 bp. Since the average size of the library molecules is known, the distance between forward and reverse read is also known. This gives the ability to map to a reference (or *denovo*) genome using that distance information. It helps to resolve larger structural rearrangements (insertions, deletions, inversions), as well as to assemble across repetitive regions.

Data analysis is a mandatory step to perform once sequencing is complete: raw sequenced data must be preprocessed to remove adapter sequences and low-quality reads. A mapping to a reference genome or a brand new alignment of the sequenced reads is desirable, together with an analysis of the compiled sequence using wide variety of Bioinformatics assessments, such as detection of novel genes or regulatory elements, and assessment of transcript expression levels [66].

3.2.1 RNA-Seq

RNA Sequencing (or RNA-Seq) uses any of a variety of next-generation sequencing techniques to reveal a snapshot of RNA presence and quantity in a genome. RNA-Seq permits to study the transcriptome of a cell (i.e. the set of all RNA molecules *transcribed* in one single cell or in a population of cells) and looks for genes mutations and changes in genes expression. Understanding the transcriptome is essential for interpreting the functional elements of the genome and revealing the molecular constituents of cells and tissues [115, 157].

Transcription is the first step of gene expression, in which a segment of DNA is copied into RNA by the enzyme RNA polymerase. Both RNA and DNA

are nucleic acids, which use base pairs of nucleotides as a complementary language. The two can be converted back and forth from DNA to RNA by the action of the correct enzymes. During transcription, a DNA sequence is read by an RNA polymerase, which produces a complementary RNA strand that can be translated into a protein during the translation process. The application of next-generation sequencing techniques allows for increased base coverage of a DNA sequence, as well as higher sample throughput. RNA-Seq can include total RNA, small RNA (such as miRNA and tRNA) and ribosomal profiling.

Prior to NGS, transcriptomics and gene expression studies were conducted with expression microarrays, which contain thousands of DNA sequences that potentially match complementary sequences in the sample, yielding a profile of all transcripts being expressed. However, the microarray technique presents a number of limitations, including reliance on existing knowledge about genome sequence, high background biases owing to cross-hybridization, and a limited dynamic range of detection owing to both background and saturation of signals. Moreover, comparing expression levels across different experiments is often difficult and can require complicated normalization methods [35].

3.2.2 ChIP-Seq

ChIP-Sequencing (or ChIP-Seq), combines Chromatin Immunoprecipitation techniques (ChIP) — used to determine whether specific proteins are associated with specific genomic regions — with massively parallel DNA sequencing, in order to identify those sites where a protein binds to the genome [111, 86]. Immunoprecipitation (IP) permits to produce a protein from a solution, using an antibody that specifically binds to that particular protein. This process can be used to isolate and concentrate a particular protein from a sample containing many thousands of different proteins.

ChIP-Seq is generally used to map global binding sites precisely for any protein of interest, in order to determine how chromatin-associated proteins influence the mechanisms that regulate the phenotype of a cell. ChIP-Seq first cross-links bound proteins to chromatin, fragments the chromatin, captures the DNA fragments bound to one protein using an antibody specific to it and

sequences the ends of the captured fragments using next-generation sequencing techniques [20]. Computational mapping of the sequenced DNA identifies the genomic locations where enzymes, modified histones, chaperones, nucleosomes, and transcription factors attach to the DNA, thereby illuminating the role of those proteins that interact with DNA within the gene expression process.

Determining how proteins interact with DNA to regulate gene expression is essential for fully understanding many biological processes and disease states. This epigenetic information is complementary to genotype and expression analysis. The use of NGS provides relatively high resolution, low noise, and high genomic coverage compared with ChIP-chip assays (ChIP followed by microarray hybridization).

3.3 Chromosome Conformation Capture

The "chromosome conformation capture" (**3C**) technique was developed by Dekker et al. more than 10 years ago [46] and is normally used to study the organization of short genomic regions at high resolution, compared to the resolution of most visual techniques.

To start the 3C procedure, a population of cells is first chemically fixed with formaldehyde to create covalent bonds between chromatin segments [84, 46]. Next, the fixed chromatin is *digested* with a restriction enzyme, which cuts the chromatin at specific sites across the genome and finds points where selected DNA regions are connected through a protein complex. The digesting enzyme determines the resolution of the 3C experiment: commonly used restriction enzymes (such as HindIII, BglII, SacI, etc.) recognize 4 to 6 base pairs (bp). The digested DNA fragments are re-ligated under diluted conditions, to promote intra-molecular ligations between cross-linked fragments pairwise. DNA fragments that result far away on the linear dimension (a one-dimensional (1D) cast of the 3D nuclear structure) but co-localize in space, can now be ligated to each other [45]. This step generates unique DNA junctions ready to be analysed by measuring the number of ligation events between non-neighbouring sites. This

can be done by several PCR amplification methods, that permit to quantify a targeted DNA molecule. By comparing the amplification efficiency of different primer combinations, a matrix of ligation frequencies is established, that serves as proxies for pairwise interaction frequencies. Figure 3.2 depicts principal steps in 3C-based techniques. The chromosome conformation capture methodology is showed on the left side, from top to bottom.

Chromosome conformation capture technique has become the foundation for a host of related techniques that have been developed to achieve greater scale, throughput or specificity: from the level of single locus (3C, 4C) to a set of loci (5C, ChIA-PET), and then genome-wide (Hi-C), it allows to link chromatin structure to gene regulation, DNA replication and chromatin folding.

4C

The chromosome conformation *capture-on-chip* (4C) techniques were the first set of methods designed to improve throughput and resolution of 3C. Each technique was developed independently, to enable the identification of previously unknown DNA regions that interact with one specific locus of interest, which makes 4C especially well suited to discover novel interactions with a specific region under investigation [143, 167].

The 4C procedure follows the same steps as in 3C, but adds ulterior operations before the quantification of the fragment of interest: once the 3C library has been obtained, the restriction fragments are subjected to another round of restriction digest (see Figure 3.2), this time with a frequent cutter (i.e., an enzyme that recognizes 4 base pairs) that will produce smaller DNA circles (possibly containing also 3C ligation junction). Self-circularization of the DNA fragments is then applied, because they are not bound to other proteins or fragments. Intra-molecular ligation occurs to induce the formation of the circular fragments, that will become the 4C library: at this point microarrays or high-throughput sequencing techniques can be used to sequence the 4C library [143]. When used in combination with next-generation sequencing techniques to analyse contacting sequencing, this method is called *4C-Seq*.

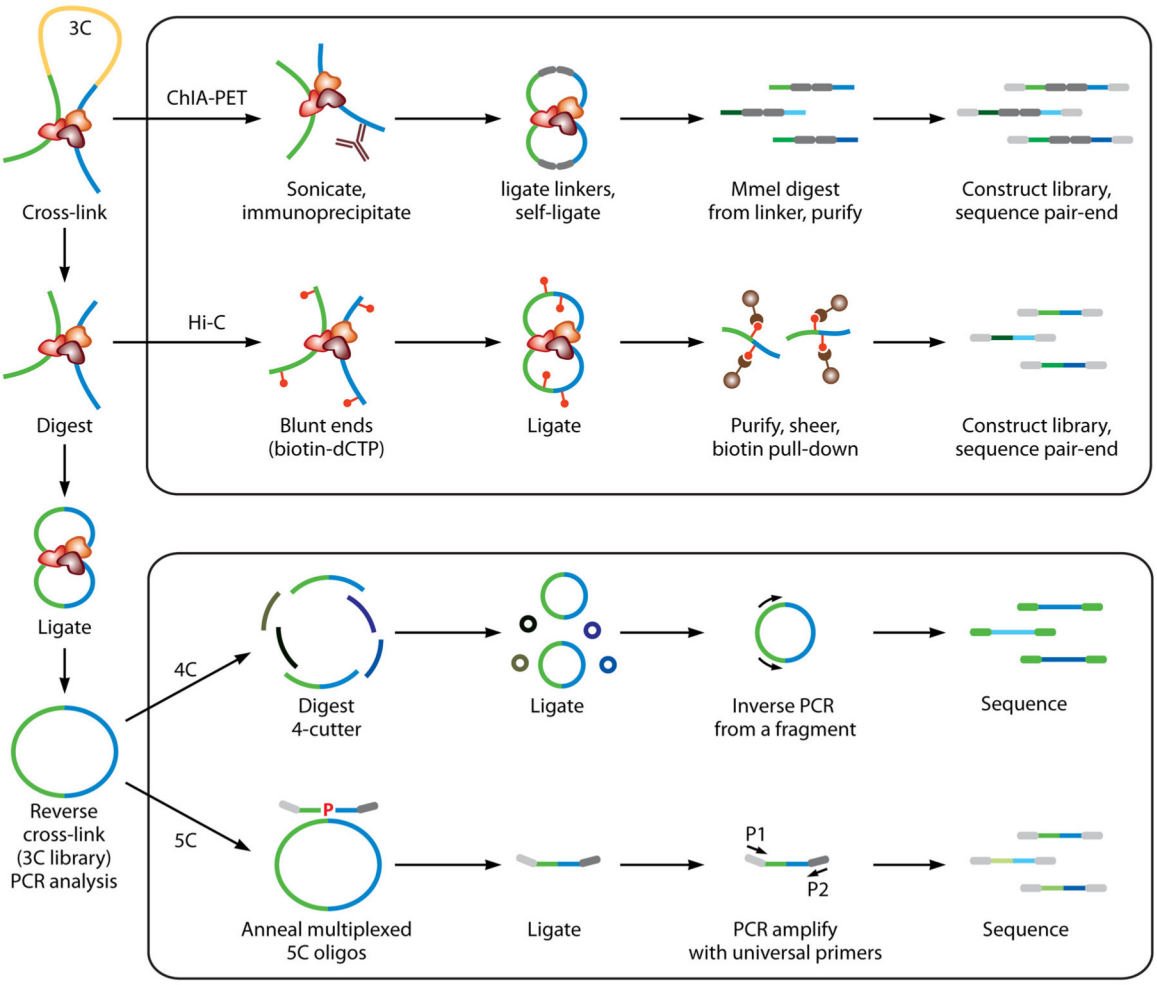


Figure 3.2: *A graphical description of 3C-based methods, adapted from Fraser et al., 2015 [59]. The phases that deploy the 3C method are depicted on the left, from top to bottom: chromatin interactions identified via the formaldehyde cross-linking are digested using a restriction enzyme and then ligated before reverse cross-linking and purification. The top panel shows that ChIA-PET and Hi-C methods relate to 3C: ChIA-PET starts its process from 3C's formaldehyde cross-linking phase, while Hi-C differentiates its methods after DNA fragments have been digested. The bottom window clearly explains that 4C and 5C operate over a 3C library by adding further experimental steps that lead to higher throughputs and better coverage.*

5C

The chromosome conformation capture *carbon copy* (5C) method is suitable to predict the conformation of entire domains or chromosomes. This technique generates a library of any ligation products from DNA regions that associate with the target loci, which are then analysed by NGS techniques [49]. 5C is ideal when great detail about all the interactions in a given region is needed, for example when seeking for a detailed interaction matrix of a particular chromosome. However, 5C is not truly genome-wide, since each 5C primer must be designed individually, that makes it best suited to particular regions.

After the construction of the 3C library, the resulting DNA sequence is amplified (i.e., thousands to millions of copies of that particular DNA sequence are generated). Pairs of DNA fragments that correspond to interacting fragments are juxtaposed on the 3C library and can be ligated together. Resulting ligation products are subsequently amplified simultaneously in a PCR reaction and sequenced using either microarray or high-throughput sequencing techniques (see Figure 3.2).

Hi-C

The traditional Hi-C method follows a standard procedure, whose initial steps are in common with 3C: it first cross-links cells with formaldehyde, resulting in covalent links between spatially adjacent DNA segments. DNA is then digested with a restriction enzyme which is filled with a biotinylated residue [102]. The resulting fragments are ligated under dilute conditions that favour ligation events between the cross-linked fragments. The ligated chromatin complexes get purified by pulling down biotin, making them ready for being analysed using massively parallel DNA sequencing. This whole process will produce a catalogue of interacting fragments, a genome-wide sequencing library that provides a valuable mean for measuring the three-dimensional distances among all possible locus pairs in the genome.

The resolution of a Hi-C dataset is determined by the restriction enzyme used. Normally, a Hi-C experiment uses a 6 bp cutting enzyme, which might however be undesirable and not deep enough to obtain a sufficient number of

reads and measure specific interactions of a small region. Instead, data can be binned into various fixed-size genomic intervals, in order to aggregate data and smooth out noise: data is typically binned into sizes ranging from 40 kb to 1 MB and bin-bin interactions are aggregated by taking the sum of the interactions [94].

The Hi-C technique has recently been extended to create a genome-wide catalogue of chromatin loops at the unprecedented scale of 1 kb [128]: by applying an *in situ* Hi-C protocol in which DNA-DNA proximity ligation is performed in intact nuclei, the chromosome maps obtained comprise over 5 Tb of sequence data, recording over 15 billion distinct contacts, which is an order of magnitude larger than all published Hi-C data sets combined together.

ChIA-PET

Hi-C does not provide full information about the nature or the function of chromatin interactions. The ChIA-PET method (chromatin interactions analysis by paired-end tag) was developed to extract these information by mapping chromatin networks associated with specific proteins, to determine *de novo* long-range chromatin interactions genome-wide [60]. ChIA-PET uses a chromatin immunoprecipitation (ChIP) step to isolate chromatin interactions identified via the formaldehyde cross-linking (i.e., step 1 of 3C method, see Figure 3.2). The fixed cells are sonicated to break-up the chromatin and reduce non-specific interactions. Then using a specific antibody the protein of interest is enriched. DNA linkers are next used to connect proximity ligated DNA fragments, and the resulting cross-linked DNA fragments are ligated together intramolecularly. The purified DNA is digested using MmeI enzyme and the ChIA-PET junctions generated by this process are then cut out with restriction sites featured in the linkers, prior to purification on streptavidin beads and paired-end sequencing. Each half of the ChIA-PET products is finally mapped to a reference genome and joined to reveal the location of protein-mediated chromatin contacts [59].

3.3.1 Normalisation

DNA sequencing readouts may contain either very noisy or too low signals, that could lead to false-positives or false-negatives, causing incorrect data interpretation. As this thesis will focus on Hi-C data analysis and interpretation, we will discuss some potential sources of biases and methods to reduce these inconveniences.

A common way of representing Hi-C data is a matrix-based representation. A *contact map* is a matrix that represents non-overlapping "bins" across the genome (Figure 3.3). Each entry in the matrix contains a count of read pairs that connect the corresponding bin pair in a Hi-C experiment. In other words, each entry in a contact map tells the intensity of the interactions between two chromosomes or two locus. These bins can be either fixed-size windows or can correspond to a fixed number of consecutive restriction fragments, and the size of bins should be a trade-off between resolution and statistical power. Once the bin size is determined, the resulting raw contact map serves as the input for the normalization methods. However, several factors may introduce bias and noise to these measurements: first, the sequence composition of the fragments can affect the sequencing step. For example, a higher *GC* content may be associated with a higher sequencing coverage; the ability to uniquely map a fragment back to the genome (its *mappability*) may vary, for example due to repeats or to the fragment length. Consequently, some fragments may be over or under represented or wrongly identified. Since different restriction enzymes cut the genome at different cleavage sites, the fragmentation produced will be different as well. As a result, contact maps that were generated with distinct restriction enzymes, may provide a different view for the same region. These confounding effects ultimately mean that the observed contact frequency between two bins may provide a biased estimation of their "true" contact frequency. Considering also the 3C protocol, some contact frequencies may not provide meaningful spatial information, even after correction.

Normalisation and correction methods for Hi-C data have been developed to remove these biases. Some of them can be employed when there is *a priori* knowledge of the factors that may cause bias in the data: these methods are

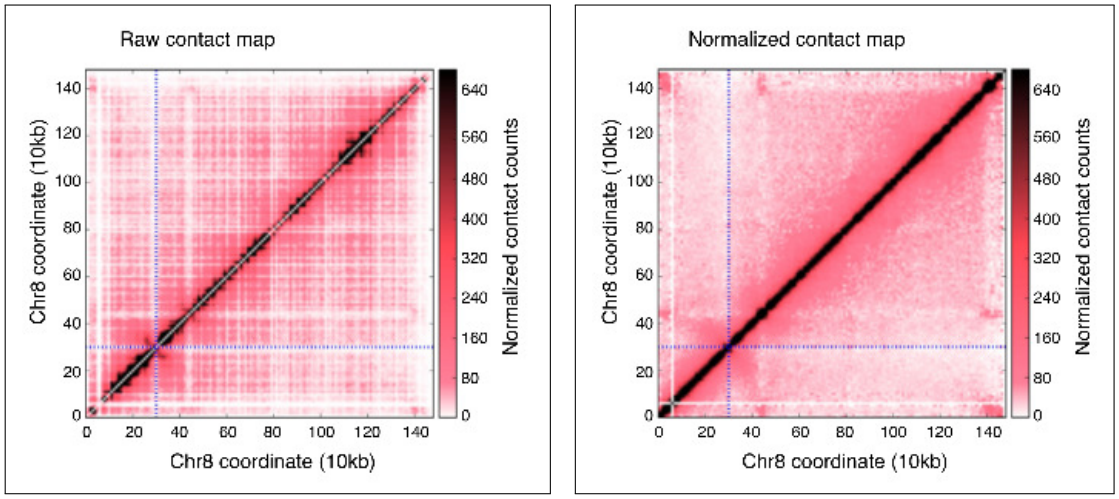


Figure 3.3: *Hi-C contact maps of chromosome 8 at 10 kb resolution before and after normalization, adapted from [18]*

called *explicit-factor* corrections. Other approaches try to correct all factors that may cause biases, without explicitly modelling them: these methods are also known as *matrix balancing* corrections.

Yaffe and Tanay were the first to discover such factors [164] and developed an explicit correction procedure that models the probability of observing a contact between two regions, but requires *a priori* knowledge of the genomic features that affect contact counts (i.e., GC-content, mappability, and fragment length). Hu et al. improved such method providing a significantly faster explicit correction technique that uses regression-based models [76].

Matrix balancing methods are conveniently defined *implicit*, and rely on the assumption that if there were no bias, roughly an equal number of Hi-C reads should originate from each region of equal size (in terms of base pairs) in the genome, and each locus in the genome would give rise to an equal number of reads in a Hi-C experiment. This approach applies a matrix balancing method that finds a decomposition of the observed contact map, and updates the interaction information according to the number of valid interactions the associated fragment is involved in.

Both implicit and explicit techniques are valuable means to remove biases

from 3C-based experiments, and have proved to be highly accurate. These studies have shown that normalization is essential for Hi-C data, because normalized contact maps are visually smoother than their raw versions, facilitating the identification of potentially interesting contact patterns. On the other hand, a significant computational effort is required to perform these corrections, and this is particularly true for those techniques based on matrix balancing algorithms: with high-resolution data that reach the kilobase level or beyond, Hi-C interaction matrices are of massive size ($O(N^2)$, where N is the number of genomic regions involved). Normalisation requires expensive computing resources, such as large memory and long computation time. Most current implementations of the normalization methods discussed above cannot directly handle high-resolution human Hi-C data below 10 or 50 kb. Moreover, when normalizing Hi-C data using contact maps the sequence information is blurred and not preserved for subsequent mapping of genomic features.

3.4 Genomic spatial data analysis - State of the art

Sequence aligners map reads against a reference genome, in order to study the structure and functionalities of sequenced data[18]. The most common and most used fast sequence aligners are based on the Burrows-Wheeler transform (BWT) algorithm [30], that is essentially a string permutation algorithm also applied in data compression tools.

Bowtie [97] is probably the most famous, BWT-based sequence aligner, designed for aligning short DNA sequence reads with small memory footprint and possibility to exploit multi-threading. *Tophat* [151] is a widely used aligner that performs spliced alignments and works directly on reads from transcriptomic experiments, using the reference genome indexed according to the Burrow-Wheeler Transformation.

HiCUP [19] is the best known and most used tool for Hi-C data processing: it mostly focuses on the initial steps of the Hi-C procedure, such as mapping and filtering. However, it does not provide a normalisation step, but produces

[18]See "mapping" entry in the Glossary

summary statistics and quality control at each stage of the pipeline, helping to identify potential problems and to refine the experimental protocol. HiCUP outputs paired-end reads in SAM (or BAM) format, that may be used by Hi-C analysis tools to extrapolate knowledge.

Considering general software for the interpretation of Hi-C data, an interesting package is *HOMER* [136], which contains several programs and routines to facilitate the analysis of Hi-C data. Like most of the available applications, HOMER relies on the creation of contact maps for the interpretation of Hi-C data, exploiting Principal Component Analysis and hierarchical clustering with this representation. Several of the HOMER programs support multiple processors to help speed up the computation, although, at the time of writing this thesis it only works at the chromosome level.

HiTC [138] has been designed to facilitate the exploration of high-throughput, 3C-based data. It allows users to transform, normalize and visualize contact maps whose entries correspond to the number of times two restriction fragments in a given genomic region (bin) have been ligated together during the 3C procedure and sequenced as a pair. The HiTC package proposes a list of options to define the appropriate data visualization, such as contrast, color or counts trimming.

Fit-Hi-C [17] assigns statistical confidence estimates to mid-range, *cis* contacts (i.e., intra-chromosomal) by jointly modelling the random polymer looping effect and previously observed technical biases in Hi-C data sets.

If FISH experiments are available, a good normalization solution is represented by *FisHiCal* [139]. This is an R package that performs an iterative *FISH*-based Hi-C calibration that exploits the information coming from both these methods. It is the first tool that integrates FISH and Hi-C data, and operates over these information to calibrate the direct measure for physical distance provided by FISH experiments and the genome-wide capture of chromatin contacts obtained by Hi-C experiments.

Yaffe and Tanay [164] proposed a probabilistic model to correct biases based on the observation of the genomic features. This approach can remove the majority of systematic biases, at the expense of very high computational costs, due to the observation of paired-end reads spanning all possible fragment end pairs.

Tool	Aligner	Normalisation	Visualisation	Implementation Language
HiCUP	Bowtie / Bowtie2	-	✓	Perl, R
HiCLib	Bowtie2	Matrix balancing	✓	Python
Homer	-	simpleNorm / norm	✓	Perl, R, Java
HiTC	-	normLGF / normICE	✓	R
Fit-Hi-C	-	-	✓	Python
Fish-Hi-Cal	-	-	✓	R

Table 3.1: *Software tools for Hi-C data analysis*

Hu et al. proposed a tool named HiCNorm [76] that uses a parametric model based on a Poisson regression to correct technical and experimental biases from Hi-C readouts. This is a simplified, and less computationally intensive normalisation procedure than the one described by Yaffe and Tanay, since it corrects the systematic biases in Hi-C contact maps at the desired resolution level, instead of modelling Hi-C data at the fragment end level. The drawback here is that the sequence information is blurred within the contact map.

Concerning data visualisation, some genome browsers [78, 87, 118] are used to visualise thousands of data tracks for human, mouse and other organisms. These browsers are mainly designed for visualisation of one-dimensional signals and are not easily extensible to visualising two-dimensional Hi-C or any conformation capture data. Furthermore, Hi-C data provide a 3-D model of the chromosome, which requires tools not only for two-dimensional, but also for three-dimensional visualisation. Genome3D [16] is an interesting effort towards this achievement.

3.5 Discussion

3C-based methods have a high capture probability between neighbouring fragments, conforming to their close spatial proximity. Drifting apart from a given fragment leads to exponential decrease of capture probability, until it reaches a

baseline level [102, 45]. This means that junctions between two given sites located far away on the chromosome, or on different chromosomes, will be rare. For this reason, the 3C strategy is unsuitable for the analysis of long-range chromatin contacts. 4C is currently limited to the description of interactions between a single restriction fragment and the rest of the genome, but it cannot be used to predict the conformation of larger regions elsewhere on the chromosome (*cis*) or on other chromosomes (*trans*). Local interactions are not yet readily picked up due to a lack of resolution [45]. 5C was designed to increase the throughput and accuracy of 3C and allows the concurrent determination of interactions between many selected domains.

*N*C methods give a population-based measure that relies on spatial proximity but does not directly convey a spatial context. Hi-C data add a spatial context to biological inquires and facilitate the discovery of gene regulation factors. On the other hand, it cannot measure the dynamics of interactions between multiple genomic loci, like fluorescent microscopy would do. However, it does provide the ultimate connectivity between the genomic sequence and the spatial conformation. ChIA-PET reduces the complexity for genome-wide analysis and is applicable to many different protein factors involved in transcriptional regulation or chromatin structural conformation, and is dependent on the identification of sequences that can be mapped to the reference genome [45].

The Hi-C method has been the first to demonstrate that the genome is partitioned into numerous domains, even though the 3D architecture of the nucleus was inferred since a late Eighties' seminal study by Mukherjee et al. [113], where chromatin looping in the presence of ligase was first detected. Subsequent analysis suggested that chromosomes are organized in hierarchical length scales. From low to high resolution, chromosomes first fold to occupy distinct territories and positions in the nuclear space defined in part by interactions with nuclear subdomains. Individual chromosomes are then folded into compartments *A* (open/active) and *B* (closed/silent) that preferentially interact together, respectively. Within compartments, the chromatin is characterised by the presence of smaller domains, which are in turn partitioned into condensed structures ~ 1 Mb in size, called "topologically associated domains" (TADs),

largely conserved between cell types and across species. The chromatin is further folded into sub-TADs, the topologies of which can vary in a tissue-specific manner. TADs are identified as densely interacting squares on the diagonal of the *contact map* (see Figure 3.3).

In this thesis we investigate a novel approach that permit to render a picture of the genome organisation and subsequently study the interactions that occur among genomic entities, which are likely to play a part in many functional and regulatory processes. To achieve this purpose, we will start focusing on Hi-C data processing, because it provides a three-dimensional snapshot of the chromosome by measuring the number of ligation events between non-neighbouring sites. Upon this 3D snapshot, we integrate heterogeneous one-dimensional data coming from different experiments (e.g., ChIP-Seq and RNA-Seq), that will enrich the information about genome spatial organisation with additional functional elements.

3.5.1 Visualisation of biological data

Visualisation of genomic data is crucial for both hypothesis generation and detection of potential artefacts. Molecular interaction data that are stored in a variety of databases are the cornerstone of many computational approaches aiming to analyse, model, interpret and predict biological phenomena. Interactions are often thought of as constituting networks, which are used to model molecular interactions as they were large systems of interacting particles [81, 132].

Despite a small number of interactions might be difficult to understand through a graph-based representation[19], statistical properties relating to all interactions could contain valuable information: graphs are convenient representations of molecular interaction networks, and could be treated similarly to large systems of interacting particles. From these premises emerged *network biology* [21], a combination of systems biology, graph theory and computational and statistical analyses in which the topology of the graphs representing molecular interaction networks themselves became the subject of study. Over the past decade,

[19]While *network* and *graph* are two distinct, though related, concepts, they will be used interchangeably throughout this work.

network biology has changed from being a descriptive approach, to a predictive tool used to discover biologically relevant facts.

It is worth recalling that graphs are structures representing relationships between pairs of objects. A graph $\mathcal{G} = (V, E)$ is composed by a set V of *nodes* (or *vertices*) and a set E of node pairs called *edges* (or *links*) that describe the relationships between pairs of nodes. When two nodes u and v ($u, v \in V$) are linked (i.e. $(u, v) \in E$), u is said to be a *neighbour* of v, and vice-versa. A neighbourhood graph for a node u is thus the set of all nodes adjacent to u, included u. In *directed* graphs, used for modelling non-symmetric relationships, each link is directed and has a source node (*origin*) and a target node (*destination*). The number of neighbours of a node is called its *degree*. Weighted graphs normally describe non-binary relations by associating scalars (or weights) with links. An induced subgraph $G' \subseteq G$ is a subset of the nodes of G, along with all links whose endpoint nodes are both in G .

With a graph-based representation it is possible to apply network analysis over the resulting graph and study graph metrics to reduce structural properties of a network to real numbers, facilitating the comparison of different networks. Topological measures (such as node degree, and path metrics) capture graph's structure for nodes and edges and highlight the "importance" of the actors. Centrality metrics describe the interactions that (may) occur among local entities while ranking of nodes by topological features (such as degree distribution) can help to prioritize targets of further studies or lead to a more local, in-deep analysis of specific chromosome locations. Here studies of functional similarity can suggest new testable hypotheses [162].

3.5.2 Research niche

To the best of our knowledge, few tools are able to compare, visually or statistically, two Hi-C contact maps [136, 140], and none of these tools allow the joint analysis of more than two datasets coming from multiple time points, conditions, or cell types. Also, many of the existing methods do not scale to high-resolution Hi-C data from large genomes, such as human and mouse. Moreover, the integration of two-dimensional Hi-C data or three-dimensional chromatin

models with the vast quantity of available one-dimensional datasets — such as replication timing, histone modifications, protein binding and gene expression — is still rather understudied.

Table 3.1 lists some common Hi-C tools for Hi-C data analysis. The solution that emerges from our work can be clearly compared to the above, but proposes a different point of view concerning chromosome structures and interactions: by leveraging the graph-based representation, our solution permits to analyse genomic processes with a "social network" point of view, focussing on the features that lead to tie formations and that likely foster interactions among elements. Furthermore, most of the tools listed in Table 3.1 do not seem to account for performance concerns, mostly because these tools are logically composed of several different computational stages (such as pipelines), where each stage has different computing requirements. Notably, those tools that encompass an alignment phase are able to exploit some multi-processing capabilities during this phase.

The first NuChart prototype [107] applied Hu et al. solution to normalise sequencing biases and estimate a score for each read, identifying half of the Hi-C contact instead of normalizing the contact map. In this thesis we leverage this solution, proposing an *ex-post* normalisation that is used to estimate a probability of physical proximity between two genes, expressed as a score assigned to an edge connecting two nodes in the neighbourhood graph. In this way, the sequence information contained within the read that determined an interaction between two genes, is preserved, while the simple contact map representation would hide this information behind the contact frequencies.

Concerning visualisation, all the tools listed in Table 3.1 provide visualisation means to support processing results: normally, Hi-C data is depicted through heat maps where highly interacting regions are visible at first glance (see Figure 3.3). In our work, the graph-based visualisation in fostered: differently from the previous, a graph clearly depicts the actors of the complex system in use (e.g., genes), and permit to annotate additional features on the graph, or to customise some visualisation features in order to render a better picture of the analysed system.

Network visualisation has its drawbacks: *in primis*, common visualisation

engines work fine with small-to-medium size networks, but become hard to read when graph dimensions increase (at least, on common visualisation media). We will discuss our efforts for an exhaustive representation of the results we obtain, while in Chapter 7 we discuss open issues and future works, including the visualisation problem.

Chapter 4

Scalable chromosome exploration: introducing NuChart-II

In this chapter we describe our approach for developing a fast and scalable tool aimed at facilitating the analysis of chromosome conformation using a gene-centric, graph-based representation of genomic data. Here we propose NuChart-II, a software that allows to visualize and integrate genomic features involved in the chromosome spatial organization, built on top of FastFlow, a C++ framework for structured parallel programming.

4.1 Three-dimensional chromosome exploration

A huge amount of information is daily produced in molecular biology laboratories all around the world, but the representation and interpretation of this data in an effective way is a complex and challenging task. Specifically, sequencing results from expression profiles, methylation patterns and chromatin domains are difficult to describe in a systemic view. Also, an increasing number of experiments highlight the importance of studying the spatial organisation of the DNA in the nucleus, in order to gather insight on the processes ongoing within a cell: there is an undeniable need for a software that permits the integration and the interpretation of genomic data on a nuclear map, capable of representing the effective disposition of genes in the three-dimensional (3D) space.

Over the last decade, a series of molecular and genomic approaches have been developed to study three-dimensional chromosome folding at increasing resolution and throughput [45]; these methods are all based on the Chromosome Conformation Capture (3C) technique and allow the determination of the frequency with which any pair of *loci* in the genome is in close enough physical proximity (probably in the range of 10-100 nm) to become cross-linked [46]. A 3D view of chromosomes might reveal that widely separated functional elements actually result to be in close spatial proximity, and their interaction can be the key for detecting critical epigenetics patterns and chromosome translocations involved in the process of genes regulation and expression.

Among 3C-based techniques, the *Hi-C* method exploits next-generation sequencing (NGS) techniques to investigate genomic loci that physically interact in the nucleus [102]. Hi-C gives information about coupled DNA fragments that are cross-linked together (during the formaldehyde fixation step of the experimental protocol) due to spatial proximity, providing data about the chromosomal arrangement in the 3D space of the nucleus.

The output of a Hi-C process is a list of pairs of locations along the chromosome, which can be represented as a square matrix Y, where $Y_{i,j}$ stands for the sum of read pairs matching in position i and position j, respectively. This matrix-based representation, called *contact map*, gives the contact frequencies between a group (or groups) of genomic bins. The contact frequency between two bins relies on their spatial proximity and thus it is expected to reflect their distance. A contact map is reliable while looking at the intensity of the interactions between two chromosomes, but becomes unsuitable to depict the neighbourhood of a gene (or of a cluster of genes), lacking a possible emphasis on which *actors* could play a significant role in the gene regulation process.

On the other hand, a graph-based representation of Hi-C data can be very useful to create a map where other *omics* data can be mapped, in order to characterize different spatially-associated domains: a graph has a high level of expressiveness, insofar as nodes represent the actors of a process while edges identify relationships among the actors. Structural properties of a graph can reveal significant information on how the actors of the represented process interact, while parallel algorithms can be employed to operate over a graph.

The construction of such graphs is based on the exploration of static datasets: raw data resulting from Hi-C experiments are processed through the HiCUP pipeline [19], which produces millions of paired-end *reads* (i.e., short DNA sequences with start/end coordinates) listed in a `.sam` file. These reads are evaluated against a reference genome and an organism's list of genes. Static data structures are constructed from these datasets and are needed throughout the stages that characterize the genomic data analysis. However, using full data structures dramatically increases the size of used memory and induces an artificial memory-bound nature that can be avoided. For instance, not all information is needed at every phase of data exploration and data analysis: by reducing or optimising the working set (that is, the collection of information referenced by a process during the execution) and by applying memory optimisations, it is possible to substantially improve the overall performance.

Although several solutions for Hi-C data processing exist, most of them poorly exploit computing capabilities and optimised memory access in modern shared-memory architectures. Moreover, the majority of them propose contact maps for the analysis of the chromosome structure (see Section 3.4), but this approach flattens the possibilities for data interpretation uncovered by Hi-C experiments to a mere frequency count, while the genomic information — such as the DNA sequence, mapping quality score, fragment length, gene expression, etc. — is blurred beneath. We propose an approach that overcomes these limitations: based on an early R prototype [107] we designed NuChart-II, a C++ application that uses advanced parallel computing techniques (such as lock-free synchronisation and algorithmic skeletons) and applies memory optimisations to provide a (*gene-centric*), graph-based representation of the chromosome organisation.

The first prototype [107] entirely relied on the R environment, and presented evident limitations in performance and scalability: its overhead in managing large data structures and its weaknesses in exploiting the full computational power of multi-core platforms made the first NuChart prototype unfit to scale up to larger data sets and highly precise data analysis (which requires many iterations of graph building process).

NuChart-II has been designed according to a structured parallel programming approach [15, 4]; in particular, it has been designed on top of the *FastFlow* parallel programming framework [8], that provides high-level parallel programming patterns for the C++ language[20].

Contributions

The whole application is characterized by four main phases: *1)* data retrieval from static datasets; *2)* construction of the graph; *3)* weighing of the edges as a result of the *normalisation* step; *4)* output and statistics. Phases two and three are suitable for being rewritten in terms of loop parallelism, since their kernels can be run concurrently on multiple processors with no data-dependencies involved: the construction of the graph and the weighing of the edges. These two phases constitute by far the most onerous parts of the application in terms of execution time: particularly when the diameter of the graph increases, these phases take up the 80% of the whole execution time.

In this chapter we will discuss how these phases have been deployed and which results we obtained from each step: our efforts have focused either on the parallelisation of the application for shared-memory multi-core architecture, or on the design of a fast and reliable tool for Hi-C data analysis. A comprehensive list of available tools for Hi-C data analysis has been presented in Section 3.4, together with a discussion on normalisation techniques needed to correct systematic biases that may arise from sequencing and mapping: we refer to those pages for a background on *omic* tools for Hi-C data analysis. An up-to-date list of such applications is also available from the *Omictools* website[21]: some of them focus more on the initial steps of chromosome conformation studies, such as mapping and filtering, whereas others focus on normalization, visualization, and statistical confidence estimation. In our view of a usable mean for Hi-C data analysis, visualization and statistics assume a dramatic importance. To the best of our knowledge, no other tool proposes a gene-centric, graph-based visualization of the neighbourhood of a gene, as NuChart-II does.

[20] see Section 2.4

[21] `http://omictools.com/3c-4c-5c-hi-c-chia-pet-c298-p1.html`

With respect to the normalisation step, the first NuChart prototype [107] proposed a modification of Hu et al. solution [76] and attempted to estimate a score to each read, identifying half of the Hi-C contact instead of normalizing the contact map. In this way, the sequence information contained in the read is preserved and not blurred within the contact map's numbers. With NuChart-II we leverage this solution and propose an *ex-post* normalisation that is used to estimate a probability of physical proximity between two genes, expressed as a score assigned to an edge connecting two nodes in the neighbourhood graph (see Section 4.3).

Concerning the re-engineering of the application, the FastFlow library has been our first choice among available algorithmic skeleton frameworks (see Section 2.4.3): despite many others provide support for share-memory multi-core architectures — which is the type of architecture we have been using for this work — we were already acquainted with FastFlow, and largely enjoyed the possibility to seamlessly parallelise `for` loops by means of lambda functions. We also evaluated our solution against TBB- and OpenMP-based implementations, so that we could have a comparison with parallel programming frameworks created by major industry powerhouses, as opposed to other algorithmic skeleton frameworks developed by academic institutions.

As a matter of fact, performance achieved with the two alternatives we evaluated reached comparable results to those obtained with FastFlow. Main differences where found mostly in terms of ease of coding: OpenMP permits to easily achieve the desired result, by simply using the pre-compiler directives that permit to exploit the loop parallelism. On the other hand, Intel TBB implementation was not as much straightforward: the `parallel_for` pattern needs more tuning and fixes with respect to FastFlow's one, and thread-local storage is not as easy to use.

4.2 Neighbourhood graph construction

We recall that a graph G is a formal mathematical representation of a collection of vertices (V), connected by edges (E) that model a relationship among

vertices. In this context, vertices represent Genes (e.g., an ordered set of an organism's genes) labelled with genes names. Here we define a paired-ends Hi-C read as a connection c, $(c \in C)$, meaning a spatial relationship between two genes. Follows that two genes $g_1, g_2 \in V$ are *connected* if there exists a connection $c(g_1, g_2) \in C$ encompassing both of them. If such a connection exists, then exists an edge $e = c(g_1, g_2) \in E$.

The neighbourhood graph $N_G,\ N_G \subseteq G$, can be defined as undirected weighted graph $N_G(V_N, E_N, w)$ where:

- $V_N \subseteq V$ is a set of Genes;
- $E_N \subseteq E$ is a set of existing Edges;
- $w : E \to \mathbb{R}^+,\ 0 \leq w \leq 1$, is a function that assigns a probability of actual physical proximity for each pair of adjacent genes $c(g_i, g_j)$ connected by means of a paired-ends Hi-C read.

The neighbourhood graph is thus the induced *subgraph* obtainable starting from a given root vertex v, and including all vertices adjacent to v and all edges connecting such vertices, including the root vertex (see Figure 4.1). With these premises, our neighbourhood graph represents a topological map of the specific nucleus region to which a gene belongs.

Graph construction

A typical Hi-C analysis begins with the pre-processing of FASTQ files with HiCUP, which produces a SAM file containing millions of paired-end reads. These reads represent the main input of NuChart-II, because they expose the spatial information exploited by the process to infer a topological structure of the DNA.

By refining the algorithm proposed in the original prototype [107], NuChart-II evaluates reads against a reference genome that contains the coordinated of chromosome fragments generated by a digesting enzyme, and a list of genes with their positions (again, coordinates) along the DNA. The basic mechanism in the exploration stage loops over all wanted genes: for each gene, it looks for

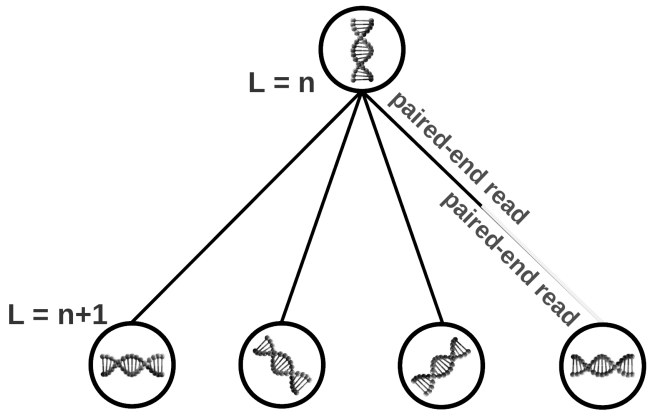


Figure 4.1: *Neighbours of a gene in our graph representation. An edge between two genes exists if they are connected, that is to say, if a paired-end Hi-C read that encompasses two genes exists, then exists an edge that connects the two genes.*

all those paired-ends Hi-C reads (connections $c_i \in C$, in our case) whose first end encompasses the current gene — basically comparing chromosome fragment and gene coordinates. Among found connections, it searches for neighbouring genes that might be located within c's second end. The reason for searching adjacent genes in a read's second end come from the way Hi-C (and 3C-based) experiments are conducted: Hi-C identifies spatially adjacent DNA segments — in terms of three-dimensional space. If a gene is found on a read's first end, a possible gene found in the second end is likely to be spatially adjacent, unless of sequencing errors and biases.

If we define the root of our neighbourhood graph to be at *level* 0, a search at level 1 yields all the genes directly adjacent to the root. Follows that a search at level i returns all genes directly adjacent to any gene discovered at level $i-1$, starting from the root. The final graph is returned in form of a list of edges.

Listing 4.1 reports a pseudo-code for the (sequential) graph construction. Each iteration of the outermost `while` loop pops an item q from the queue of unvisited genes and explores the reads dataset to find those reads whose first end encompasses q (Listing 4.1, row 10). For each discovered read, the algorithm searches for a gene on the second pair of the active read (Listing 4.1, row 13): if a candidate gene g is found, an edge (q, g) is possibly added to the edges list E (unless it already exists), and g is pushed into the working queue Q, as

```
1 BuildGraph (roots, Reads, Genes) {
2    Q = V = E := ∅
3    G := ∅
4    lv := 0
5
6    push roots in Q
7    while (Q not ∅) {
8       pop q from Q
9       for_each (c in Reads, c.FirstEnd.Chr == q.Chr) {
10         if (q overlaps c.FirstEnd) {
11            // find neighbour genes for q
12            for_each (g in Genes, g.Chr == c.SecondPair.Chr) {
13               if (g overlaps c.SecondPair and (q,g) not in E) {
14                  add (q,g) to E
15                  if(not g.Visited) {
16                     add g to V
17                     push g in Q
18                  }
19               }
20            }
21         }
22      }
23      lv := lv + 1
24   }
25   G := (V, E)
26 }
```

Listing 4.1: *Sequential graph construction*

in any typical graph exploration procedure, unless it has already been visited. When the queue Q is empty, all genes reachable through the given set of reads have been found: the procedure terminates returning the graph $\mathcal{G}$ as a list of edges. Note that each edge represents a connection between two genes: for each edge of the graph, the DNA information contained in the Hi-C read is still available, including mapping quality score, DNA sequence and fragment coordinates. We also keep count of the level of each found gene, intended as its "distance" from the starting node: the variable lv keeps track of this information.

At each iteration, a subset of the Hi-C reads file is accessed — namely, those reads whose first end falls in the same chromosome as the one enclosing the gene q (Listing 4.1, row 9). Then, for each read c, a subset of the genes dataset

is accessed — namely, the genes enclosed in the same chromosome as the one enclosing the second end of the connection c (Listing 4.1, row 12).

4.2.1 Data-parallel BFS-like graph exploration

The graph exploration proceeds according to a *Breadth First Search* (BFS) strategy: starting from one or more root genes (the starting node(s) of the graph), it expands the discovered graph one level at a time, until either all the reachable nodes have been found (i.e. fix-point) or up to a chosen distance from the root. The *BFS-like* graph exploration results in a data-parallel procedure, in which any arbitrary subset of reads can be processed independently from each other, provided that no data dependency is involved in their manipulation. Ideally, it can be parallelised in a seamless way by just taking the kernel of the procedure and putting it into a `ParallelFor` loop pattern (see Section 2.4.3). High-level parallelisation of graph exploration has been treated, among others, in [75].

This high-level approach requires some adjustments. For instance, the BFS-like graph exploration should be organised in a level-synchronised way, and concurrent write accesses to data structures shared between worker threads must be managed. For example, each iteration of the loop should build a local graph, and some mechanism of graph merging from local graphs to a global output graph (actually one for each level) should be provided. Globally, this approach amounts to provide a *reduce* phase after each `ParallelFor` instance, in which per-thread local structures are merged into per-level global ones.

4.2.2 Memory-optimised graph construction

User-defined data structures used for describing complex data often gather several (related) elements in a single data type. This logical organisation also reflects how these elements will be mapped in physical memory and this — ideally — should not affect the data access performance. However, current architectures are highly optimised for contiguous memory access, thus extra care should be taken when dealing with arrays of complex user-defined data structures.

```
1 BuildGraph (roots, Reads, Genes, L_MAX, NTH) {
2    Q = Γ = G := ∅
3    C[NTH] = V[NTH] = E[NTH] := ∅
4    lv := 0
5
6    push roots in Q
7    while (Q not ∅ and lv < L_MAX) {
8       pop q from Q
9       // find Hi-C Reads for q
10      ParallelFor (c in Reads, NTH) {
11         if (q overlaps c.FirstPair and q.Chr == c.Chr)
12            add c to C[th]
13      }
14      // find neighbour genes for q
15      ParallelFor (c in C[th], NTH) {
16         for_each (g in Genes, g.Chr == c.SecondPair.Chr) {
17            if (g overlaps c.SecondPair)
18               add g to V[th]
19               add (q, g) to E[th]
20         }
21      }
22      // level synchronisation
23      Γ := BuildPartialGraph(V[th],E[th])
24
25      for_each (v in V[th], 0 ≤th<NTH) { // next level vertices
26         if (not v.Visited)
27            push v in Q
28      }
29      lv := lv + 1
30      C[th] = V[th] = E[th] := ∅
31   }
32   G := BuildGraph(Γ)
33 }
```

Listing 4.2: *Parallel BFS Graph Construction*

The basic BFS implementation of Listing 4.1 relies on full data structures containing a number of fields required in different phases of the application, event though many of them are not accessed in the graph construction stage. For example, much of the information concerning genes symbols, DNA sequence, chromosome name, etc. At a first glance, they might not seem to harm the overall performance, but the actual results do not achieve expected performance: using full data structures simply showed extremely poor scalability results, which was caused by the loading of (lots of) unused data into caches due to spatial locality. This overhead can actually saturate the memory bus, making it nearly impossible to exploit multiple processors in a multi-core system, even in the case of an embarrassingly parallel application.

The approach we propose here aims at creating data structures that only define the subset of variables used in each specific part of the program: for each of these parts, the needed data is *duplicated* and stored in novel data structures, so that the memory intensive computations can be performed using a substantially reduced working set. This permits to improve the memory bandwidth usage and to reduce cache misses. Since the duplicated data has read-only semantics, the choice of data duplication is preferred in this case, because it can be easily implemented without breaking the application logic: the original data structures are still usable in other parts of the code (e.g., friendly print output results). In cases where the duplication is not affordable, it is also possible to optimise data structures at the price of a more complex software design, with the need of substantial refactoring of all the source code.

Listing 4.2 presents a pseudo-code with a parallelised implementation of the graph construction phase, where a `ParallelFor` pattern is used. `Q` represents our working queue, that contains unique genes discovered throughout the current iteration. `L_MAX` determines the maximum distance from the root that has to be reached: in this way we can decide the coverage of our search. `C[NTH]`, `V[NTH]` and `E[NTH]` are used to store per-thread local data, where `NTH` defines the degree of parallelism to be used (i.e., the number of threads in use) and `th` identifies thread's own container, such that $0 \leq$ `thid` $<$ NTH. `V[NTH]` and `E[NTH]` contain the found genes and the edges so far identified, respectively, by each of the working threads. `C[NTH]` will contain all paired-ends reads

that each worker thread identifies as encompassing a gene. Γ is used at every level synchronisation to store partial graphs (Listing 4.2, row 23), where the `BuildPartalGraph` function is responsible for removing duplicate edges and returns a graph with unique vertices and edges identified so far. The definitive graph is built at the very end (Listing 4.2, row 32, `BuildGraph` function).

The algorithm starts searching for those Hi-C paired-end reads whose first end fragment encompasses the gene in focus. This yields a list of reads (connections) containing only chromosome fragments where neighbour genes may be located (Listing 4.2, rows 10–13): upon this list the search for neighbours takes place, using NTH independent threads over the set of connections (rows 15–21). Each thread looks for genes whose coordinates overlap the second end of the read. When a gene matches the test, the new found gene is added to the thread-local vertices set, and an edge is created between the considered vertex and the new one.

At each iteration level, the algorithm first collects all potential connections for a gene, and then searches for adjacent genes (neighbours), in parallel, over all the connections. At the end of each level iteration, the parallel execution is synchronized: at this point thread-local sets are processed and a *partial* graph is constructed with unique nodes and edges discovered at the current iteration level. The definitive graph is built at the end of the execution. The iterations proceed until all the nodes of the graph have been visited, or preferably up to the desired level specified through `L_MAX`.

Discussion

Notoriously, when threads access global data structures or shared data structures, these are potential sources of false sharing, particularly when using a single-heap allocator that gives to many threads parts of the same cache line (see Section 2.2). In our implementation we tried to minimise the number of dynamic memory allocations performed by each thread, mostly using pre-allocated containers (e.g., C++ vectors) for read-only accesses: memory allocation is performed padding each allocated slab to the cache line size, so that objects are spaced far enough apart in memory that they cannot reside on the same cache

line, thus limiting false sharing. A drawback here is the risk of memory blowup, due to the padding that slightly increases each object's size, and the overall used memory: this risk can be minimised by adhering to the well know rule of thumb, which dictates that all dynamic allocated memory must be freed, in order to avoid memory leaks and undesirable memory fragmentation.

NuChart-II's graph construction performance is strongly affected by the size of data structures used throughout the computation: most of biology-related applications deal with memory-bound problems, as a consequence of the huge amount of data that are normally involved in data analysis and simulations. Datasets used for DNA exploration are ordered sets of genomic features — such as paired-end reads, chromosome fragments, human genes labelled with genes names — whose sizes range between tens of Megabytes to several Gigabytes.

Datasets from Hi-C experiments easily reach several Gigabytes in size, and they are normally used in an application together with supplementary data: at run-time, the total memory load quickly grows up, easily exceeding 8 GB of used memory for a 4 GB Hi-C dataset. The use of a dedicated memory pool for datasets allocation, as we attempted to realise, reduces memory fragmentation and avoids memory leaks, but can easily over-load main memory, causing the OS to swap out pages and irremediably compromising performance. The working set reduction alleviates this problem: by only keeping actually needed "fields", data structures are contiguous and consecutively accessed. In this way the underlying cache optimisation mechanism works more efficiently, less unused data is loaded into the cache and more memory bandwidth is available.

4.3 Normalisation

Particular attention is given to the detection and normalisation of systematic biases — as already discussed in Section 3.3.1 — that can be associated with sequencing platforms (such as GC-content) and read alignment (such as mappability), while others are specific to Hi-C experiments (such as the frequency of restriction sites). A normalisation process is needed to remove these biases

and avoid false-positives or false-negatives results, that could lead to incorrect data interpretation.

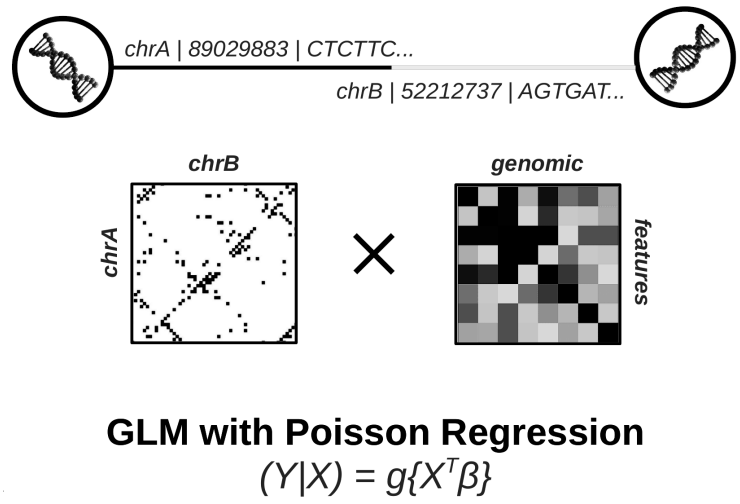


Figure 4.2: *A graphical representation of the edges weighing procedure*

In Section 3.4 we discussed the achievements by Hu et al. [76]: based on their technique, NuChart-II applies an *ex-post* normalisation to each edge of the resulting neighbourhood graph.

The presence of an edge is related to the existence of a Hi-C read that encompasses two genes: normalising each edge using genomic features — which may include the DNA sequence, genes and gene order, regulatory sequences and other genomic structural landmarks — yields a significance estimate of fragments interactions. Such estimate is then used as the weight of the edge, that assumes the role of likelihood of physical proximity for the involved genes. Using the local genomic features that describe the chromosome (fragment length, GC-content and mappability), we can set up a *generalized linear model* (GLM) with Poisson regression, with which we estimate the maximum likelihood of the model parameters (Figure 4.2). The model is given by the formula:

$$e(Y) = g\{X^T\beta\}.$$

Here Y, the dependent variable, is the contact map that contains the measured contact frequencies: the assumption of this GLM is that the measured interaction frequencies are generated from a particular distribution in the exponential family, the *Poisson* distribution in our case, which is used to count the occurrences in a fixed amount of space. X is the independent variable, which is built from chromosome length and GC-content, measured for each locus of the contact map. β denotes the parameter vector to be estimated: $X^T\beta$ is thus the *linear predictor*, that is the quantity which incorporates the information about the independent variables into the model. It is related to the expected value of the data through the link function, g, which is the natural logarithm in our case because it is the canonical link function used with a Poisson distribution.

The maximum likelihood estimates for each edge is computed using the *Iteratively Weighted Least Squares* algorithm (IWLS), proposed by Nelder and Wedderburn [116]. The best-fit coefficients returned by the linear regression are used to compute the final score of an edge, so that the edge contains an estimate of the physical proximity between the two genes it links, plus the genomic information for both genes. Listing 4.3 reports the pseudo-code for the normalisation of a single edge.

For each edge, a contact map (`CMap`) is constructed directly modelling the read count data at a resolution level of 1 Mb (or according to the resolution of the Hi-C experiments used) for the chromosomes identified by the Hi-C read. The rows and the columns of the contact map correspond to genomic regions (bins), and each point of $CMap_{i,j}$ denotes the intensity of the interaction between positions i and j. The contact frequency between two bins relies on their spatial proximity, and thus it is expected to reflect their distance. Also, Hi-C data matrix is symmetric, thus we consider only its upper triangular part (denoted with '^'). Other matrices are built after parsing text files containing the required values for each locus of interest. Text files containing these feature have been downloaded from online repositories (e.g., NCBI[22], EBI[23]). Model components are built using arrays containing such upper-triangular values (excluding the diagonal: diagonal values are all zeros, because a chromosome locus does

[22] `http://www.ncbi.nlm.nih.gov/`
[23] `https://www.ebi.ac.uk/`

```
1 NormaliseEdge(e, τ, NTH) {
2    LenM = GCcM = MapM = CMap := ∅  // genomic features matrices
3    X = Y = β := ∅
4    Conv := false
5
6    // all matrices have the same size
7    CMap := ContactMap(e.Chr1, e.Chr2)
8    LenM := BuildLengthMatrix(e)
9    GCcM := BuildGCcontentMatrix(e)
10   MapM := BuildMappabilityMatrix(e)
11
12   X := Matrix(LenM^, GCcM^)
13   Y := Cmap^
14
15   while (not Conv) {
16      ApplyLinkFunction(Y)
17      β := ApplyGLM(Y, X, MapM^)
18      Conv := CheckConvergence(β)
19   }
20   e.Weight := f(β)
21 }
```

Listing 4.3: *Normalisation*

not interact with itself), thus halving the memory consumption for each edge analysed.

The edges weighing phase is a data parallel application, where any arbitrary subset of the edges can be processed independently from each other by mean of a parallel loop pattern. This data parallelism can be properly exploited to boost up performances and drastically reduce execution time, by just using the code in Listing 4.3 as the lambda function executed by the `ParallelFor` pattern: the skeleton will be responsible for partitioning the data structure containing all the edges, and will assign a bunch of edges to each worker.

The regression is run until a convergence criterion is met: in our case, we check that the absolute value of the χ^2 (*chi-squared*) difference at each iteration is less than a certain threshold τ:

$$|\chi^2 - \chi^2_{old}| < \tau.$$

In Listing 4.3, the function `ApplyGLM` writes the best-fit parameters in vector

β, which is the result of the regression: these coefficients are used to calculate the score (i.e. the estimation of physical proximity) for the edge connecting the two genes. Also, we compute *dispersion* and *standard error*, so as to provide a useful summary of model fit.

Figure 4.3 shows a neighbourhood graph for the gene TP53. The root gene is yellow coloured. The neighbourhood graph at *Level 2* already shows some important evidences on how the genes interact and "socially" behave: few adjacent neighbours of the root gene TP53 exhibit a high degree of connected components: the role of *hub* that the two genes (KIA0753 and PHF2) acquire, suggest their importance in maintaining the interactions in that particular genomic region.

4.4 Experiments

NuChart-II has been designed to overcome the weaknesses of the R prototype, which had significant bottlenecks in memory management and limitations in the exploitation of the available computational resources, causing restrictions in the usability of the tool. This novel implementation addresses these weaknesses, making possible a genome-wide exploration of Hi-C contacts — a graph that comprises almost all human genes can now easily be constructed from a Hi-C experiments over a cell line. The optimal memory management and data structure design helped in achieving this result, with outstanding improvements in terms of execution time obtained exploiting loop parallelism techniques on multi-core architectures. We have conducted a number of experiments to verify correctness and goodness of NuChart-II: starting from the work in [107] we have replicated some of the tests conducted there, in order to have a basis for comparing the accuracy of the results. We have increased the number of iterations to further explore genes' neighbourhood, while also testing the novel tool on bigger datasets.

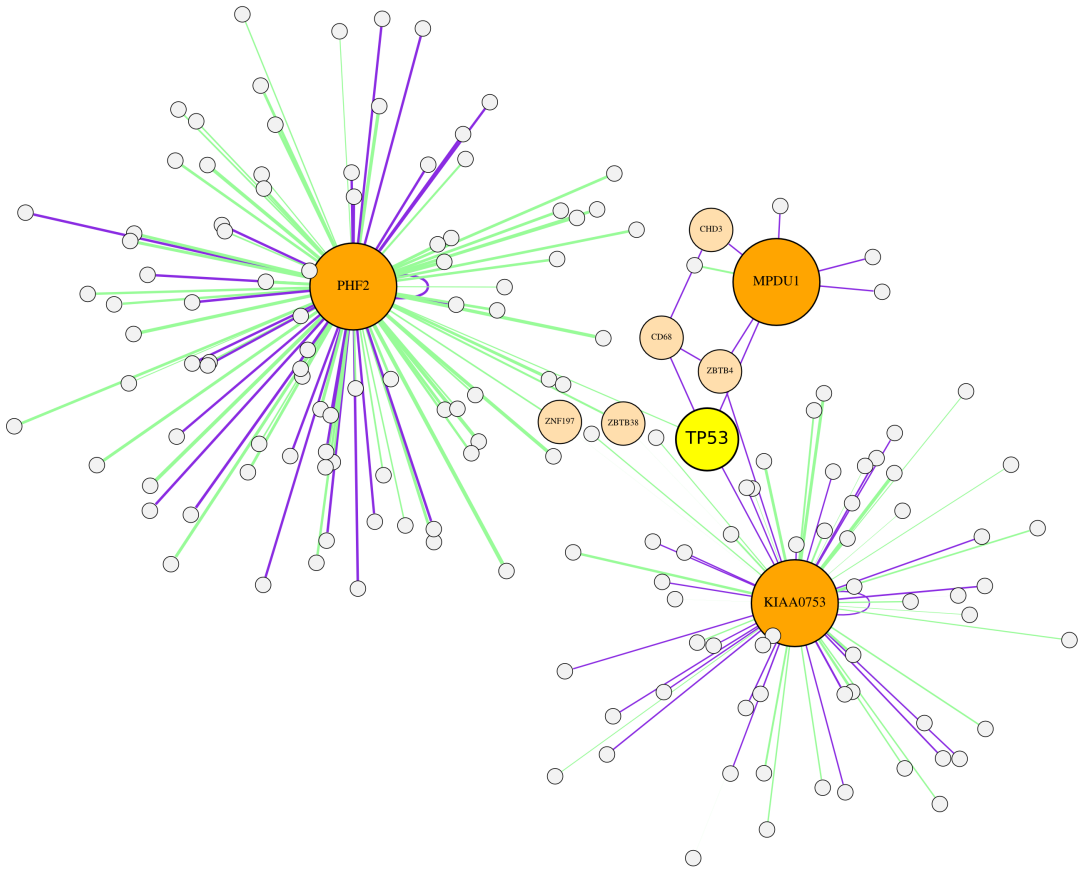


Figure 4.3: *Neighbourhood graph for gene TP53 according to LiebermanAiden's experiment SRR027963, with neighbours distant 2 hops from the root. Green edges are* inter-*chromosome connections; violet edges are* intra-*chromosome connections. Edges thickness reflects edges weight*

DNA Exploration

We created several neighbourhood graphs for relevant genes or gene clusters, at different levels of iterations, in order to verify how Hi-C can be used for cytogenetics studies. In particular, we focused on *Philadelphia* translocation, which is a specific chromosomal abnormality that is associated with chronic myelogenous leukaemia (CML). The presence of this translocation is a highly sensitive test for CML, since 95% of people with CML have this abnormality, although sometimes it occurs also in acute lymphoblastic leukaemia (ALL) and in acute myelogenous leukaemia (AML). The result of this translocation is that a fusion

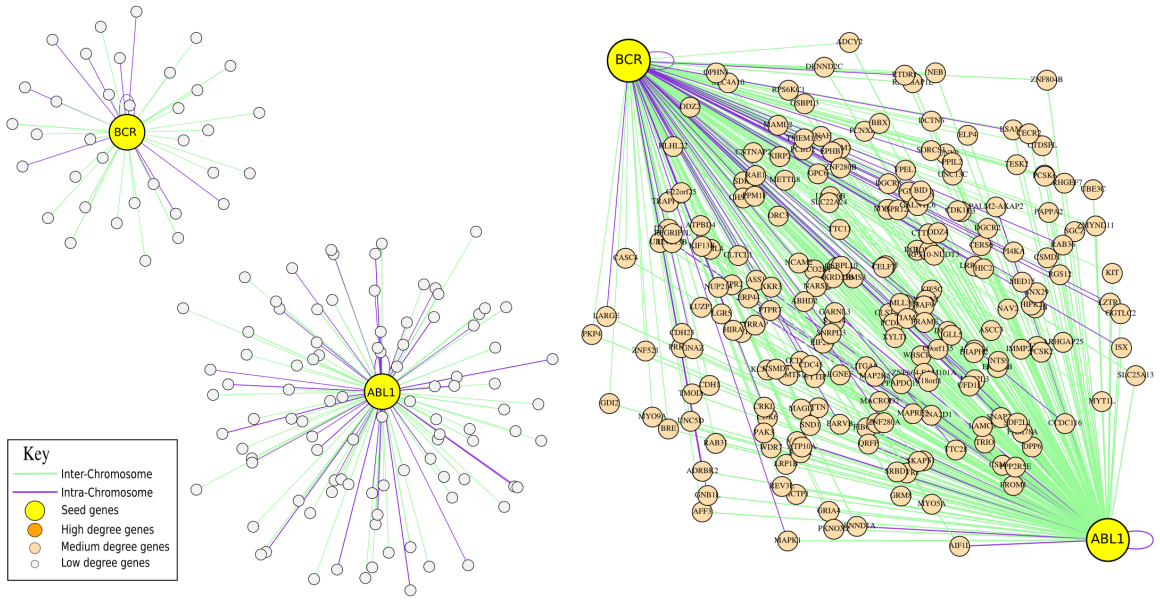


Figure 4.4: *Neighbourhood graph with genes ABL1 and BCR, according to LiebermanAiden's SRR027956 (*left*) and SRR027962 (*right*) experiments*

gene is created from the juxtaposition of the ABL1 gene on chromosome 9 (region q34) to part of the BCR ("breakpoint cluster region") gene on chromosome 22 (region q11). This is a reciprocal translocation, creating an elongated chromosome 9 (called *der 9*), and a truncated chromosome 22 (called the Philadelphia chromosome). The Hi-C technique can be used to study such kind of translocations, and subsequently dare to answer questions such as "are these kinds of chromosomal translocations occurring between nearby chromosomes?".

With NuChart-II we compared the distance of some couples of genes that are known to create translocation in CML/AML. In particular, our analysis relies on data from LiebermanAiden experiments [102], which consist of 4 lines of karyotypically normal human lymphoblastoid cell line (GM06990) sequenced with Illumina Genome Analyzer, compared with 2 lines of K562 cells, an erythroleukemia cell line with an aberrant karyotype. We used well-established data related to the cytogenetic experiments, and we tried to understand if the Hi-C technology can successfully be applied in this context, by verifying if translocations that are normally identified using Fluorescence *in situ* hybridization (FISH) can also be studied using 3C data.

We studied 5 well known couples of genes involved in translocations and we analysed their Hi-C probability contacts in physiological and diseased cells. For validating the presence of an edge in the graph, we used the *p-value* function as a test for quantifying the statistical significance of our experiments. Considering a $p < 0.05$ threshold, we see that ABL1 and BCR are distant 2 or 3 contacts in sequencing runs concerning the GM06990 cell line (Figure 4.4, left), while they are in close contact according to runs related to the K562 cell line (Figure 4.4, right)[24].

This implies that the DNA conformation in cells is effectively correlated to the disease state and also that Hi-C can be reliable in identifying these cytogenetic patterns. Neighbourhood graphs built for AML1 and ETO genes in leukaemia cells (Figure 4.5, bottom) show a considerable number of "shared" genes in between the two — meaning that there is a bunch of genes which are in close spatial proximity with both AML1 and ETO, according to the Hi-C experiment. The presence of these entities likely affects the gene regulation process, thus confirming the high probability of translocations happening. A betweenness analysis in this graph highlighted the importance of the shared genes for playing a part in every possible interactions among the two graphs. In normal cells (Figure 4.5, top), only two genes are shared, and a fast glance at edges thickness reveals a low probability of spatial proximity for the two genes with respect to AML1 (RUNX1, yellow node on the right) thus justifying a lower probability for a translocation to happen.

Considering the translocation CBFβ-MYH11, they are distant 2 or 3 contacts in GM06990 (Figure 4.6, left), while they are proximal in K562 (Figure 4.6, right).

These results are of utmost importance for the biomedical community: with the decreasing of sequencing costs, the Hi-C technique can be an effective diagnostic option for cytogenetic analysis, with the possibility of improving the knowledge on chromosomal architecture nuclear organisation. For example, Hi-C can be used to infer non trivial risk markers related to aberrant chromosomal conformation, like the Msc5a loci for breast cancer, which is known to play a critical role in the re-organization of a portion of chromosome 9 by CTCF proteins.

[24] LiebermanAiden's experiment presented in [102]

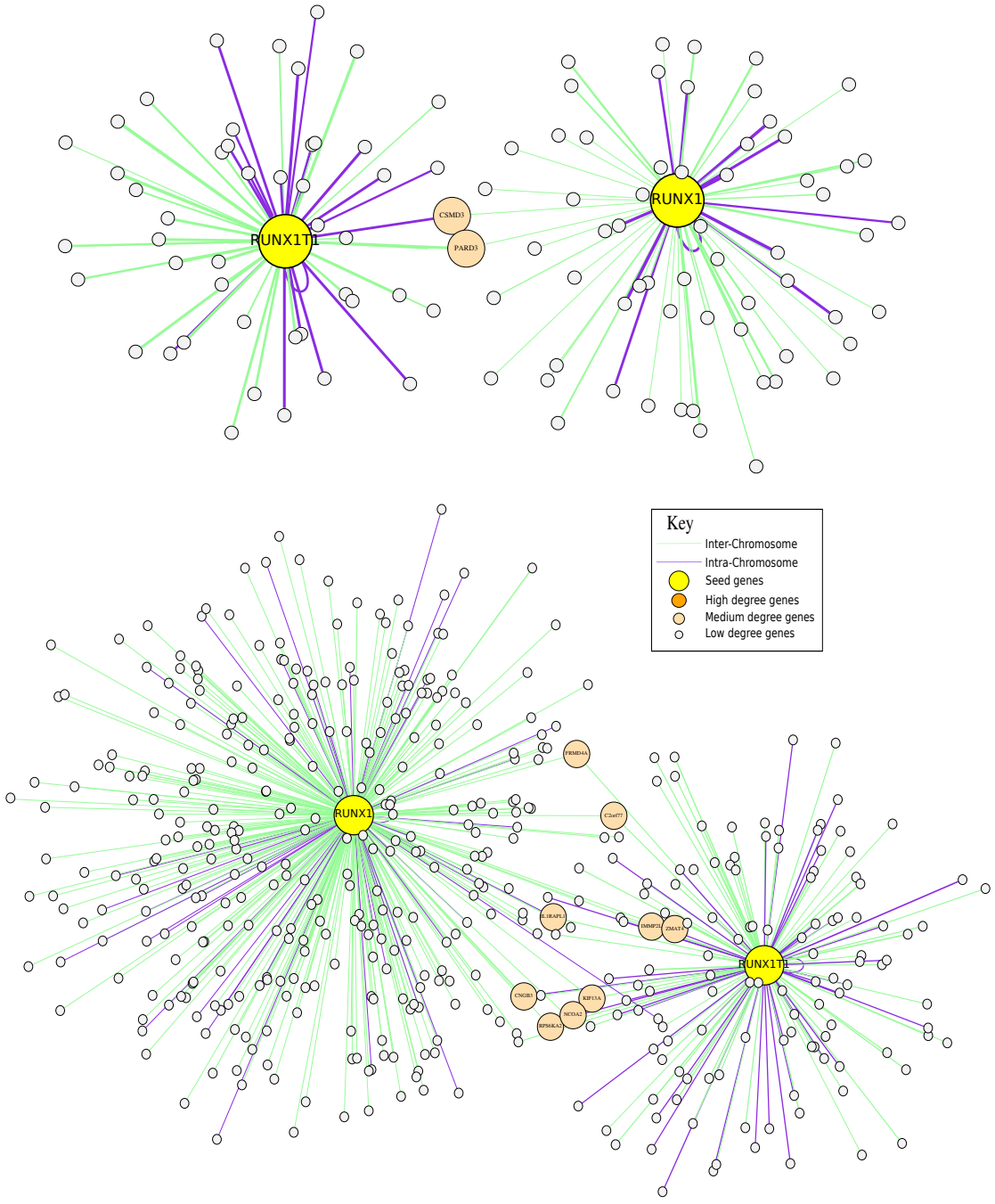


Figure 4.5: *Neighbourhood graph for genes AML1 (RUNX1) and ETO (RUNXIT1) according to LiebermanAiden's SRR027956 (top) and SRR027962 (bottom) experiments*

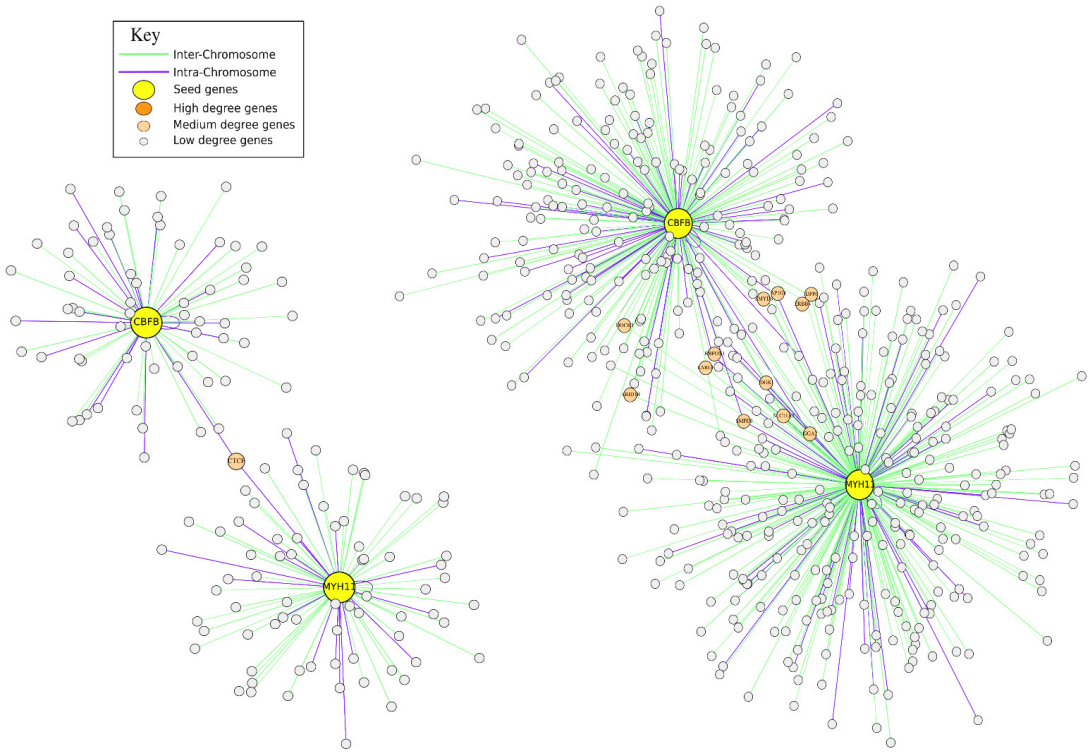


Figure 4.6: *Neighbourhood graph with genes CBFβ and MYH11 according to LiebermanAiden's SRR027959 (left) and SRR027963 (right) experiments*

Concerning visualisation, a drawback is that the readability is dramatically compromised when the number of nodes and edges increases, likely resulting in a tangle of edges hardly understandable. NuChart-II supports plotting with *iGraph* and *GraphViz*: these tools perform nicely with small-to-medium sized graphs, but cannot provide useful representation of huge graphs with more than ten thousand edges (as it happens when the diameter of the graph increases). Textual and tabular outputs become useful for the analysis of the genomic regions explored: the probability of a connection can be estimated by evaluating an edge's weight, while the overall graph structure is shown in terms of the distance of each discovered gene from the root(s).

Dataset	Size	Nodes	Edges	Time (ms)
SRR027956	576 MB	18019	356720	69144.5
SRR027957	665 MB	18086	414964	81948.2
SRR027962	1.9 GB	18605	1049599	219727
SRR027963	958 MB	18450	588635	115111

Table 4.1: *Genome-wide graphs built from different datasets starting from gene ABL1*

4.5 Discussion

Chromosome conformation capture data actually reveal *long-range* contacts, either between locus pairs that are on the same chromosome but far from each other (long-range *intra-chromosomal*), or when detected on different chromosomes (*inter-chromosomal*). Identifying statistically significant contacts between two intra-chromosomal loci is not trivial and depends heavily on the genomic distance between the loci, that is affected by random looping of the DNA. On the other hand, with inter-chromosomal loci it is much simpler: once biases are eliminated by normalization, in the absence of any prior information on the pairwise distances among chromosomes all possible pairs of inter chromosomal loci are expected to interact equally, under the null hypothesis.

This is because the probability of finding a contact on different chromosomes is so low that even one contact might be significant: highly self-interacting regions, named Topologically Associated Domains (TADs), have been recognised as regions of chromatin (bins) that fold as discrete three-dimensional (3D) structures [48]. They tend to favour internal, rather than external, chromatin interactions, and are bounded by narrow segments where the chromatin interactions appear to end abruptly. When we apply our normalisation on the edges of the graphs we aim at controlling this random polymer looping by assigning statistical significance to the observed contact counts, and we operate both on intra-chromosome and inter-chromosome contacts.

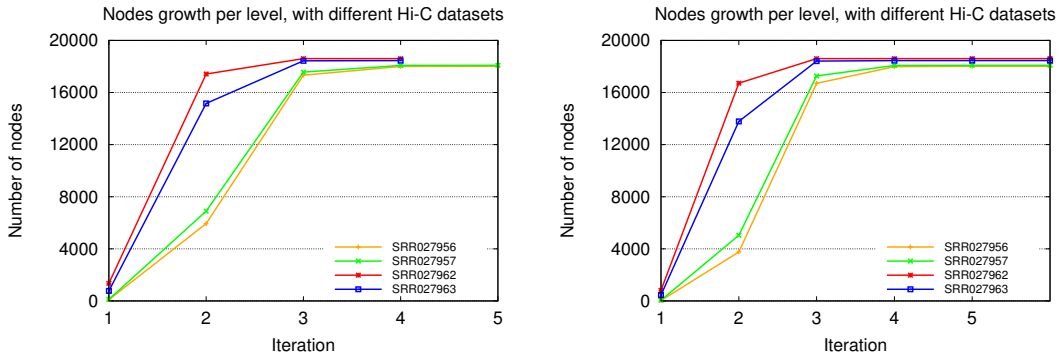


Figure 4.7: *The graph growing trend, as described by graphs created from genes ABL1 (left) and BCR (right) on different Hi-C experiments, shows that a graph almost reaches its full size at the second (or third) iteration, depending on the Hi-C experiment used as dataset*

4.5.1 Network Analysis and Statistics

Hi-C datasets containing paired-ends reads are the principal input of NuChart-II: their resolution determines the size of the resulting graph and affects execution time and memory load. Table 4.1 reports some details concerning graphs constructed from gene ABL1, using different Hi-C datasets produced by LiebermanAiden's Hi-C experiments [102]: two resulting from experiments conducted over normal cells (SRR027956, SRR027957), two obtained from experiments conducted over leukaemia cells (SRR027962, SRR027963). Dataset size is reported, together with graph sizes and execution times.

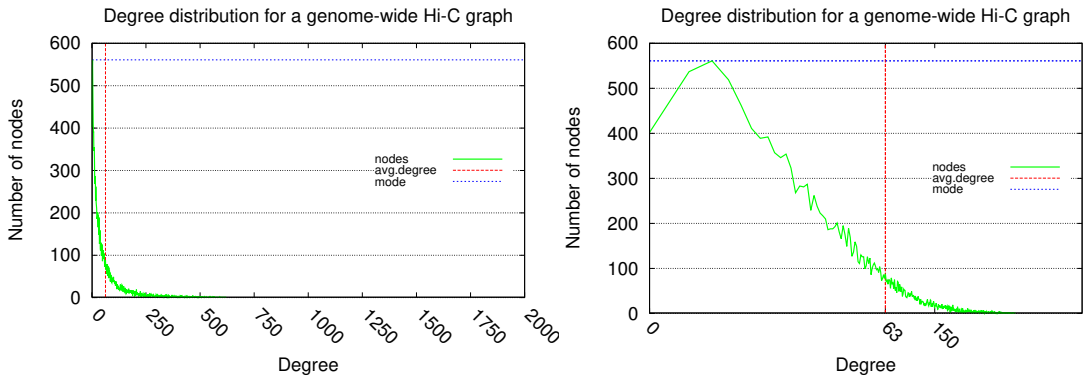
Interestingly enough, starting from a single gene a graph almost reaches its full size at the second (or third) iteration, depending on the Hi-C experiment used as dataset (Figure 4.7). This behaviour again highlights the three-dimensional conformation of the chromosome in the nucleus of a cell, and can explain why long-range interactions between genes and distal regulatory elements can happen: the best-studied long-range interactions are those between genes and distal regulatory elements, such as enhancers. Such long-range interactions are made possible by chromatin looping in the 3D space, allowing gene regulatory elements to interact with promoters through direct protein interactions [64].

Ideally, chromosomes could be represented as masses of spaghetti-like fibers, packaged and stretched to fit into cells nuclei. TADs compartmentalise chromosome into discrete domains, characterized by frequent long-range interactions of loci in the same domain but less frequent interactions of loci in adjacent domains. TADs are in turn folded into sub-TADs, that suggest a hierarchical organisation of the chromatin.

The graph-based approach has been proved to be a valuable way for the interpretation of genomic information by mean of complex, dynamical structures that organize items in an integrated way. Furthermore, it opens new perspectives on the study of the 3D chromosome conformation and the genes interaction: the *social network* point of view allows to study the relationships among genes in terms of network theory.

Representing a genome as a graph changes the mindset, and permits to focus on interactions among genes, which can in turn be interpreted using graph theory and network analysis. For instance, centrality metrics describe the interactions that (may) occur among local entities and help to identify the most important and influential genes. Ranking of nodes by topological features (such as degree distribution) can help to prioritize targets of further studies or lead to a more local, in-deep analysis of specific chromosome locations. Here studies of functional similarity can suggest new testable hypotheses [162].

The degree distribution in all graphs we have produced from Hi-C experiments asymptotically follows a power law distribution (see Figure 4.8), and leads us to consider genes networks as *scale-free* networks: few nodes are likely to act as "hubs", serving for specific purposes in their networks. The clustering coefficient distribution also follows a power law, suggesting that low-degree nodes belong to very dense sub-graphs and those sub-graphs are connected to each other through hubs. This opens to new hypothesis and further investigations, towards detecting the presence of community structures and study the mechanisms that drive tie formations and genes interaction.



(a) *degr. distrib* **(b)** *zoom in*

Figure 4.8: *The two plots above show the degree distribution of a genome-wide graph built starting from the BOLA3 human gene, according to the LiebermanAiden's SRR027962 experiment. The resulting graph has 18450 genes and 588635 edges. The average degree of the graph is 63, the maximum degree for a node is 2217 and the most frequent degree among the nodes is 561.*

4.5.2 Performance

The memory size required to hold the data strongly affects performance of both phases of the application, particularly the graph construction phase. We have accurately tuned the crucial steps in order to maximize the use of memory hierarchy and fully exploit cache locality, while minimising cache trashing.

Our first target architecture is a NUMA workstation equipped with 4 computing nodes, each having an eight-cores Intel E7-4820 Nehalem processors running at 2.0GHz, featuring 18MB L3 cache per NUMA node, 256KB L2 cache and 64KB L1 cache with 64 GB of main memory. The Nehalem processors use hyper-threading with 2 contexts per core. We use up to 32 threads in our executions, in order to exploit all physical cores without making use of the second context. We used the GNU gcc 4.8.0 compiler with the optimisation flag '-O3'. Thanks to the internal structure of FastFlow's *ParallelFor*, it is possible to use all physical cores while thread pinning is automatically managed by the FastFlow library.

Considering that the application is also tested on a NUMA platform, we alse executed NuChart-II using an interleaved memory allocation policy via the *numactl* utility, which gives some control over memory policies on NUMA systems.

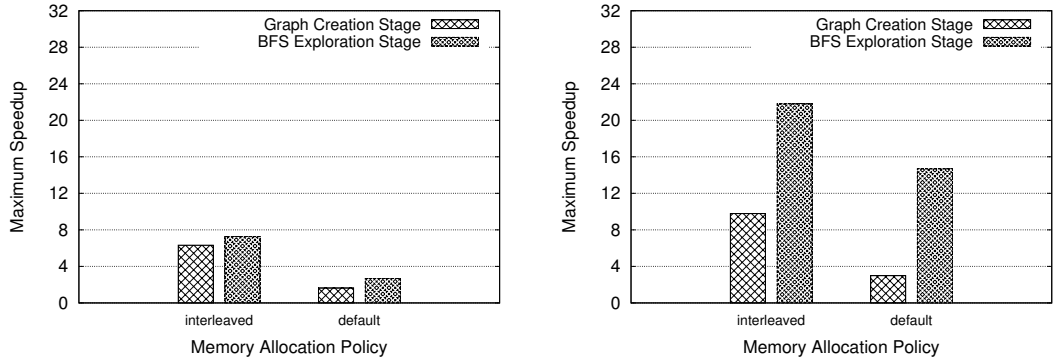


Figure 4.9: *The chart on the* left *shows the execution of NuChart-II with no optimisations. The chart on the* right *refers to the memory-optimised implementation of NuChart-II. The maximum speedup obtained by the parallel execution of the graph creation and BFS exploration phases is reported.*

Graph construction

In Figure 4.9 we show the maximum speedup obtained executing NuChart-II without optimisations (*left*), with a simple inclusion of the BFS graph exploration within the `ParallelFor` skeleton. Both interleaved and default memory allocation policy have been tested: the interleaved policy permits to allocate memory pages in a round-robin fashion over all nodes in the NUMA system. This allocation strategy usually leads to some advantages in terms of memory bandwidth, since spreading the memory load across all nodes prevents a single memory interface to become a bottleneck, at the cost of higher memory access latencies in case of a remote memory access.

However, the gain is still minimal and negligible, and the reason can be found on poor memory hierarchy exploitation: a frequent context in this graph exploration is to have huge arrays of objects (e.g., paired-end reads, chromosome fragments, genomic features, etc.) from which we wish to retrieve those who match a given criteria (for example based on *start* and *end* coordinates). When a matching is found, the program goes on employing the found variables to perform further operations using other arrays of custom objects that model our datasets. On average, only few elements match the criteria, and many values will be loaded into the cache even though they are not going to be used by the program. Since cached memory accesses are optimised for contiguous

	Default Memory Allocation		Interleaved Memory Allocation	
#Threads	**Graph Creation**	**BFS**	**Graph Creation**	**BFS**
1	137	1006	133	962
2	83	522	77	502
4	77	310	43	248
8	72	328	31	173
16	68	357	27	138
32	63	331	21	132

Table 4.2: *Execution times (seconds) for the graph construction without optimisations*

consecutive accesses, data that is fetched is usually larger than one primitive variable. In our context, explicitly reading two coordinates typically translates to automatically loading into the cache some (or maybe all) consecutive variables from the same object.

Table 4.2 refers to execution times obtained using up to 32 parallel threads. Each BFS exploration execution time is the sum of all times needed to explore the graph at each level, until the fix point is reached. At each level, the number of nodes reached is highly unbalanced, resulting in very different execution times during the BFS exploration.

The optimisation we propose minimises the needed modifications on existing code, but creates *ad-hoc*, trivial *POD* data structures[25] that only contain variables needed for each specific phase of the algorithm. Arrays of these data structures are contiguous and can be consecutively accessed: chances are that the underlying cache optimisation mechanisms will work more efficiently and less unused data will be loaded into the cache (meaning more useful memory bandwidth available).

Table 4.3 shows execution times obtained with the memory-optimised implementation using up to 32 parallel threads. Each BFS exploration execution time is the sum of all times needed to explore the graph at each level. It can be noticed that the total running time of the BFS step in the sequential run is about

[25]POD is an acronym for Plain Old Data. A POD data structure is an aggregate class that contains only PODs as members, has no user-defined destructor, no user-defined copy assignment operator, and no nonstatic members of pointer-to-member type

	Default Memory Allocation		Interleaved Memory Allocation	
#Threads	**Graph Creation**	**BFS**	**Graph Creation**	**BFS**
1	84	570	107	566
2	63	291	59	289
4	39	141	31	139
8	34	76	17	73
16	31	51	12	44
20	29	50	11	37
24	28	43	11	31
28	29	39	11	27
32	28	38	10	26

Table 4.3: *Execution times (seconds) for the memory-optimised graph construction implementation*

twice faster compared to the sequential execution of the non-optimised implementation, considering both memory allocation policy. The best performance is achieved by using memory interleaving, obtaining a maximum speedup of 21.81 by using 32 threads.

It can be noticed how performance improves dramatically, obtaining a maximum speedup of ~ 22 starting from a speedup of ~ 7 of the naive implementation. The reduction of the working set permits to better exploit caches and, accordingly, the algorithm makes fewer requests to main memory to retrieve data, thus speeding up the computation.

Normalisation

The edges weighing phase is an embarrassingly parallel application: any arbitrary subset of the edges can be processed independently from each other by means of a parallel loop pattern. With FastFlow's `ParallelFor` this data-parallelism can be properly exploited to boost up performances and drastically reduce execution time. This can be accomplished by simply defining our weighing kernel as the lambda function of the `ParallelFor`.

During execution, each worker thread gets a bunch of edges to work on, according to the grain size: we have found that the best performances are reached when the grain size is purposely kept small. Each thread uses three thread-local read-only static data structures that hold information about local genomic

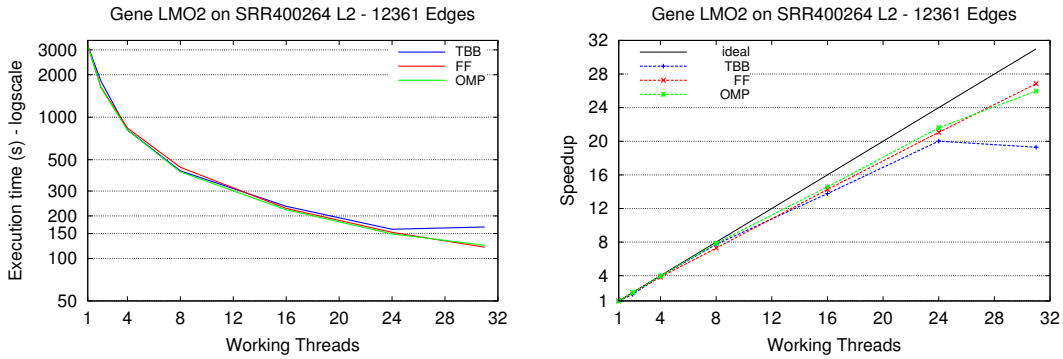


Figure 4.10: *Execution time (left) and speedup (right) of the normalisation phase for 12361 edges, resulting from a neighbourhood graph for the gene LMO2 with genes distant up to 2 hops from the root, according to the Dixon et al. SRR400264 experiment [48]*

features. These data are used to build all matrices needed to construct the regression workspace (see Listing 4.3). The task presents loops doing Floating Point arithmetic calculations on data that has purposely been modelled in order to fit the L3 cache, so as to fully benefit from compiler optimizations and vectorization. On the other hand, a number of dynamic memory allocations are necessary during the execution of the normalisation step. The use of a memory allocator not designed for parallel programming causes a serialization of the operations, that leads to a reduction of the total execution time.

Despite the large memory footprint, the implementation based on FastFlow reaches a quasi-ideal speedup: the memory intensive computations performed hide the latency to memory accesses, and when compared against OpenMP and Intel TBB, the recorded performances are substantially similar (Figure 4.10). Intel TBB begins to suffer for the dynamic memory allocations when the number of threads is greater than 24, causing its performance trend to flatten.

Tests have been conducted using as much similar configurations as possible, trying either with static scheduling or with dynamic scheduling and variable chunk size. With Intel TBB's `parallel_for` we used the *affinity partitioner*, because it attempts to perform some automatic cache optimizations. Eventually, it did not bring substantial improvements with respect to the default *auto partitioner*. With FastFlow's `ParallelFor`, we found that the best

performance was reached when using the scheduler as thread: the scheduler can be adapted to be run as thread or as an object. When the former solution is chosen, the number of running threads is $\#worker\_threads + 1$. When this number equals the number of cores, the extra thread used (which performs busy-waiting during synchronisation) introduces non negligible overhead, especially in fine grain computations. We have anyway noticed that this configuration yields more desirable results in terms of overall performance. For the OpenMP `parallel for` evaluation, we used the dynamic scheduling, because it produced better performance with respect to dynamic scheduling.

4.6 Concluding remarks

The novel implementation of NuChart-II allows the software to scale genome-wide, which is crucial to exploit its full capability for a correct analysis, interpretation and visualisation of the chromosome conformation. This graph-based approach opens new perspectives for the analysis and processing of Hi-C data, focusing on the interactions of a gene with its neighbourhood.

We have shown that a dataset reduction might be the key for a substantial performance improvement in memory-bound algorithms: at each specific stage of Hi-C data analysis, we drop unneeded fields that would saturate the memory bus, leading to an overall performance degradation. A working set reduction brings immediate improvements in memory bandwidth and cache utilisation, taking full advantage of multi-core architectures. The normalisation phase has been revisited and provides a valuable estimate of physical proximity for two genes, while keeping available all genomic data related to the spatial region where the genes lie. Such genome-wide exploration and analysis is possible with the aid of novel high-level parallel programming patterns, that allow many of the issues that burdened the original R prototype to be addressed, obtaining performances that would have been inconceivable with the original R prototype.

Chapter 5

Nucha*R*t: embedding NuChart-II in R

In this chapter we discuss the integration of our C++ application into the R environment, an important step toward our objective of augmenting the usability of Bioinformatics tools: we aim at obtaining a high-performance pipeline that allows users to orchestrate analysis and visualisation of multi-omics data, making optimal use of the computing capabilities offered by modern multi-core architectures, combined with the versatile and well known R environment for statistical analysis and data visualisation.

5.1 Motivation: efficiency and usability

In the previous chapter we described NuChart-II as a highly optimised, C++ application designed to integrate information about genes positions and paired-end reads resulting from Hi-C experiments. The goal is to build a sort of topographical map of the chromosome spatial organisation using a gene-centric, graph-based approach [107, 149]. We have investigated the possibility of introducing network concepts to represent the behaviour of genomic actors: a network (or graph) has a high level of expressiveness, since nodes represent the actors of a process while edges identify relationships among the actors. Structural properties of a network can reveal significant information on how the actors of

the represented process interact, while parallel algorithms can be employed to operate over a network. The graph-based is a valuable way for the interpretation of genomic information by means of complex, dynamical structures that organize items in an integrated way.

NuChart-II has been designed using high-level parallel programming patterns [38, 4] that facilitate the implementation of the algorithms employed over the graph: this choice permits to boost performances while conducting genome-wide analysis of the DNA. Furthermore, the coupled usage of C++ with advanced techniques of parallel computing (such as lock-free algorithms and memory affinity) makes possible to process much faster, much more data: informative results can be achieved to an unprecedented degree [50].

However, parallel programming in C++ is not widely used in Bioinformatics, because it requires highly specialised skills and does not fully support the rapid development of new interactive pipelines. Conversely, the modularity of R and the huge amount of already existing statistical packages facilitate the integration of exploratory data analysis and permit to easily move through the steps of model development, from data analysis to implementation and visualisation.

In order to improve the usability of the software while preserving the high performance achievable with NuChart-II, we opted to combine it with the R environment, developing a package that can fulfil the needs of a fast and usable tool for Hi-C data interpretation. We used *Rcpp* to bridge C++ and R: Rcpp comes as an R add-on package that provides a consistent API for accessing, extending or modifying R objects at the C++ level [52]. It can be used to accelerate computing by replacing an R function with its C++ equivalent and facilitates data interchange from R to C++ and vice-versa.

Embedding NuChart-II in R creates an application that can be used either to conduct a *step-by-step* analysis of genomic data (i.e., interactive inspection of each phases' output), or as a high-performance work-flow that takes heterogeneous datasets in input, processes data and produces a graph-based representation of the chromosomal information provided, supported by a rich set of default descriptive statistics derived from the topology of the graph.

5.1.1 Hi-C data analysis step-by-step

The Hi-C data analysis conducted with NuChart-II walks through five main steps:

1) data retrieval and parsing;

2) neighbourhood graph construction;

3) weighing of the edges as a result of data normalisation;

4) statistical analysis;

5) output and visualisation.

NuChart-II parses a number of options from CLI[26] to set up and characterise each execution. Once started, the application walks through all the steps outlined above in a "monolithic" fashion, and yields its results as a summary of the whole process: the final output is available in terms of a neighbourhood graph drawn using some plotting engine, together with formatted text files (such as `csv` files) that contain whole information necessary to examine the represented data. This include the actual sequences "contained" in edges, edges probability, network analysis metrics and various statistical annotations.

Nothing wrong about this, but genomic data analysis, as it is the case with many other scientific fields, does not work as one monolithic process: different stages of data analysis are just fundamentally different, and have different parallelism patterns, memory access and data access requirements. Also, it often makes sense to run the same stage of an analysis in a number of different ways to demonstrate the robustness of novel results or to tackle different sorts of data, for example one in which a reference genome is available, compared to one where it is not.

If we consider the possibility to map additional multi-omics features on a graph — such as genes expression, CTCF binding sites or methylation profiles — we would choose a dataset from which to gather the required information and re-execute the application from the beginning, until we get our output with

[26] Command Line Interface

mapped omics data. This means that no intermediate inspection is allowed, nor we could choose some quick statistics to satisfy whatever curiosity or to banish some doubts: an application like NuChart-II is designed to create neighbourhood graphs from Hi-C experiments, and to furnish output results via different means (graphs and textual). Little can a user do to interrogate the application during the consecutive steps.

Despite its undeniable efficiency, this lack of modularity highlights a clear limitation in usability of the C++ implementation. Concerning visualisation, as the graphs become wider and richer in nodes and edges, it gets harder to produce appreciable graphical representations.

These factors led us to re-consider R as a "hosting" environment for a scalable and usable tool for Hi-C data analysis. From the early R prototype — developed within the R environment [107] — we learned that high-performance and good memory hierarchy exploitation is hard to achieve within the R environment, due to specificities of the environment itself, and requires a substantial programming effort. Nonetheless, research during the last decade has widely explored the use of parallel computing techniques with R.

5.1.2 Parallelism facilities in R

R is an open-source programming language and a well-known software environment for statistical computing and graphics. By default, it does not take advantage of multiple processing elements available on a modern computer: if a user wants to execute a parallel application she needs first to register a sort of "back-end" that effectively permits to run a portion of code in parallel.

R's high extensibility is possible through the use of *packages*, libraries for specific functions or specific areas of study frequently created by R users and distributed under suitable licenses[27]. For what it concerns high-performance computing, many libraries exist that foster parallel programming in R, most of which focus on distributed architectures and clusters of computers. Worth

[27]Most of the R packages are available from CRAN and are listed in `https://cran.r-project.org/web/packages/available_packages_by_name.html`. Another repository that hosts several R libraries is `https://r-forge.r-project.org/`. If not otherwise stated, packages mentioned in this section can be found in these two repositories.

to mention are *Rmpi* and *Snow* (the latter also permits multi-core exploitation through socket programming).

Rmpi is a wrapper to MPI and exposes an R interface to low-level MPI functions. The package provides several R-specific functions, beside wrapping the MPI API: for example, parallel versions of the R's `apply()`-like functions, scripts to launch R instances in a distributed environment, and some error-handling to report errors from the workers to the manager. Snow (Simple Network Of Worksations) provides support for simple parallel computing on a network of workstations. It supports several different low-level communication mechanisms, including private virtual machine (PVM, via the *rpvm* package), MPI (via Rmpi) and raw sockets. The package provides high-level parallel functions like `apply()` and simple error-handling mechanism. Some extensions to the Snow package, such as *snowFT* and *snowfall*, add fault tolerance, reproducibility and additional management features, together with simplified interfaces that wrap Snow's interface.

The *multicore* package provides functions for parallel execution of R code on machines with multiple processing elements: all jobs share the full state of R when parallel instances are spawned, so no data or code needs to be copied or initialized. Spawning uses the `fork` system call (or Operating System-specific equivalent) and establishes a pipe between the master and child process. The pipe can be used to send data from the child process to the master. Note that multicore starts its workers using `fork` without doing a subsequent `exec` call. This poses limitations in the variety of operations that can be parallelized, which normally fall back into simple independent math computations on a collection of indexed data items (e.g, an array).

The *doMC* package acts as an interface between *multicore* functionalities and the *foreach* package, that provides a looping construct for general iteration over elements in a collection, without the use of an explicit loop counter. When used in combination with doMC (hence with multicore), foreach can execute looping operations on multiple processors/cores.

Starting from release 2.14.0, R includes the package **parallel** that incorporates slightly revised copies of packages *multicore* and *snow* libraries.

Some other libraries provide parallel computing facilities for multi-core systems without the use of multicore as a back-end. Among them, *pnmath* (and *pnmath0*) offer a hard-coded parallel implementation of most of the basic R numerical math routines, using OpenMP (and Pthreads, respectively) to manage low-level operations. Once loaded, these packages replace built-in math functions with hand-crafted parallel versions, and require no further changes to user code. However, these libraries only speed up things for those functions that have been specifically coded and permit no customisation. *R/parallel*[28] enables automatic parallelization of loops without data dependencies by exposing a single function: `runParallel()`. The implementation is based on C++, and combines low-level system calls to manage processes, threads and inter-process communications. The user defines which variable within the enclosed loop will store the calculation results after each iteration, and how these variables have to be operated and reduced.

These solutions bring good support for parallel computing to the traditionally single-threaded R environment: a parallelization back-end for vectorized and looping computations, with explicitly hacks of numerically intensive math routines, to provide higher efficiency and better scalability.

It is worth to mention that an interface to Intel TBB for R also exists[29], that pretty much resembles our approach and permits to use TBB's `parallel_for` pattern to convert the work of a standard serial `for` loop into a parallel one, and the `parallel_reduce` construct can be used for accumulating aggregate or other values. This solution enforces a master/slave behaviour between R and C++, so that data-parallel computations can be offloaded to C++.

We will shortly see that our approach pretty much resembles this latter one. However, in Section 4.5.2 we compared our solution based on FastFlow against TBB's `parallel_for` and OpenMP's `#pragma omp parallel for`, showing comparable — when not better — results in terms of speedup.

[28] R/parallel is available at `www.rparallel.org`
[29] `https://github.com/jjallaire/TBB`

5.1.3 Memory management in R

The notoriously "poor" memory management mechanism in R is actually a combination of multiple factors, which also include the way operating systems allocate memory. Since our development relies on Linux OS, a discussion about these factors will shed some light over this problem.

When allocating memory, R relies on the OS default memory allocator. In a Linux system this is normally the *glibc* allocator that uses the `malloc` function to wrap the underlying system call `brk`. The allocator does not use exact allocation, but normally reserves fixed-size small contiguous chunks of memory to place any new object. This behaviour normally affects memory management in all Linux systems, because when releasing parts of a chunk (imagine an object that occupies half of a chunk's size), *glibc* can't release the whole chunk until active objects exist in the chunk.

This also influences R's memory management, because R looks for contiguous bits of memory to place any new object: when allocating lots of different sized objects without a game plan, memory fragmentation dramatically grows and if R cannot find a contiguous space for a new object it returns an error message about lack of memory space. Follows that when manually deleting unused (small) objects, free blocks that are contained in a memory chunk cannot be given back to the OS, because there are other objects in the same segment that are still active. The OS takes care of these situations by swapping out pages from unused blocks, but this operation is likely to become more and more frequent if the available memory is small, because page faults occurs at higher rates and also active pages are likely to be swapped out.

R uses a *lazy* memory reclaim policy, meaning that it will not reclaim memory until it is actually needed. Hence, R might be holding on to memory because the OS hasn't yet asked for it back, or it does not need more space yet. In order to decide when to release memory, R uses a garbage collector (GC) that automatically releases memory to the OS when an object is no longer used. It does so by tracking how many references point to each object, and when there are no references pointing to an object it deletes that object and returns memory to the OS. This means that when we have one or more copies of a big object, explicitly

removing the original object does not correspond to free memory space: until references to that object exists, the memory wont be released. Even a direct call to the GC does not force R to release memory, rather it acts as a "request", but R is free to ignore it [160].

What seems clear from this description is that R has limited control over memory management mechanism, but it simply uses `malloc/free` functions plus a garbage collector. One attempt to force memory to be released to the OS is the use of the `malloc_trim` function, that explicitly forces memory release, provided that a sufficiently large chunk is ready to be released.

A solution to alleviate this problem can be found in some simple rules of good application design: constructing big objects first, so that smaller objects can fit inside the footprint left by the larger objects; pre-allocate objects in order to avoid continuous dynamic memory allocations; avoid unnecessary copies of objects and correctly free objects as soon as they are no longer needed; avoid memory leaks that cause memory fragmentation to explode.

5.2 Nucha*R*t

We aim at building a tool for Hi-C data analysis that is both efficient — in terms of speed and memory resources exploitation — and usable. We decided not to use off-the-shelf libraries for parallel computing, because of the well known R's limits in memory management: since we are dealing with a memory-bound algorithm, parallel memory-intensive tasks should be kept on C++ side, while we rely on R for setting up a usable working environment. Also, we already had a fully tested C++ solution to our problem: for this reason our choice fall into Rcpp: Rcpp facilitates data interchange from C++ to R and vice-versa. C++ objects holding the output of a computation are made available within the R environment, ready to be used as source for advanced statistical analysis, by mean of a *wrapping* mechanism based on the templated functions `Rcpp::as<>()` and `Rcpp::wrap()`. These functions convert C++ object classes into a *S expression pointer* (called `SEXP`), that can be handled on the R side to construct `Lists` or

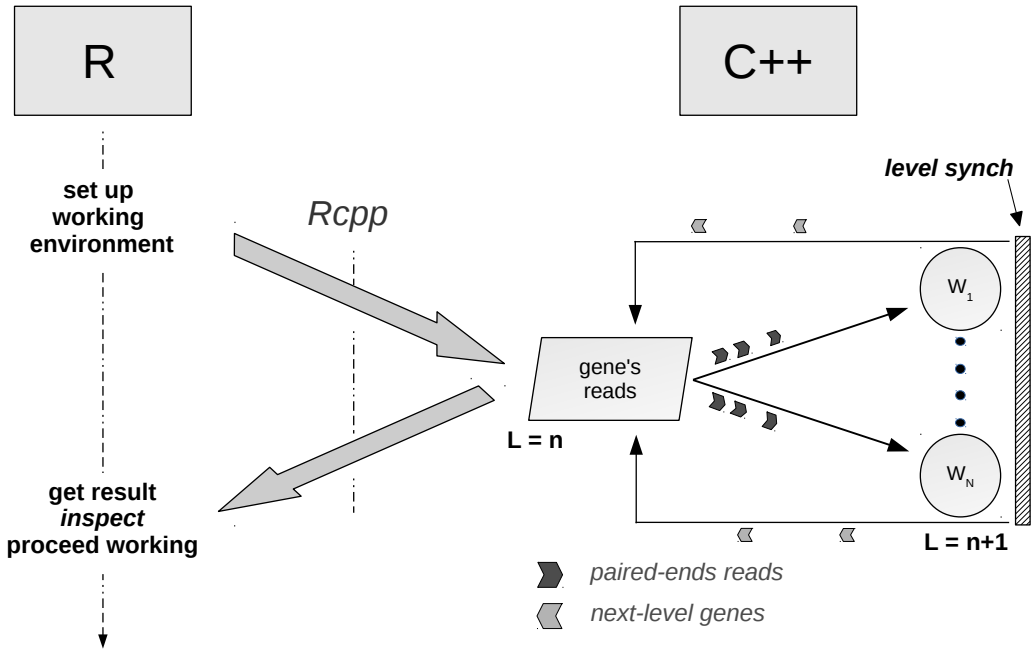


Figure 5.1: *Master/Slave behaviour between R and C++, on the graph construction phase: on the R side the "background" for the computation is set up, and then the computationally intensive task is offloaded to C++. Once finished, results are moved back to the R side.*

`DataFrames`[30], which are essential object types in R and are used by almost all modelling functions.

In this respect, our application clearly exhibits a *master/slave* behaviour: on the R side we set up the "background" for the computation, and then we offload computationally intensive tasks to C++ (see Figure 5.1). Once it terminates, the needed information is moved back to the R side and is ready to be processed, drawing from the huge R's library basket.

[30]We actually use `data.tables` as basic data structures for our data: `data.table` is an enhanced version of `data.frame` that allows us to easily optimise operations for speed and memory usage.

Nucha*R*t and Rcpp In our context, we have dealt with four C++ objects that abstract the leading actors of our software: `SamData`, `Gene`, `Fragment` and `Edge`. These objects contain much of the information that is needed to build a topographical map of the DNA from Hi-C data. Just to give an example, the `SamData` class has private fields that describe the reads, thus containing the chromosomes' names and a starting coordinate for each chromosome of the paired-ends read, plus the genomic sequence. In order to exchange a `SamData` object between C++ and R we have specialised the templated functions above: a `std::vector<SamData>` is thus treated by R as a list of `List`s, while a list of `List`s in R (or a `DataFrame`) is managed in C++ by casting the `SEXP` object to a `Rcpp::List` (or a `Rcpp::DataFrame`) object, and by subsequently filling each field of the `SamData` class with the value contained in the respective field of the `List`.

```
template<> SEXP wrap(const SamData &s) {
   List ret = List::create( Named("Id")  = s.getId(),
                            Named("Chr1") = s.getChr1(),
                            Named("Start1") = s.getStart1(),
                            Named("Chr2") = s.getChr2(),
                            Named("Start2") = s.getStart2(),
                            Named("Seq") = s.getSeq()
                          );
   return wrap(ret);
}

template<> SamData as( SEXP s ) {
   List samL = as<List>(s);
   SamData sam;

   sam.setId   (  as<long>( samL["Id"])  );
   sam.setChr1  ( as<string>(samL["Chr1"])  );
   sam.setStart1 ( as<long>( samL["Start1"]) );
   sam.setChr2  ( as<string>(samL["Chr2"]) );
   sam.setStart2 ( as<long>( samL["Start2"]) );
   sam.setSeq  (  as<string>(samL["Seq"])  );

   return sam;
}
```

Listing 5.1: *Example of `as` and `wrap` usage*

Recalling Section 5.1.1, Nucha*R*t can be described as a 5 stage pipeline: from

data retrieval to output and visualisation, these phases can now be broke up and used as loose modules. Phase *1)* is responsible for data collection and early data processing: datasets are provided as static `csv`-like files, but can also be downloaded from on-line repositories. The information contained therein is parsed and processed, in order to build the data collections needed to perform the computations: unneeded fields are dropped and elements are ordered in a consistent way, while a unique identifier for each element of a collection is generated, when needed. Thanks to novel advanced data types available in the R environment (e.g., `data.table`), we can parse and store big datasets on the R side and efficiently operate with such datasets, provided that the machine has enough physical memory to store data. This step encompasses Hi-C datasets parsing and processing, with file sizes that range from about 1 GigaByte to several GigaBytes: as the file size increases, file parsing and processing time also grows, up to some minutes. This operation is however unavoidable, and little can be done to speed it up. Data interchange between R and C++ can be accomplished through the wrapping mechanism explained above, which also promotes loose modularity among analysis steps: datasets that are only needed for a specific phase are loaded on demand, used and removed when finished, thus lowering and optimising memory utilisation, while main collections are kept alive in a global environment for immediate access.

Phases *2)* and *3)* constitute by far the most onerous parts of the application, in terms of execution time. Both of them are suitable for being revisited in the context of loop parallelism, since their kernels can be run concurrently on multiple processors with no data dependencies involved. These phases have been thoroughly explained in Chapter 4, and they basically do not change: when offloading the graph construction or the normalisation phase to C++, the very same logic is used and the `ParallelFor` skeleton permits to speed up both phases in a seamless way. Data transfer overhead is negligible, in comparison with the computationally intensive tasks that take place: it consists of moving a pointer to an object, plus a thin protection layer that prevents R garbage collector to delete variables in use [52].

Phase *4)* encompasses essential features that Nucha*R*t ought to provide, in order to fulfil the requirements of a useful tool for genomic data interpretation.

With a graph-based representation we can apply network analysis over the resulting graph and highlight the "importance" of the genes, by evaluating the interactions that occur among local entities. Other topological features, such as degree distribution, can help to prioritize targets of further studies or lead to a more local, in-deep analysis of specific chromosome locations [162].

Finally, visualisation is crucial for a tool that aims at facilitating a better interpretation of genomic data. NuChart-II supplies both tabular output and graphical visualization. Concerning the latter, *iGraph* and *GraphViz* are used as plotting engines, but while these tools perform nicely with small-to-medium sized graphs, they cannot provide useful representation of huge graphs. On the R side there are several graphic libraries — *MuxViz* or *networkD3* just to cite a few — that facilitate the interactive visualization and exploration of complex networks. With Nucha*R*t we can seamlessly exploit these libraries to create navigable and interactive maps of the chromosome. At the moment *networkD3* has been tested to plot our neighbourhood graph: *networkD3* produces interactive graphs that can be visualised through a browser, so that the JavaScript language it embeds can be interpreted. Despite good looking, it does not offer many possibilities of graph customisation and it still remains impractical for our purpose. Nevertheless, hacking the JavaScript source code might lead to better results.

A last resort for the visualisation problem is to decouple visualisation from Nucha*R*t, and make use of external applications purposely designed for interactive visualisation of networks. One such application is *Gephi* [22], that permits to interact with the representation, manipulate structures, shapes and colors to reveal hidden properties. Gephi can import graphs from a number of standard graph file formats, including Graphviz's `.dot` files, but it does not provide the same support for all file formats. In order to get the most out of Gephi, the preferred graph formats are XML-structured `GEXF` and `GraphML`. Nucha*R*t can write a resulting graph in one of these formats, so that the user can easily browse the result of Hi-C data analysis through Gephi interface.

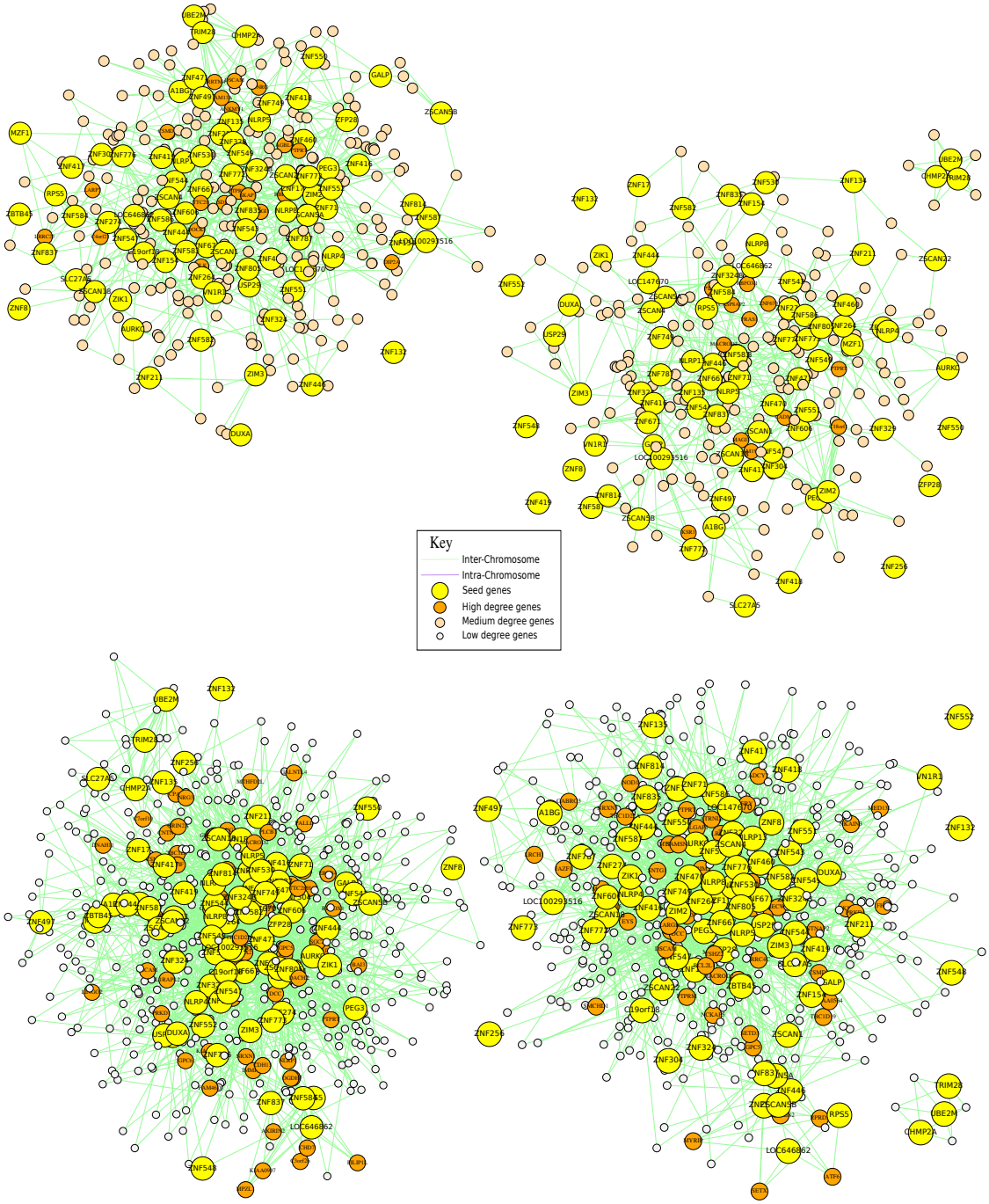


Figure 5.2: *Neighbourhood graphs of the KRAB cluster of genes in four different runs from the Hi-C experiments of Dixon et al.*

5.3 Discussion

The novel package benefits of the combined use of parallel programming techniques, provided by the C++ engine, and the flexibility of the R environment. Nucha*R*t maintains the same performance and scalability showed in NuChart-II (as discussed in Chapter 4), and can perform genome-wide analysis of Hi-C data with reduced memory footprint. Both the graph construction phase and the normalisation maintain loop parallelism benefits from employing FastFlow's `ParallelFor` skeleton: while the application is now based on the R environment, computationally intensive tasks can be offloaded to C++. Moreover, with the R environment the five steps listed in Section 5.1.1 become loose but totally compatible modules, and could be either executed in order, or as services that permit to accomplish a specific task on existing data. Also, with the wide rang of R's statistical and visualisation libraries at hand, each step can the starting point of further analysis. The visualisation step has only been partially addressed, and deserves a further analysis.

Results of each module are made globally available in form of `DataTable`s, optimised extensions of `DataFrame`s, and can be easily queried and inspected. The graph can be plotted and results can be visualised and browsed. Eventually, one can draw from the huge R's libraries basket the one that suits her need, and conduct advanced analysis over the resulting data. For instance, we also tested the *ERGM* package [69] that permits to understand the processes of network structure emergence and tie formation: the Exponential-family Random Graph Models package provides an integrated set of tools for creating an estimator of the network through a stochastic modelling approach. In particular, ERGM functions are able to extrapolate salient characteristics of a network by implementing a maximum likelihood estimator that helps to quantify how much a given feature mapped on the graph influences tie formations on the graph. For example, we evaluated the probabilities that edges are a function of a specific genomic feature (see Table 5.1).

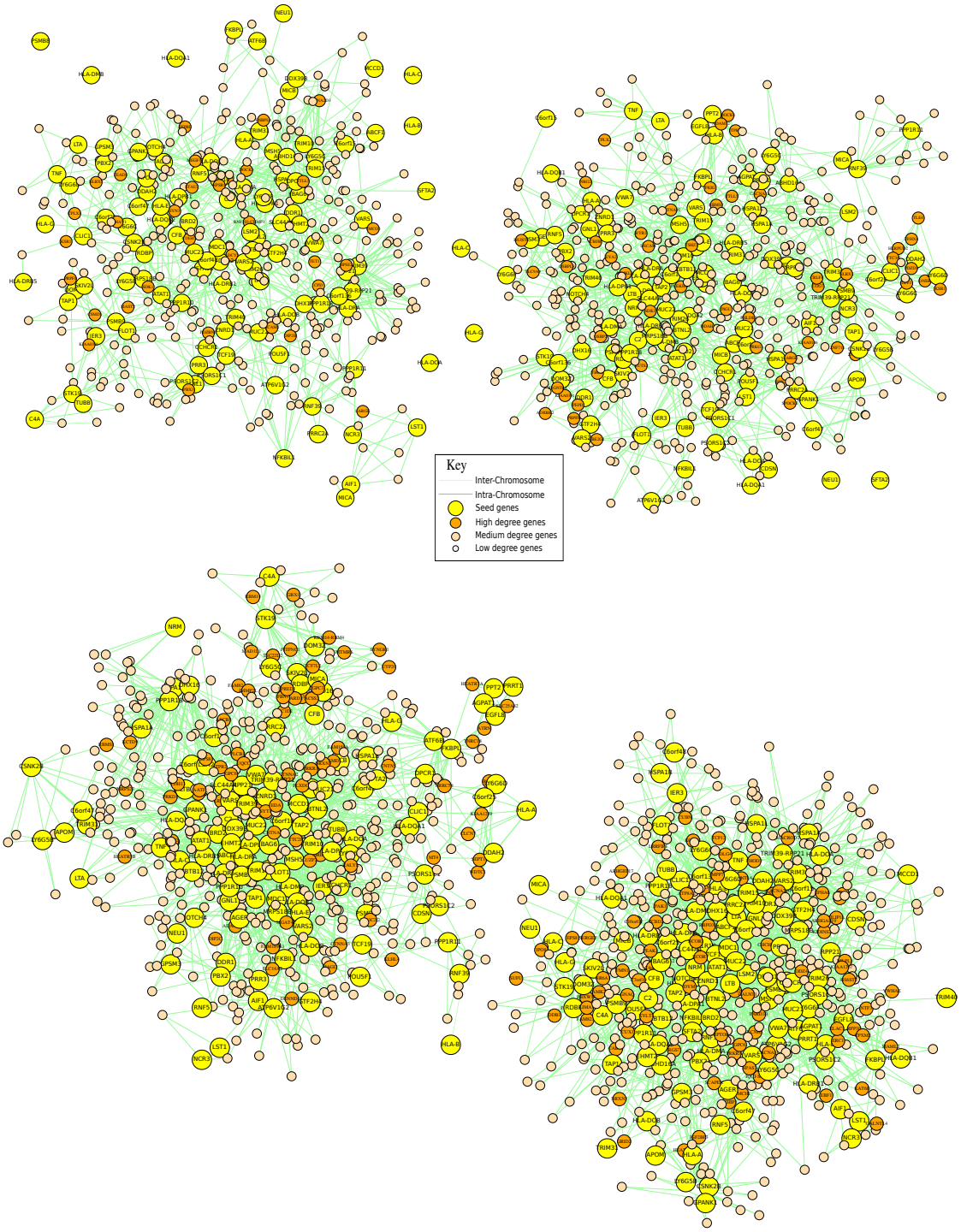


Figure 5.3: *Neighbourhood graphs of the HLA cluster of genes in four different runs from the Hi-C experiments of Dixon et al.*

5.3.1 Experiments

The study of the interactions of the actor genes with the environment is of critical importance for understanding the entire system. By using the modelling functions of the package we can statistically characterize the distribution of the edges in relation to the characteristics of the nodes that represent mapped multi-omics features. In order to test the possibilities of Nucha*R*t in terms of statistical inference on the graph, we performed the analysis of the clusters of genes Human Leukocyte Antigen (HLA, Figure 5.3) and Kruppel-Associated Box (KRAB, Figure 5.2) in the context of four Dixon experiments (SRA:SRR400260, SRA:SRR400261, SRA:SRR400266, SRA:SRR400267) [48], to verify the correlation of the edges distribution in relation to some genomic features (hypersensitive sites, CTCF binding sites, isochores, RSSs).

The first analysed locus is located in cytoband chr19.q13.12 and concerns the clusters of Kruppel-type zinc finger genes, related to the KRAB, which are peculiar for their tandem organization. Zinc finger proteins are a family of transcription factors that regulate the gene expression, and most of these proteins are members of the KZNF family. There are 7 human-specific KZNFs and 10 KZNFs that have undergone pseudo-gene transformation specifically in the human lineage. 30 additional KZNFs have experienced human-specific sequence changes that are presumed to be of functional significance. Members of the KZNF family are often in regions of segmental duplications, and multiple KZNFs have undergone human-specific duplications and inversions. In Figure 5.2, top panel drawings concern sequencing runs from the hESC cell line, obtained from Dixon et al experiments [48]; bottom panel drawings in the same figure concern sequencing runs from the IMR90 cell line according to Dixon et al experiments [48]. Yellow nodes are the genes given as input to the algorithm — in this case, all the genes belonging to the clusters under analysis — while output genes are differentially represented according to their importance (in terms of node degree).

The second analysed gene cluster concerns the human leukocyte antigen (HLA) system, which is the name of the locus containing the genes that encode

for major histocompatibility complex (MHC) in humans. It belongs to a superlocus that contains a large number of genes related to the immune system function in humans. The HLA group of genes resides on cytoband chr6.p31.21 and encodes for cell-surface antigen-presenting proteins, which have many different functions. The HLA genes are the human version of the MHC genes that are found in most vertebrates (and thus are the most studied of the MHC genes). The major HLA antigens are essential elements for the immune function. In Figure 5.3, top panel drawings concern sequencing runs from the hESC cell line while bottom panel drawings in the same figure concern sequencing runs from the IMR90 cell line [48]. Again, yellow nodes are the genes given as input to the algorithm, while output genes are differentially represented according to their importance (in terms of node degree).

The correlation between the presence of CTCF binding sites and edges was predictable, since linking gene-regulatory elements are demanded to keep different regions of the genome close to each other, but it is very interesting to quantify this association (see Table 5.1). On the other hand, regions with isochores seem less involved in long-range interactions (estimate < 0), which can be quite surprising considering that these portions of the genome are considered gene-rich. The correlation between cryptic RSS sites and edges is more pronounced in the HLA cluster, in comparison to the KRAB cluster, probably due to a more consistent presence of this kind of sequences in genes related to the immune system. Finally, the correlation between hypersensitive sites (super sensitivity to cleavage by DNase) and edges, although positive, is poor, probably because the accessibility of these regions are impaired by a large number of long-range interactions.

The network estimators are all computed using 100 iterations of the ERGM's stochastic modelling algorithm, which provided a good statistical significance for all the experiments ($p < 0.01$). It is very interesting to see the high reproducibility of results in the four sequencing runs, more reproducible than the simple correlation between the graphs. As we can see, there is a high correlation between the presence of specific genomic features, such as CTCF binding sites and cryptic RSSs, and the probability of presence of an edge. DNse sensitivity sites are weakly correlated with the presence of an edge, while isochores

	KRAB		HLA	
	Estimate	**Std. Error**	**Estimate**	**Std. Error**
SRA:SRR400260				
edges + nodecov(“dnase”)	0.2867	0.08451	0.1711	0.07961
edges + nodecov(“ctcf”)	0.6531	0.01157	0.5545	0.01253
edges + nodecov(“rss”)	0.5804	0.06176	0.6304	0.08196
edges + nodecov(“iso”)	-1.047	0.09269	-0.9406	0.09156
SRA:SRR400261				
edges + nodecov(“dnase”)	0.2042	0.07932	0.1706	0.07822
edges + nodecov(“ctcf”)	0.6629	0.04158	0.5687	0.02005
edges + nodecov(“rss”)	0.5378	0.03566	0.6319	0.03776
edges + nodecov(“iso”)	-1.015	0.09566	-0.93035	0.08969
SRA:SRR400266				
edges + nodecov(“dnase”)	0.2042	0.07932	0.1706	0.07822
edges + nodecov(“ctcf”)	0.6629	0.04158	0.5687	0.02005
edges + nodecov(“rss”)	0.5378	0.03566	0.6319	0.03776
edges + nodecov(“iso”)	-1.015	0.09566	-0.93035	0.08969
SRA:SRR400267				
edges + nodecov(“dnase”)	0.2042	0.07932	0.1706	0.07822
edges + nodecov(“ctcf”)	0.6629	0.04158	0.5687	0.02005
edges + nodecov(“rss”)	0.5378	0.03566	0.6319	0.03776
edges + nodecov(“iso”)	-1.015	0.09566	-0.93035	0.08969

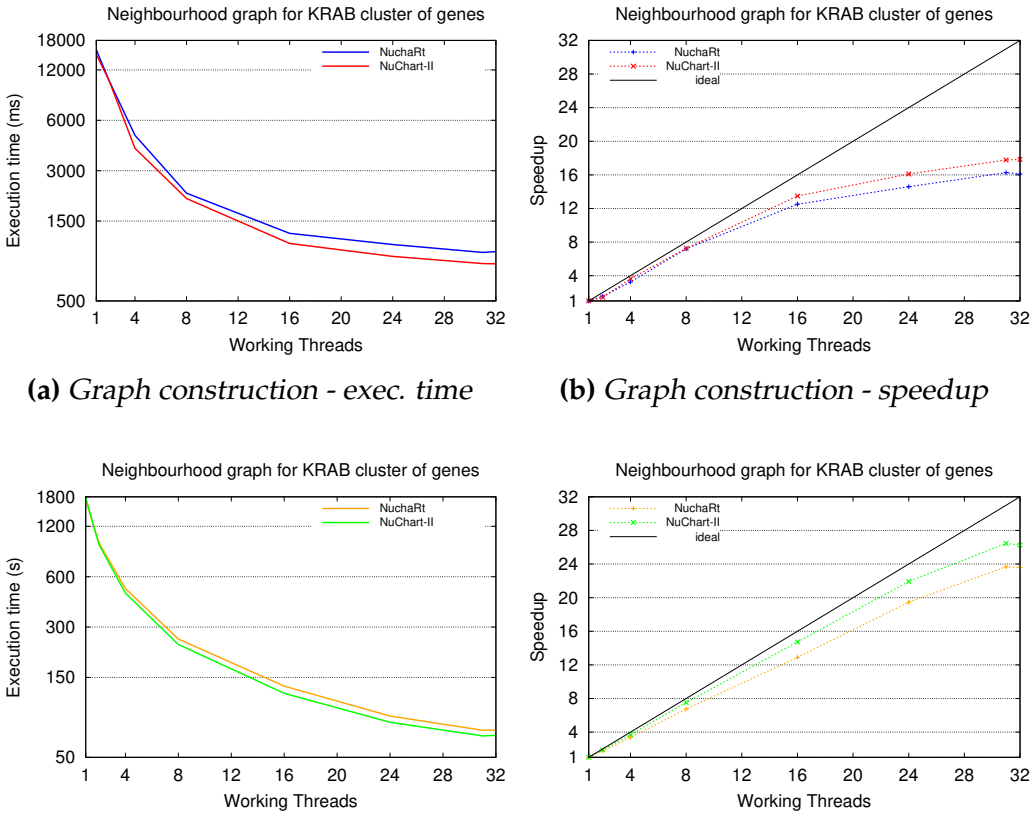
Table 5.1: *Analysis of the impact of the CTCF binding sites, isochores, cryptic RSSs, and DNase hypersensitive sites on the edge distribution of the KRAB cluster of genes and of the HLA cluster of genes*

are strongly anti-correlated with the presence of an edge.

5.3.2 Performance

comparison between the pure C++ application and the combined R with C++ package reports substantially similar behaviours: the graph construction execution is strongly affected by datasets size and resolution, that determine the “search space” for the BFS-like graph construction and the overall memory load. Reducing the working set ameliorates execution times and overall scalability with NuChart-II, and clearly helps in obtaining good performance when offloading the graph construction from R to C++.

Figure 5.4 compares execution time (left) and speedup (right) in the two approaches: Figures 5.5a and 5.5b show the performance for constructing a graph at level 1 starting from the KRAB cluster of genes using Dixon’s SRR400266 experiment as Hi-C dataset. Despite similar timings and scalability, Nucha*R*t has slightly worse performance and shows a higher execution time. Figures 5.5c and 5.5d show a comparison of the performance during normalisation phase



(a) *Graph construction - exec. time*

(b) *Graph construction - speedup*

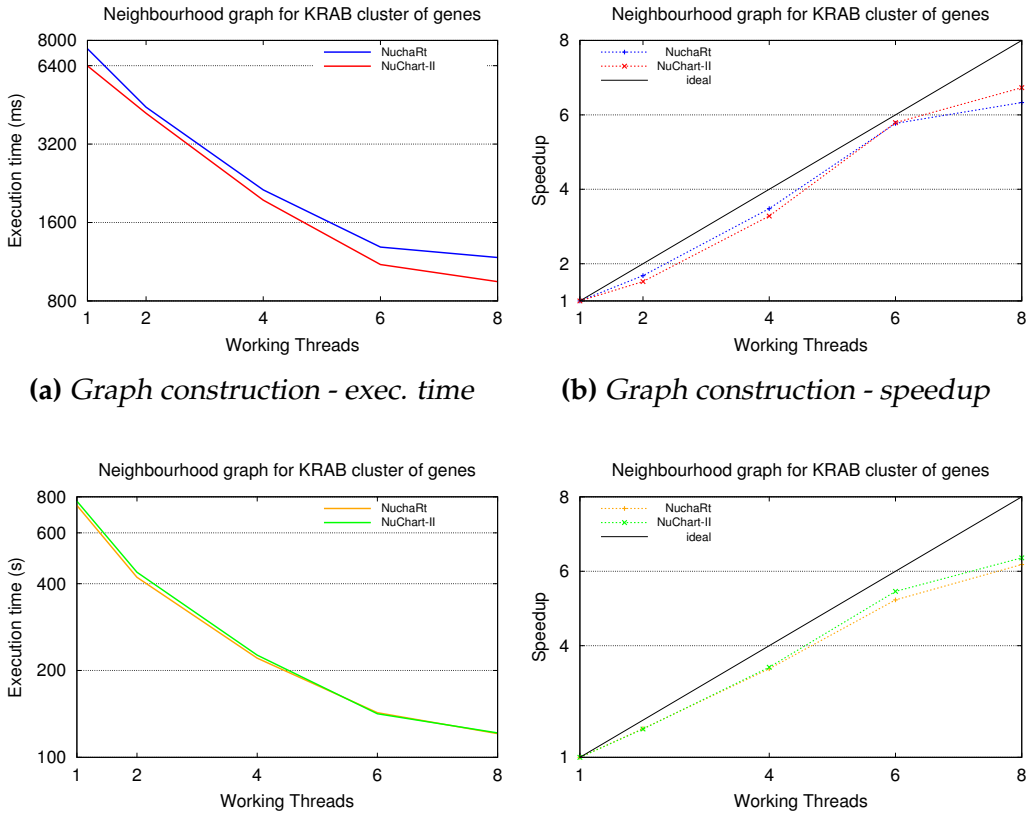
(c) *Normalisation - exec. time*

(d) *Normalisation - speedup*

Figure 5.4: *Comparison between NuChart-II and NuchaRt during the graph construction and normalisation on a NUMA machine. Graphs built on average running times, from 4 runs with very same configurations.*

with NuChart-II and Nucha*R*t: again both implementations yield similar results, both approaching a quasi-linear scalability, even though Nucha*R*t's execution time is slightly higher with respect to NuChart-II's. This is likely due to the worsening of memory access time when offloading computation to C++: while the multi-threaded C++ application is running, the R environment is kept alive. R stores additional information, beside the data itself, for each object created: when this small overhead is combined to the lazy memory reclaim policy adopted by R's garbage collector, and to the massive size of the dataset used for neighbourhood graph construction, resident memory consumption remains high at run-time, thus affecting memory access times and overall performance.

Our experiments where conducted on an Intel NUMA system, equipped



(a) *Graph construction - exec. time* **(b)** *Graph construction - speedup*

(c) *Normalisation - exec. time* **(d)** *Normalisation - speedup*

Figure 5.5: *Comparison between NuChart-II and NuchaRt during the graph construction and normalisation on a UMA, single socket machine. Graphs built on average running times, from 4 runs with very same configurations.*

with 4 eight-cores E7-4820 Nehalem running at 2.0GHz, with 18MB L3 cache and 64 GB of main memory. The Nehalem processor has Hyper-Threading capability with 2 contexts per core, but we decided not to use it and stick to the number of physical cores: the heavy memory usage would dramatically damage performance, and likely increase chances of false-sharing among threads in the same context that share L2 cache. With this machine the cache-coherence mechanism plays an important role on this performance degradation, where cache misses are likely frequent and cache line updates occur frequently.

Performance differences seem to flatten when the same applications are executed on a different machine (Figure 5.5): we also conducted experiments on

a workstation equipped with a single eight-cores Intel Xeon CPU E5-2650 running at 2.60GHz. This machine features 20MB of L3 cache with 64 GB of main memory. The SandyBridge processor also has Hyper-Threading capability allowing 2 contexts per core. Here as well we decided to not run more than 8 threads, so that the second context is not used an only physical cores are employed during computation. In this case the gap between the two solutions is reduced, though the total execution time is higher due to the limited degree of parallelism that can be achieved because of the reduced number of available cores.

5.3.3 Graph drawing

Graphs constitute a powerful and expressive mean to describe entities and their relationships, because they give an immediate, visual representation of both natural and man-made complex systems. Nonetheless, plain graph drawing might turn out to be useless with huge dense graphs, because it would result in an obscure bundle of edges and nodes (see Figure 5.6). In this case, network analysis and data mining techniques come in handy, providing methodologies to extrapolate knowledge from the network and to examine structural properties of the graph.

Starting from the assumption that the vertex degree is one of the most fundamental properties of the structure of a graph [104], in our first attempt to ameliorate the visualisation we classify nodes according to nodes degree and degree distribution: we set a threshold for nodes degree below which a node is dropped (together with the edge linking to it) and not plotted. In our context, this naïve approach helps to better depict the neighbourhood of a gene (or an ensemble of genes), drawing attention to those highly connected entities that are likely to play an important role in the gene regulation mechanism. Edges score is not necessarily taken into account with this first classification, but it can be factored in by setting a specific threshold for the edges too: recall that the score of an edge qualifies the reliability of the gene-gene contact identified by the Hi-C experiment. Edges with score equal to or above the threshold are

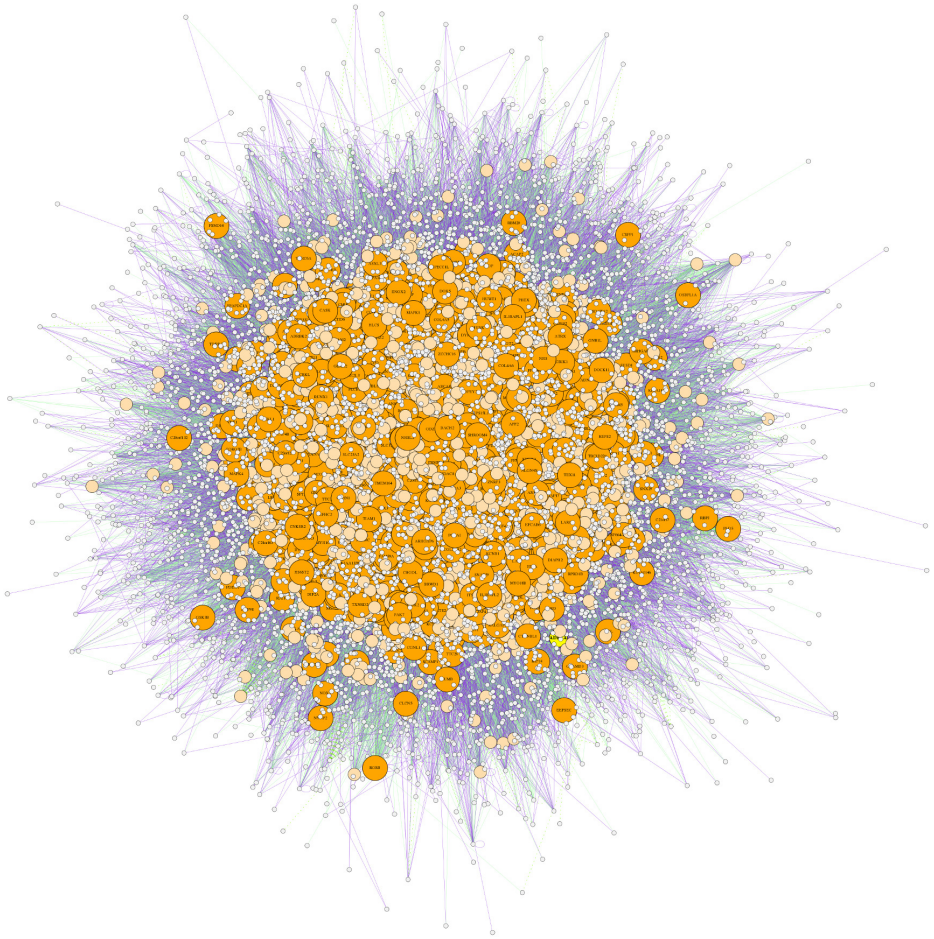


Figure 5.6: *Genome-wide graph built from human gene ADH1A, using LiebermanAiden's SRR027957 experiment. The graph reached 18450 genes (nodes) and has 588635 edges*

kept on the graph together with the nodes they link, no matter what degree the nodes have.

This naïve approach is useful to extract a representative instance of the immediate neighbourhood of a gene, and could still perform well with a graph at with less than 2000 nodes, but it soon becomes impractical to visualise a graph of the whole genome (Figure 5.6): in our vision, the network has to be browsed just like a map, so that the information contained within the genes and the edges can be accessed by simply interacting with them. This perspective requires a mechanism that changes the details of the graph as we zoom in or out without

losing consistency. Much like in a web-based cartographic system, zooming out of the graph keeps only few but relevant entities visible on the map (highest degree nodes or densest subgraphs), while details increase as we zoom in. This is one outstanding challenge we are currently dealing with.

We are investigating a number of solutions to address the visualisation problem, which will be better presented and discussed in Section 7.1.

5.4 Concluding remarks

Coupling the broad R modelling capabilities and the high performance achievable with the parallel C++ implementation of NuChart-II, yields a solid response to the demand for software tools that help scientists in drawing more concrete biological knowledge. The graph-based approach fosters a tight coupling of topological observations to biological knowledge, which is likely to bring remarkable biological insights to the whole research community. From a computational point of view, the ever-increasing amount of information generated by novel Bioinformatics techniques require proper solutions that permit the full exploitation of the computing power offered by modern computing systems, together with advanced tools for an efficient analysis and interpretation of genomic data.

Despite the results achieved in terms of performance and usability, some problems remain partially unsolved, and are open to further investigations. Among them, our main concern is the visualisation of multi-omic graphs, which we believe is an essential feature for a usable tool aimed at facilitating genomic data analysis and interpretation, and that remains an open problem.

Chapter 6

A cloud solution for multi-*omics* data integration

We have already seen that the recent advances in molecular biology and bioinformatics techniques have brought to an explosion of the information about the spatial organisation of the DNA inside the nucleus. In particular, 3C-based experiments are revealing the chromatin structure in the nucleus of the cell, for many different cell types and cell states. This information can be used to re-interpret *omics* data (in particular RNA-Seq and ChIP-Seq outputs), which can greatly benefit from analysis relying on 3D maps of the chromatin conformation in the nucleus, instead of relying on the bare linear sequence of the genome. Public on-line repositories, such as NCBI and EBI, are full of data that can be reconsidered as soon as novel genomic data becomes available.

Each of these experiments potentially opens new perspectives for re-analyse data in a three-dimensional view, but their combined interpretation requires a considerable computational support. The Cloud paradigm, which allocates resources *on-demand*, seems the most convenient way to support the development of new solutions that foster an unsupervised analysis of biological data. Here we propose a cloud-based task farm that orchestrates a data analysis work-flow, leading to the integration and the interpretation of multi-omic features on a nuclear map, capable of representing the effective disposition of genes in a graph-based representation.

6.1 Motivation: a flood of data

In Chapter 3 we discussed novel techniques for DNA sequencing and chromosome conformation analysis: DNA sequencing is driving unprecedented discoveries in the life sciences since the emergence of next-generation sequencing (NGS) technologies, and the research is moving from a hypothesis-driven to a data-driven approach. With the advent of massively parallel DNA sequencing, the rate at which the human genome is studied is limited by the computational tools required to analyse these data, more than by sequence generation. Indeed, every experimentally analysed genomic pattern requires an extensive computational process to transform hundreds of gigabytes of raw output into aligned sequences: the sequencing industry has been demanding increasingly larger compute and storage resources to keep up.

The development of new technologies and research models that would help to address NGS computational needs is a hard path to go, because these fields are literally "caught in a flood of data" and are now facing many of the scale-out issues that high-performance computing (HPC) has been addressing for years. Data must be managed, analysed, integrated and archived. These steps pose substantial requirements in speed (in terms of execution time), application scalability and data representation. Orchestrating applications to fulfil these requirements is a complex and delicate task.

In previous chapters, our graph-based approach to represent genomic data has been explained: graphs are used to provide a better description of the social behaviour of genes inside the nucleus, and can be very useful to map and re-interpret omics data. A characterization of different spatially-associated domains creates a topological description of the genome folding in the nucleus, in relation to different cells and cell states. These maps will be the ground for the integration and analysis of omics information, which will be annotated as "points of interest" on the chromatin 3D structure. We will test our solution to characterize results from a gene therapy clinical trial using public available Hi-C and epigenetic data. Achieved results can boost the development of a new generation of 3D genome browsers, which will take into account the spatial distribution of the genome in the nucleus, facilitating the identification of

spatio-functional patterns (co-localization/regulation/expression).

Considering the computational workload that comes from DNA sequencing, the critical performance parameters that drive effective system architectures for the given workloads, and the algorithms and applications being used, the computational demands of genomic data analysis and interpretation can be satisfied by using lab-scale and enterprise-oriented technologies. Cloud computing, and its various declinations (*DaaS, SaaS, PaaS, IaaS*), deliver computational power and storage as dynamically allocated virtual resources via the Internet, relieving scientists from the daunting expense of establishing and maintaining complex computational infrastructures for data processing.

Cloud computing turns out to be also very useful for the kind of analysis we are interested on, considering that the *on-demand* paradigm properly addresses the need of re-mapping available expressions and epigenetics data on new Hi-C experiments. These experiments permit to re-analyse multi-omic data available in public repositories (such as NCBI, and EBI) relying on a three-dimensional genomic view. As new information about the chromatin conformation in different cell types and in relation to different cell states becomes available, grows the need for novel pipelines of data analysis and integration designed to efficiently and easily handling these tasks.

6.2 A bit of background

The high demand for low-cost genomic data has driven the development of high-throughput technologies that parallelise the sequencing process. Nowadays, the advent of novel rapid sequencing methods has greatly accelerated biological and medical research, generating millions of biological sequences (reads) in each experiment. The cutting-edge idea was to amplify each piece of genome in a local fashion, using nano-wells or similar approaches, to automatically analyse the signal corresponding to each piece of genome in parallel, generating millions of sequencing reads. Operatively, the addition of one (or more) nucleotide(s) at each sequencing step results in a local reaction that generates a signal that is recorded by the instrument. If the signal is mediated

through light, a CCD camera is used for the identification of wells, while in case of PH variation a semiconductor technology is used to capture the signal.

Although sequencing mainly refers to DNA, NGS techniques are used in many related genomic researches which can be re-conducted to this case, such as transcriptome profiling (RNA-Seq), epigenome characterization (ChIP-Seq), and for the analysis of the chromosome conformation inside the nucleus. Concerning RNA-Seq, it is used to characterize gene expression in cells, and study which genes are expressed in what tissues, and at which levels. ChIP-Seq helps determining how proteins interact with DNA to regulate gene expression, and is essential for understanding many biological processes and disease states.

If the genome is available for the specie in analysis, the first computational effort is usually the alignment against the reference sequences. This is useful to highlight the genomic regions involved in expression and regulation. Although alignments are computed with very sophisticate algorithms, usually employing the Burrow-Wheeler Transformation [30], NGS data analysis is notoriously a time-consuming task. Since dealing with sequencing experiments is one of the most resource-demanding applications in Bioinformatics, it is very common to use parallel, distributed and cloud infrastructures for this kind of analysis.

In Chapter 3 we have given a description of next-generation sequencing and related techniques: we refer to those pages (and references therein) for a more detailed explanation.

6.2.1 Cloud for Bioinformatics

High Performance Computing technologies and Big Data paradigms are very important in Bioinformatics, since they allow to accelerate biological and medical analysis, moving genomic data analysis from laboratories to healthcare system. With the amount of data growing continuously, it is becoming increasingly daunting for small laboratories, or even large institutions to establish and maintain computational infrastructures for data processing.

Accordingly, Bioinformatics is experiencing a new leap-forward from in-house computing infrastructure to utility-supplied, on-demand computing delivered over the Internet, that permits to handle the vast quantities of biological data generated by high-throughput experimental technologies [133]. At the core of this computational paradigm are virtualisation techniques, which enable a larger resource usage by sharing a given hardware among several users, thus reducing the required amount of instances of that particular device. As a result, virtualisation is being increasingly adopted in data centres.

Data are both a blessing and a curse in Bioinformatics: for this reason it can seriously take advantage from the cloud paradigm. However, cloud computing is a broad term that describes a broad range of services built on top of one another (see Section 2.5). The Cloud holds great promises in effectively addressing data storage and analysis problems in Bioinformatics: a typical such analysis often involves downloading data from public sites, installing software tools locally, and running analyses on in-house computer resources. By placing data and software into the cloud and delivering them as services, data and software can be seamlessly integrated into the cloud so as to achieve adequate data storage and analysis.

In the past several years, efforts have been made to develop cloud-scale tools, including sequence mapping [117, 134], expression analysis [165], peak caller for ChIP-Seq data [55], and various cloud-based applications for NGS data analysis (BGI Cloud[31] and EasyGenomics[32]). These names can all be seen as examples of the *SaaS* model.

To the best of our knowledge, two remarkable *PaaS* platforms in Bioinformatics are Eoulsan[33], which is a cloud-based platform for high-throughput sequencing analyses, and Galaxy Cloud[34], which is an open, web-based platform for data intensive biomedical research.

Amazon EC2 represents an example of an *IaaS* model [61], and it offers a variety of VM images provided with a good variety of Bioinformatics tools. Other important examples are Cloud BioLinux [92] and CloVR [11]. The former is

[31] `https://www.bgionline.com/`
[32] `http://www.easygenomics.com/`
[33] `http://www.tools.genomique.biologie.ens.fr/eoulsan/`
[34] `http://galaxyproject.org/`

a publicly accessible virtual machine for high performance bioinformatics computing. The latter, instead, is a portable virtual machine for automated sequence analysis.

It is also worth to mention that Amazon Web Services (AWS) provides a centralized repository of public datasets, including archives of GenBank, Ensembl, Model Organism Encyclopedia of DNA Elements, Influenza Virus, etc. As a matter of fact, AWS contains multiple public datasets for a variety of scientific fields, from biology to astronomy, chemistry, etc.[35]. These datasets are delivered as services (*DaaS*) and can be seamlessly integrated into cloud-based applications [61].

6.2.2 The problem of data integration

The problem of data integration can be approached using semantic technologies and ontologies, which are essential in many domains for capturing knowledge as a hierarchy of concepts and corresponding relationships. However, these methods assume as a prerequisite an independent sampling of features. This assumption is clearly not satisfied when considering long-range chromatin interactions, since they associate regions that are known to be functionally correlated.

On the other hand, high-throughput 3C datasets are a natural ground for integrating many different types of data, since they describe the topological organization of the genome in the nucleus, for example, by enabling the interpretation of features that are shared by spatially co-localized genes. Studying point-wise properties such as gene expression, methylation, histone modification and mutations, and relational information such as co-expression and co-regulation, can benefit from the spatial context provided by Hi-C based distance matrices (derived from contact frequencies). Recent studies that have focused on comparing relational omic information with contact frequency have pointed to a high correlation of the above properties with the spatial organisation of the chromatin, encouraging further research of the 3D genome architecture and epigenetics [108].

[35] `http://aws.amazon.com/publicdatasets`

With these premises, we studied a solution for mapping point-wise information directly on the topological map of the chromatin in the nucleus, represented by the graph. This allows us to apply spatial statistics on the nucleus and highlight patterns and correlations between epigenetic features and spatial organization, that could not have been detected by considering only the bare proximity on the linear sequence. This in turn can result in insights into how spatial and chemical constraints are acting together to regulate molecular function.

6.3 Methods: a cloud-based task *farm* approach

Considering the innovation that high-throughput technologies are introducing to the analysis of omics data, and the importance of the spatial conformation of the genome in the nucleus, we propose a custom work-flow for data integration that relies on cloud computing: a cloud architecture with a scalable high-performance infrastructure where software tools for data processing are provided *as services*.

In particular, we want a solution that permits to re-interpret RNA-Seq and ChIP-Seq data every time a novel experiment of Chromosome Conformation Capture is available to researchers. The idea is to create a system that revolves around the graph representation of Hi-C data, upon which transcriptomics and epigenetics experiments can be mapped and statistically analysed taking into account the spatial component.

6.3.1 Three pipelines

Raw data from genomic experiments are downloaded from public repositories and analysed using well-known pipelines. The main repository for this kind of analysis is the Sequence Read Archive (SRA) of the National Centre for Biotechnology Information (NCBI), which makes biological sequence data available to the research community, so as to enhance reproducibility and allow for new discoveries by comparing datasets. In this work, we also use this repository as main reference for RNA-Seq, ChIP-Seq and Hi-C experiments. Datasets are

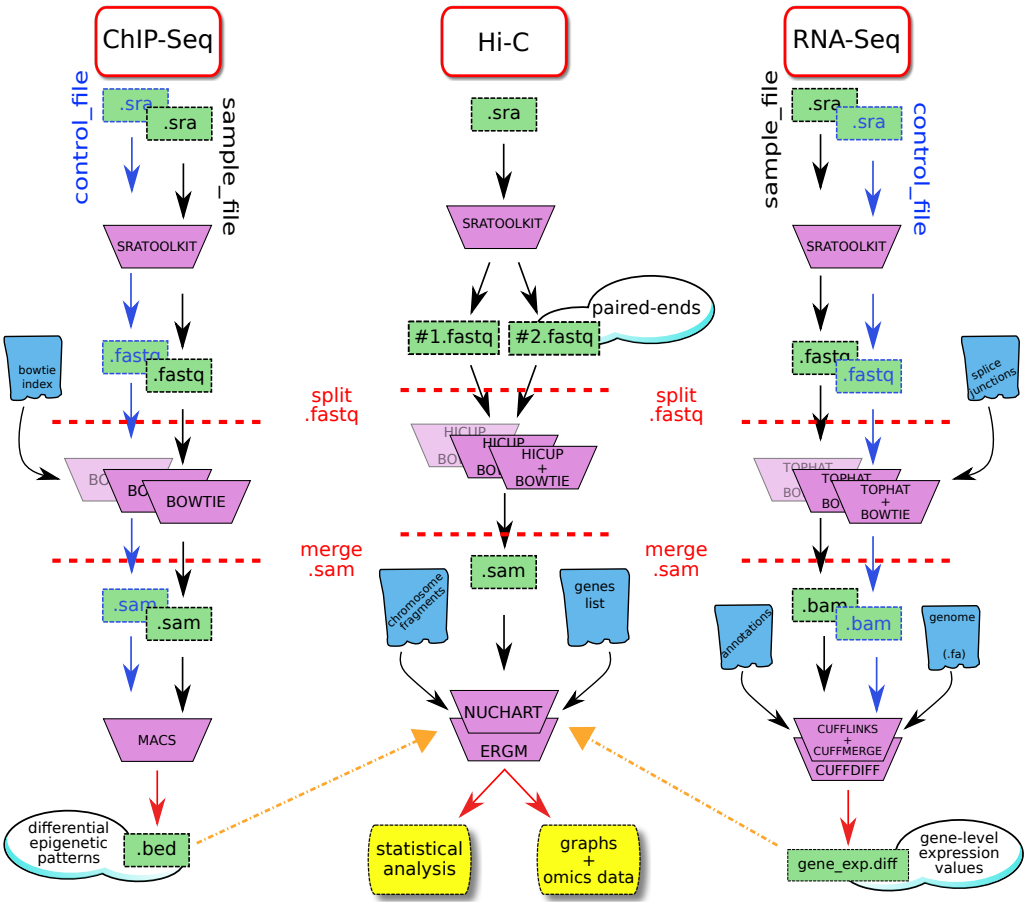


Figure 6.1: *Raw data resulting from genome sequencing are processed through the steps of data analysis pipelines before they can be integrated and statistically interpreted. Red dashed lines indicate a further dataset partitioning, where* `.fastq` *files are split and processed by many aligner services simultaneously, then partial aligned* `.sam` *files are merged together to form a definitive output of aligned sequences (see figure 6.2)*

available in the `.sra` format and can be easily managed using *SRAtoolkit* [137], a suite of tools that allow to programmatically access data housed within `.sra` files and convert them to a number of standard formats, like `.fa` and `.fastq`.

Once data have been retrieved, they must be analysed before they can be integrated and statistically interpreted. Two widely used analysis pipelines for RNA-Seq and ChIP-Seq data exist, relying on *Tophat* [151] and *Cufflinks* [152] the former, and on *Bowtie* [97] and *MACS* [166] the latter. For what it concerns Hi-C data processing, we already discussed our approach for the analysis and representation of such information in Chapters 4 and 5. Our method relies on

HiCup [19] for raw data processing and alignments, and on Nucha*R*t for mapping multi-omic information on the resulting neighbourhood graph and subsequent visualisation. Figure 6.1 illustrates the stages that compose the three pipelines: raw data resulting from genomic experiments (`.sra` files) are converted and aligned against reference genomes, and then processed using specific tools (MACS, Cufflinks, Nucha*R*t). ChIP-Seq and RNA-Seq pipelines expect the analysis to be conducted both on sample data and control data, so that significant differences can be detected.

Dashed lines in Figure 6.1 indicate a further dataset partitioning: `.fastq` files are split into multiple parts, as many parts as there are services available for data alignment (see Figure 6.2). Partial files are processed simultaneously and converted to `.sam` (or `.bam`) file. All partial results are then merged to form a single aligned file ready to be used in the subsequent steps: ChIP-Seq yields differential epigenetic patterns; RNA-Seq produces gene-level expression values; Hi-C reveals the chromosome spatial conformation. Once all data have been processed, we proceed by mapping them onto the neighbourhood graph produced by Nucha*R*t.

The definitive step aims at verifying if co-localization patterns can be found in transcription and/or epigenetic experiments. Although many different approaches can be used for this purpose, such as ontology annotation or machine learning analysis, in this work we use a statistical approach relying on the *Experimental Random Graph Model* (ERGM), which is very popular in the network analysis field [69]. This technique has the advantage of overcoming the limiting assumptions of independence and identical distribution of standard statistical

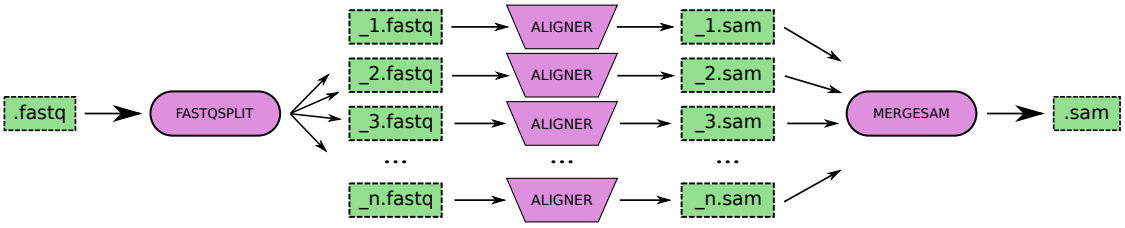


Figure 6.2: *`.fastq` files are split into several parts, according to the available services, and each piece is dispatched to a remote service. Outputs from each service are merged to form a definitive, aligned SAM (or BAM) file used during subsequent steps*

models, like linear regression does, which is clearly violated by network data since they are inherently relational.

Multi-threading is optionally supported by some of the tools listed above. For instance, searching for alignments is a highly parallel task, and obtainable speedup is significant (though affected by memory overhead). Bowtie allows to execute a parallel search, where threads will run on separate processors/cores and synchronize when parsing reads and outputting alignments. Consequently, all tools based on Bowtie (e.g., TopHat) permit to exploit multi-threading during the alignment phase. Among the tools in the Cufflinks suite, *Cuffmerge* and *Cuffdiff* walk through short-reads alignment steps, and they allow to specify the number of threads to be used when performing such operations.

HiCup itself is a pipeline encompassing six scripts executed one after another, that enable multiplexed sequences to be sorted and mapped to the genome, pairing Hi-C fragment ends and filtering for valid Hi-C interaction products. HiCup supports multi-threading, allowing simultaneous processing of multiple files.

Finally, Nucha*R*t has already been discussed in Chapter 5 and its features in exploiting multi-threading parallelism to speedup Hi-C data analysis have been presented.

ChIP-Seq analysis protocol

The goal of ChIP-Seq experiments is to identify Transcription Factor binding sites or epigenetic modifications able to control gene expression. The first step of this analysis includes an unspliced alignment for a small subset of raw reads. There is a number of competing tools for short read alignment, each with its own set of strengths, weaknesses, and caveats. As for the ChIP-Seq pipeline we align raw sequencing data using Bowtie, by far the most popular short-read aligner relying on the Burrow-Wheeler Transform. The alignment against the genome must be performed both for the experimental file and for the control file. This step is of utmost importance if we want to identify differentially enriched areas in the genome.

Once raw reads have been aligned, we use MACS for identifying transcription factor binding sites: MACS evaluates the significance of enriched ChIP regions and uses them to improve the spatial resolution of predicted binding sites, by combining the information of both sequencing tag position and orientation. As a non-interactive CLI tool, MACS takes inputs by setting proper command line parameters and no input is needed during running. It works on mapped reads from ChIP-Seq experiments, while the control data is optional. The minimum output of a MACS execution contains the called peaks and their summits. It also permits to generate wiggle format files that can be loaded into the Affymetrix *Integrated Genome Browser* (IGB) [118] or UCSC Genome Browser [154] to visually analyse the ChIP-Seq signal.

RNA-Seq analysis protocol

RNA-Seq studies the transcriptome of a group of cells, which typically means to check for the different isoforms of genes and to compare the gene expression between two different conditions in order to identify differentially expressed genes. This is done with a well-established protocol, often called the "Tuxedo pipeline".

The key point of this technique is the alignment of transcripts reads to a reference genome, where the alignment software also places spliced reads across *introns* and correctly determines *exon-intron* boundaries. Numerous tools perform short read alignment, so the choice of aligner should be carefully made, according to the analysis goals/requirements. Here we will use Tophat, a widely used ultra-fast aligner that performs spliced junctions alignments. Tophat works directly on reads from transcriptomic experiments, using the reference genome indexed according to the Burrow-Wheeler Transformation. Aligned fragments are stored in a `.bam` file (compressed binary version of the `.sam` format) called `accepted_hits.bam`. It also creates a `.bed` file named `junctions.bed`, that defines the data lines displayed in an annotation track and stores the coordinates of the spliced junctions present in the dataset, as these have been extracted from the spliced alignments.

Once the reads have been mapped against the reference genome, the reconstruction of the transcriptome is performed using Cufflinks. Cufflinks can do transcriptome assembly either *ab initio* or using a reference annotation, and also quantifies the isoform expression in Fragments Per Kilobase of exon per Million (FPKM). Cufflinks works on the `.bam` file produced by Tophat, using also the reference genome and the gene annotations. The reconstruction process yields 4 files that summarize the process while providing also the estimated gene-level expression values and the estimated isoform level expression values. Finally, the transcripts originated on the sample and on the control are merged in a unique file through the Cuffmerge tool, a script which comes in support of Cufflinks.

The last step of a typical RNA-Seq analysis is the differential expression evaluation. The Cufflinks package includes a program named Cuffdiff, which can be used to find significant changes in transcript expression, splicing, and promoter use. Cuffdiff models the variance in fragment counts across replicates as a function of the mean fragment count across replicates. The Cuffdiff step permits to compare different cell conditions — such as control and disease, wild-type and mutant, or various developmental stages — in order to highlight different RNA expression profiles. Cuffdiff is thus fed with gene annotations file, and yields a `.diff` file that contains a list of the most significant differentially expressed genes, sorted by *q-value* (corrected *p-value* for multiple tests).

Hi-C analysis protocol

Hi-C experiments describe the conformation of the DNA inside the nucleus identifying interactions between spatially adjacent DNA segments. As for the previously discussed pipelines, the first step is the alignment of reads against the reference genome. The particular structure of Hi-C datasets requires a wrapper to the aligner, in order to handle paired reads in terms of contact frequency. One of the most popular software for this kind of analysis is HiCup, which takes in input the bare FASTA *Quality* (`.fastq`) files and performs mapping and preliminary sequences filtering using Bowtie. Then for each read representing a digested fragment, HiCup observes the distance from the nearest restriction site

to verify if the distance is reliable (it should be less than a fixed threshold, otherwise the corresponding Hi-C contact is filtered out).

We already discussed Nucha*R*t, that exploits this graph-based approach for the representation of Hi-C data. The graph-based approach has proved to be a valuable way for the interpretation of genomic information by mean of complex, dynamical structures that organize items in an integrated way.

6.3.2 Integration and statistical analysis

With a graph-based representation, it is possible to apply network analysis over the resulting graph: topological measures capture graph's structure for nodes and edges and highlight the "importance" of the actors. Exploiting Nucha*R*t we want to integrate data resulting from RNA-Seq and ChIP-Seq pipelines into the Hi-C graph.

We use a method for the network analysis that relies on Exponential Random Graph Models [69], which proposes a way to understand the processes of network structure emergence and tie formation. These models work by measuring a limited set of known statistics from a given network, and then using the distribution of such parameters generates random networks. These random networks are then compared to the observed subject to assess the likelihood of the fit.

Operatively, the software generates a huge number of networks, selects the ones having characteristics similar to the graph under analysis (i.e. degree distribution, connected components, topological conformation), and tries iteratively to optimize the generation parameters until all the created graphs have characteristics similar to the processed one. This estimator is extremely useful, since it allows to create probability distributions by which some peculiarities of the graph can be extrapolated, concerning both its intrinsic topology and considering specific attributes of the nodes [1]. In particular, the package allows to compute simple statistics about the topology of the graph, such as the significance of the vertex clustering attitude (*triangle* terms), or the significance of the network tendency to create multiple paths between two vertices (*twopath* term).

On the other hand, we can choose more complex modelling functions, and exploit the multi-omic features mapped on our graphs: for example, the `nodecov` term analyses the nodal covariate, and will produce an estimate about the effect that a particular feature has on tie formation. The term `absdiff` will test the significance of having edges in relation to the absolute difference of a vertex property. We believe that the possibility of analysing data to infer structural-activity relationships in a network is of critical importance [130].

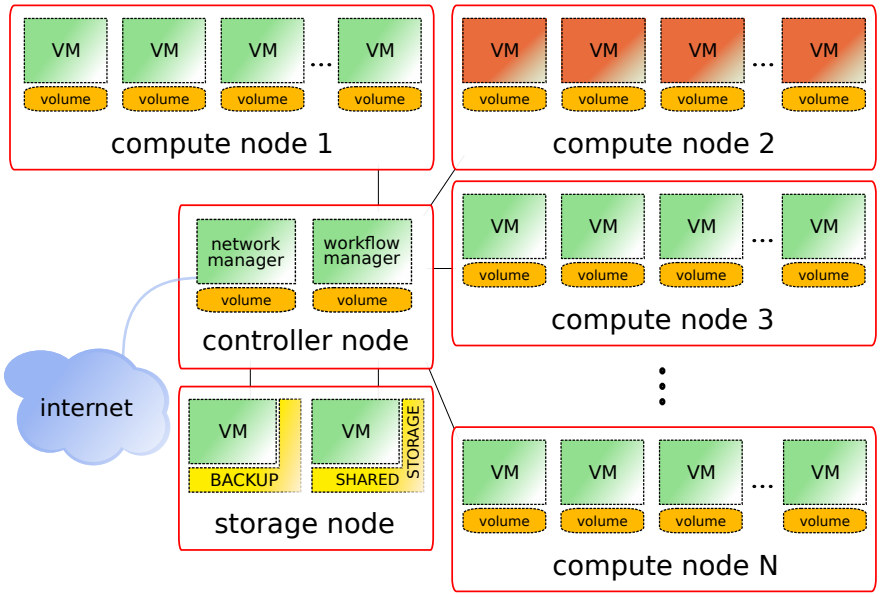
6.4 Cloud platform

Processing genomic data through the pipelines described in Section 6.3.1 appears manageable, but working on thousands of such data sets within a reasonable wall-clock time frame presents a challenge. Each single step of the pipelines could take up to few hours — depending on the size of input files and on computing capabilities available (see Section 6.5.2) — and produces results that can be combined together to obtain an enriched data interpretation. If we want our approach to scale genome-wide, we have to take into account that many genomes are quite large (the human genome is 3.3 billion base pairs long), thus data analysis represents a significant computational and data challenge. Moreover, as the resolution achievable through NGS experiments is increasing and the read-quality data is factored in, a dataset containing full DNA molecules easily reaches to hundreds of gigabytes.

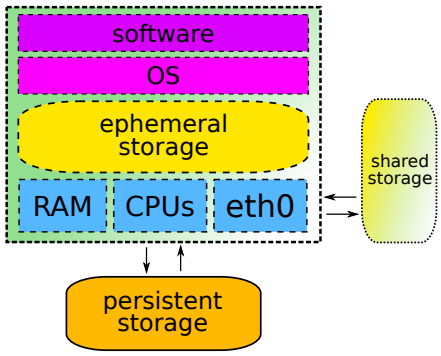
6.4.1 Set up and communication

In this context, the cloud paradigm represents an appealing solution that permits to obtain large amounts of computing capacity on-demand, with variable pricing. Upon the OpenStack cloud software we built our virtualised infrastructure: OpenStack[36] is a free and open-source cloud operating system,

[36] `https://www.openstack.org/`



(a) *Architecture of the cloud infrastructure*



(b) *A Virtual Machine instance*

Figure 6.3: *Each machine features a virtual hardware layout which includes CPUs, RAM and network interfaces, and an ephemeral storage. A persistent block storage (or volume) of 100 GB is attached to each VM. A Linux operating system runs on top of the virtual hardware, together with all software needed by data analysis pipelines*

that controls large pools of compute, storage, and networking resources, providing an IaaS remote environment for end users. OpenStack APIs are open-source Python clients, and can run on most existing operating systems, including Linux, Mac OS and Windows. A command-line interface enables to access the platform's API through easy-to-use commands that can be included in scripts to automate tasks. Internally, each command uses cURL command-line

tools, which embed RESTful APIs, and use the HTTP protocol. They include methods, URIs, media types, and response codes.

Each of our virtual machines is equipped with direct-attached *ephemeral* storage (i.e., it disappears when the virtual machine is terminated) and a secondary persistent storage where resources outlive the virtual instance (see Figure 6.3b). Persistent storage can be attached (i.e., mounted) to running instances, detached and attached again to a different virtual machine. Our virtual instances feature commodity multi-core architectures, with some VMs having a more powerful configuration (red instances in Figure 6.3a). All virtual instances are connected to each other trough a 100Mbps network and run a 64-bit Linux OS.

To fully leverage the cloud, there are some design aspects that affect architectural choices. Namely, a right combination of data storage, job orchestration and data exchange solutions would help to minimise processing costs. In our scenario, datasets are collected from on-line repositories prior to the start of the analysis. They are accessed in read-only mode, and must be available for all computing instances in order to be used throughout the steps of the pipelines. On the other hand, when a step has completed its execution, the resulting output file(s) will be used as input for a (possible) subsequent step. Consequently, novel produced files need to be accessible by the service that is going to process that data. This requirement clearly derives from the *software-as-a-service* scenario that we have designed: each step of the pipeline is accomplished by a service (trapezoid blocks in Figure 6.1), and each single virtual machine hosts all services need by all pipelines. A virtual machine can execute one service at a time, provided that all needed input is available for the service to run; a work-flow manager is responsible for orchestrating task scheduling.

While the cloud methodology would suggest data migration and replication as the favourite mean of shared storage, we decided that a distributed file system that "ties" together all compute instances is the best solution for our scenario: datasets are huge and need only to be accessed in read-only mode, thus a transferring appears to be unneeded and impractical. Every virtual instance performs temporary writes on its own local storage, where it has direct I/O access. Writes to the shared folder only happen when a service has completed

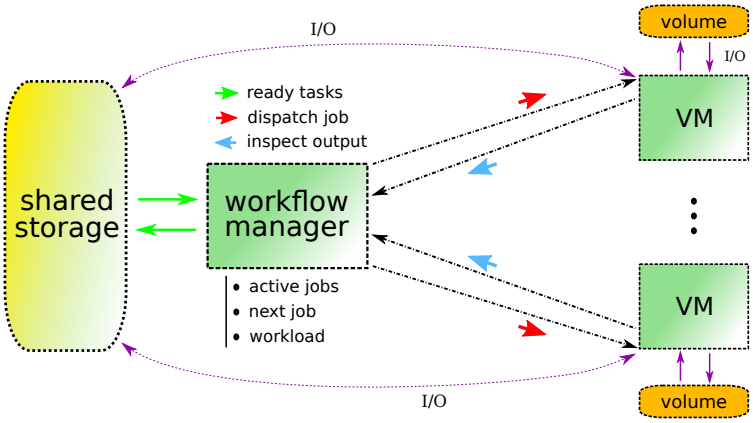


Figure 6.4: *Task scheduling*

its task. As for contention, each service does not interfere with others' operations, because each write to the shared storage only affects the files owned and produced by the service. Shared data is periodically backed up on a backup storage.

6.4.1.1 Task scheduling

The controller node in Figure 6.3a provides two services for managing work-flows and network operations: the network manager basically operates as a proxy for all data transfer operations, in and out of the cloud, acting as a gate-way to the internet while managing accesses to outputs results. A further improvement will exploit a powerful and scalable NoSQL database to store datasets and results: this is a step towards a multi-omic browser that permits to index, search and retrieve datasets and results from genomic data analysis.

The work-flow manager is aimed at organizing the steps of the pipelines into *tasks*, maintaining a list of active *jobs*[37] and a list of tasks ready to be scheduled, and finally dispatching jobs to worker machines while handling failures and uncertainties (see Figure 6.4). There are several, well established work-flow management systems (WMS) that have supported Bioinformatics and other scientific work-flows (*Pegasus*, *Taverna* and *Askalon*, among others) which provide

[37] In this chapter we will use the terms *task* and *job* interchangeably

several advanced features such as fault-tolerance, task clustering, site selection, resource provisioning, etc.

OpenStack includes a library named *TaskFlow*[38], that allows the creation of work-flows where task objects and functions are combined together. In this sense, a task is the smallest unit of work that can be executed, and it defines its desired input values and the expected outputs. A flow is a structure that links one or more tasks together in an ordered sequence. Upon these objects, TaskFlow guarantees a consistent, scalable and reliable work-flow execution, and includes an engine that permits to stop, resume and safely revert tasks incurring into failures.

Every WMS has its own pros and cons, but we found that TaskFlow has limited (or at least confusing) support for orchestrating work-flows over a distributed infrastructure: it permits to declare work-flow engines also as *workers*, that are separate processes dedicated for certain task execution. If running on other machines, engines are connected through the *kombu* python messaging library. Differently, Pegasus[39] seems to better suits our needs, likely because it has native explicit support for clusters and grids: it works in combination with *HTCondor*[40], a full-featured workload management system for compute-intensive jobs.

At the time of writing, a full working cloud solution based on Pegasus is still under development, mainly because we started with a small scale cloud infrastructure which was sufficient to test and benchmark our ideas. We have built our own simpler WMS, loosely inspired by the above, in order to be able to validate our solution. Simpler does not mean less efficient: our WMS reliably schedules tasks on worker machines that are marked as *ready* to execute a task. We used the *paramiko* python library to handle SSH connections to remote machines: paramiko takes care of monitoring available remote workers and supports a basic error handling that either attempts to retry a failed task on a different available worker, or tags it as failed and requests a user-supervised checking.

[38] `https://wiki.openstack.org/wiki/TaskFlow`
[39] `http://pegasus.isi.edu/`
[40] `https://research.cs.wisc.edu/htcondor/`

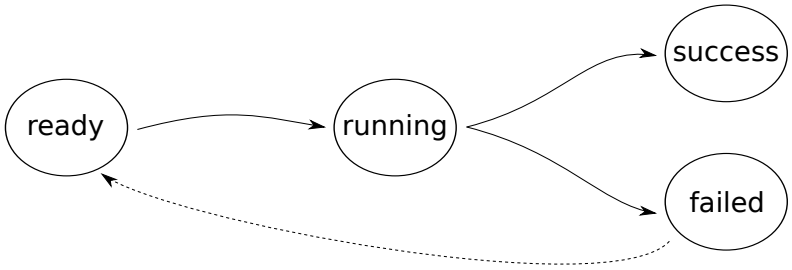


Figure 6.5: *Task states*

Each task in our system contains the name of the service (e.g., an application) to be executed on the remote machine, paths to input files, application's parameters and path to the shared output folder, according to the application's requirements.

Tasks are linked together according to a *pattern of execution*. Specifically, for each pipeline in Figure 6.1, tasks follow a *linear* pattern of execution, because the tasks of a pipeline run one after the other in a serial manner, so as to respect dependencies among tasks. When considering the whole schema, the overall execution follows an *unordered* pattern, where a set of tasks can be executed in any order, provided their input data is available.

A task life cycle is characterised by different states (Figure 6.5): a task is *ready* to be fired for execution when all input files it needs are available. When in this state, the work-flow manager dispatches the task to an available service: a ready task contains paths to the input files needed to execute the job with all optional parameters, plus directives on multi-threading exploitation (the tool used to accomplish a task might or might not support multi-threading). Once fired, a task is in the *running* state and the work-flow manager updates information concerning workers' workload, so that jobs scheduling is optimized to minimize the number of idle services. A task moves to the *success* state after it has finished successfully (i.e. no exceptions were raised during running): a successful job has written all its output files into the shared storage, so that its results can be used by a following step of the pipeline. If a task execution has finished with an error, and maybe some exceptions were raised during running, it enters the *failed* state.

At this point, the failed task is rescheduled, as soon as a worker is ready to

execute it. If it fails again it remains in the failed state: due to the nature of the pipeline's steps a supervised checking is needed in this case, because some input files might be missing or incorrect. A detailed description of the exception occurred is reported, while the pipeline where the failing task belongs is halted. Whichever problem occurred, the user is required to intervene and fix it, so that the task is brought back to the *ready* state and can be scheduled again.

A coarse-grain parallelism is kept up while services are running: every command is sent through a SSH channel and is in turn managed by a *controller* thread responsible for the low-level operations of establishing the connection, while the work-flow manager continues its operations. Once the workflow manager has dispatched a command on a worker machine, the controller thread waits for the command to terminate its execution and captures the exit status and output messages returned by the service when the assigned job has terminated. If a service halts and returns an error message, the task enters the *failed* state and the pipeline where the task belongs is suspended: here a supervised operation is requested to solve the problem. Once the issue is solved the task moves back to the *ready* state and is listed among tasks to be scheduled.

6.4.1.2 Partitioned alignment

Sequence alignment is notoriously a long, time- and resource- consuming task: Bowtie and Bowtie-based alignment tools normally exhibit execution times in the order of hours, depending on dataset size, aligning options, computing power available and memory resources. Timings are likely to increase as the size of datasets increases, but this situation is even worst if the physical memory available is too small, causing the operating system to swap pages when memory demands are greater than physically available for all processes.

Despite being able to exploit parallel threading execution during alignment, the size of the raw data files is by far the most important factor that influences this execution time. In order to cope with this issue, `.fastq` files can be further split, and alignment performed on partial files, while the actual aligned reads file is obtained by merging each partial alignment (Figure 6.2). An ideal policy would suggest to create as many partial files as there are working machines

	full .fastq		split .fastq			
fastq file	size (GB)	align (s)	split (s)	size (GB)	align (s)	merge (s)
SRR206986	3.7	1729	374	0.9	559	149
SRR207094	5.5	4871	571	1.4	1893	276
SRR501780_1	15	28800	1927	3.6	10700	1320

Table 6.1: *Fastq files split into four parts, aligned and merged. Execution times and sizes*

available: the more the parts, the smaller the files, thus the faster should be the alignment. In reality, there is a trade-off among number of split parts, splitting time, alignment time and merging plus sorting time.

Each computing resource performs alignment on a partial `.fastq` file, yielding a partial `.sam` (or `.bam`) file of aligned sequenced reads. Partial outputs are then merged together and sorted, either by mean of *Samtools* [101] facilities or by using Linux commands, depending on the file format (`.bam` files are binary files that must be processed using Samtools).

Table 6.1 reports some details concerning timings and file sizes during alignment steps for the ChIP-Seq (first row), RNA-Seq (second row) and Hi-C (third and fourth row) pipelines. `fastq` files have been split into four parts, each part has been aligned using the proper tool for each specific pipeline, and then outputs have been merged and sorted (when needed). Timings reported are in seconds and reflect wall-clock time measured using Linux `time` tool, on virtual instances equipped with 2 vCPUs. Alignment time for split files is the average of the four alignments. The last column reports the sum of the merge and sort operations. These timings could be lower if more computing power was available, because aligner tools (Bowtie, HiCup, TopHat) can exploit multi-threading for some steps of their processing.

This solution is a valuable mean for reducing alignment timings — compared to the processing of a whole dataset maintaining the same configuration — but mostly it allows to distribute the workload over several computing instances, making effective use of the IaaS cloud paradigm. Nevertheless, performance is still heavily dependent on the underlying computing capabilities and physical memory available: we used basic configurations when launching

alignments, but the number of options varies for each tool, which permit to customise the process and obtain more accurate and detailed outcomes, at the cost of higher memory consumption.

6.5 Test Case

In order to test our infrastructure, we used the datasets from the work of Shen et al. [142], which is particularly interesting since it presents associated RNA-Seq, ChIP-Seq and Hi-C experiments on the same samples of laboratory mouse. In detail, the work comprises 143 datasets from 19 different tissues of mouse, with the aim of annotating cis-regulatory elements of the mouse genome. The peculiarity of this work is the concomitant presence of Hi-C data, which allowed scientists to demonstrate that the mouse genome is organized into domains of regulated enhancers and promoters.

In particular, the ChIP-Seq datasets describe the genomic localizations of RNA polymerase II (PolII) accessibility sites, the CTCF binding sites, and three chromatin modification marks, histone H3 lysine 4 trimethylation (H3K4me3), histone H3 lysine 4 monomethylation (H3K4me1) and H3 lysine27 acetylation (H3K27ac), in 13 adult tissues, four embryonic tissues and two primary cell lines. The transcriptome of each tissue and cell type was determined through RNA-Seq experiments, using the strand-specific sequencing of complementary DNA protocol that can detect both the abundance and strand of origin of RNA transcripts. For a systematic evaluation of the enhancer/promoter pairing relationships, long-range looping interactions have been analysed genome-wide in adult mouse cortex by using the Hi-C method. Therefore, this latter tissue has been used for our experiments of multi-omics data integration.

6.5.1 Results

We selected three genes and a gene cluster for our analysis, which are all connected with the neuronal development and homeostasis, according to the availability of Hi-C data. In particular, SOX2 is a transcription factor that is essential for maintaining self-renewal, or pluripotency, of undifferentiated embryonic

Table 6.2: *ERGM estimates – REST*

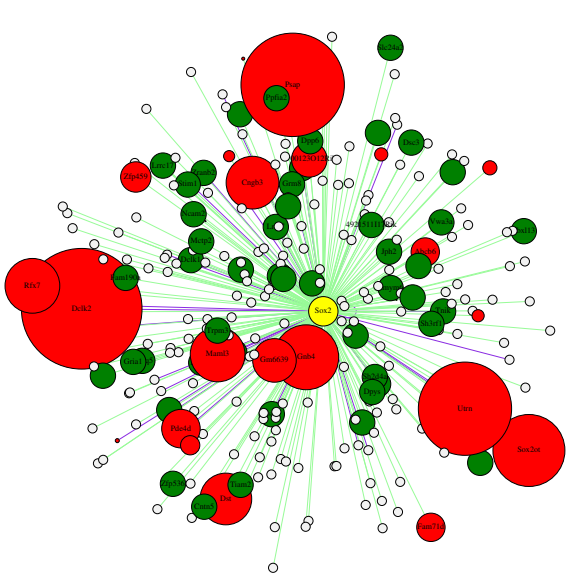
REST	**Estimate**	**Std.Error**
edges + nodecov("transcripts")	0.6841	0.0465
edges + nodecov("H3K4me3")	0.6083	0.0601
edges + nodecov("H3K4me1")	0.5714	0.0517

stem cells. SOX2 has a critical role in maintenance of embryonic and neural stem cells. POU5F1 is a homeodomain transcription factor of the POU family, which is critically involved in the self-renewal of undifferentiated embryonic stem cells and highly expressed in neurons. The Neuron-Restrictive Silencer Factor (NRSF), is a protein which in humans is encoded by the REST gene, and acts as a transcriptional repressor. REST is expressly involved in the repression of neural genes in non-neuronal cells.

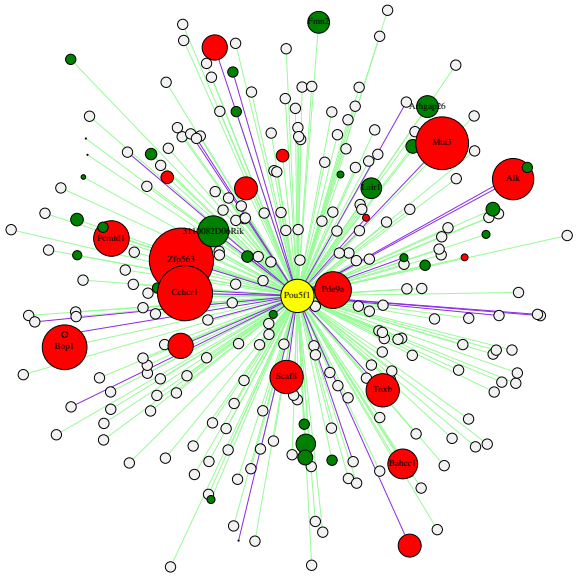
For all these genes we created the neighbourhood graph according to the available Hi-C experiments. Data about differentially expressed genes has been mapped on these "Google maps"-like representations (see Figure 6.6).

Gene expression values have been obtained from the RNA-Seq pipeline, after selecting all and only entries with *p-value* ≤ 0.05. Many co-localized genes have very similar expression patterns (Figure 6.6, red and green nodes), fact that highlights the importance of the spatial components in the interpretation of omic data. Moreover, these expression patterns correspond to what we can see from the epigenetic point of view, since both H3K4me3 and H3K4me1 — landmarks of gene activation — have a distribution comparable to what seen from the transcriptional point of view (Table 6.2, estimate values for the REST gene, computed using the ERGM package, studying the nodal covariate for the "transcripts" feature, and both histone modification marks above).

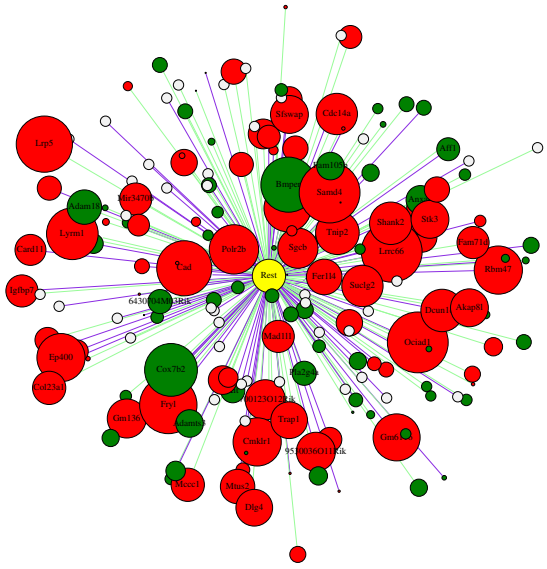
The graph-based representation for the analysis of Hi-C data and for the integration of other multi-omics information is particularly suitable when interpreting information coming from gene clusters, which are groups of genes that encode for similar proteins that collectively share a generalized function and are often located within a few thousand base pairs of each other. Concerning this experiment, Hi-C data shows that the physical partitioning of the genome is highly correlated with the enhancer-promoter units that encompass



(a) *SOX2*



(b) *OCT4*



(c) *REST*

Figure 6.6: *These figures show the neighbourhood graph of a) SOX2, b) POU5F1 and c) REST with mapped differential expression genes. The yellow node identify the gene from which the graph has been built. Red nodes describe positive fold change for the gene expression pattern, while green nodes are negatively expressed genes. The size of the node increases as the expression value increases. Blue edges are intra-chromosome connections, while green edges identify inter-chromosome connections*

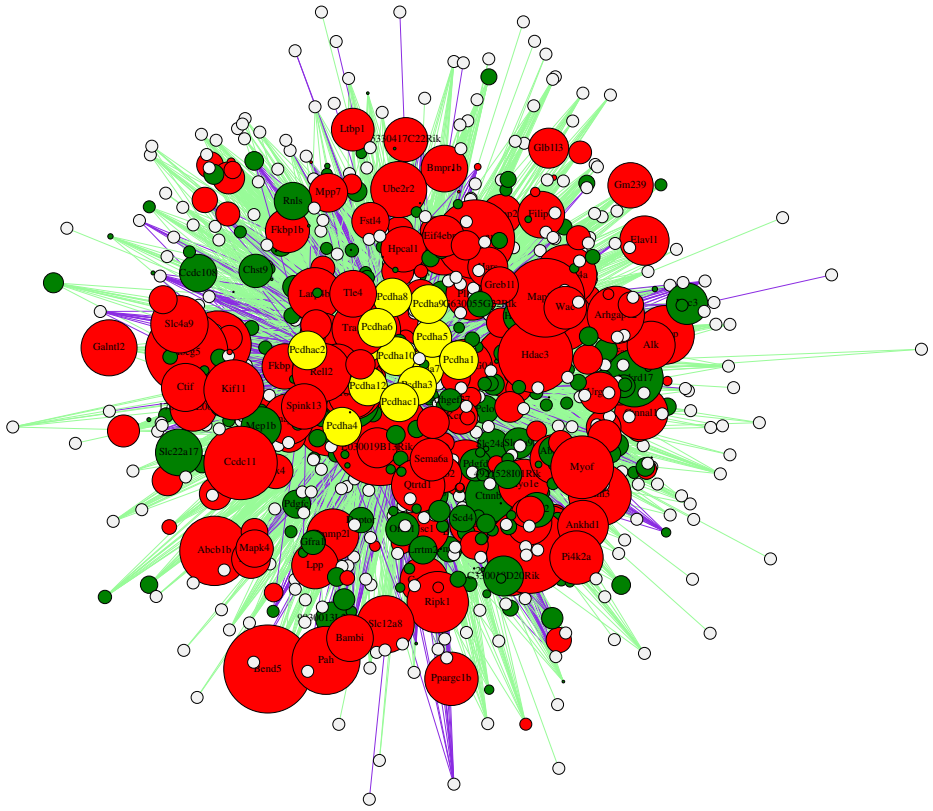


Figure 6.7: *This figure shows the neighbourhood graph of the PCDHα cluster with mapped differential expression genes. Yellow nodes are the gene of the studied gene cluster. Red nodes describe positive fold change for the gene expression pattern, while green nodes are negatively expressed genes. The size of the node increases as the expression value increases. Blue edges are intra-chromosome connections, while green edges identify inter-chromosome connections*

the Protocadherin gene cluster (Pcdhα) on chromosome 18. As reported in Figure 6.7, there is a high correlation between the expression patterns and the epigenetics landmarks, which is well evidenced by the graph-based representation presented in this work. The automatic re-analysis of data performed using our cloud approach clearly can now confirm results proposed in [142].

The last analysis performed on this case study is the statistical modelling of the Protocadherin gene cluster graph. Using ERGM we characterized the distribution of the edges in relation to the characteristics of the nodes under analysis. When running the simulation, we used default parameters to control the underlying Monte Carlo algorithm (`MCMC.burnin=10000,MCMC.interval=1000`),

Table 6.3: *ERGM estimates – Pcdhα*

Pcdhα	**Estimate**	**Std.Error**
edges + nodecov("transcripts")	0.7031	0.01157
edges + nodecov("H3K4me3")	0.6078	0.0867
edges + nodecov("H3K4me1")	0.5067	0.0547

and the final estimator of the network for the Protocadherin gene cluster has a p-value $< 1e^{-4}$. The achieved statistical results are quite interesting to analyse, since the correlation between gene expression and the presence of edges in the graph is quite high. From the epigenetic point of view, the correlation is less strong, but still present, both for H3K4me3 and H3K4me1 (Table 6.3).

6.5.2 Computational costs

The following tables (Table 6.4, 6.5 and 6.6) summarize the computational costs encountered while running our task farm over Shen et al. experiments [142]. Our task farm was implemented over a cloud infrastructure built on top of OpenStack, as described in Section 6.4, with a farm of 8 worker instances, plus the controller node. All instances had a very simple, identical configuration, with 2 vCPUs and 4GB of RAM, except one featuring 4 vCPUs and 8 GB of RAM.

Tables report timings and input sizes for each step of each pipeline, for both the control file (marked with a 'C') and the sample file (marked with a 'S'). Normally, early steps of the pipelines involve conversion of `.sra` files into `.fastq` and alignment. Afterwards, control and sample files are used together to identify significant features from the experiments. When both files are processed together (as it is the case with MACS or Cuffdiff), timings are reported in the sample column only, while input sizes are showed for both.

Tools employed in each step might output more than one file, sometimes containing statistics on the alignment or a summary of the executed operations. However, input sizes reported in our tables reflect the sum of the files (if more than one) passed as input to a specific task. Also note that alignment timings reported are the sum of the average split time, average aligning time and average

	ChIP-Seq			
	SRR206986 (S)		SRR206994 (C)	
step	**time (s)**	**input (GB)**	**time (s)**	**input (GB)**
fastq-dump	138	0.49	149	0.51
alignment	982	3.7	1017	3.9
MACS	344	2.5	–	2.6

Table 6.4: *The table shows execution times (in seconds) and size of the input file (in GB) for each step of the ChIP-Seq pipeline, measuring both the control and the sample file*

merge (and sort) time.

For what it concerns ChIP-Seq, it has been the less demanding pipeline: smaller datasets led to faster execution time. Table 6.4 reports details of a run using SRR206986 experiment as sample file, and SRR206994 experiment as control file (the latter was an input cortex cell line, while the former had CTCF cortex data). Aligned reads from both control and sample files are used to find enriched areas and transcription factor binding sites, by using them as input for the MACS tool. MACS captures the influence of genome complexity to evaluate the significance of enriched ChIP regions, and improves the spatial resolution of binding sites through combining the information of both sequencing tag position and orientation. MACS generates its peak files in a file format called `.bed` file. This is a simple text format containing genomic locations, specified by chromosome, begin and end positions, and some more optional information, including the p-value of peak region.

RNA-Seq pipeline is slightly more complex, and more steps are required to interpret transcriptome assembly and differential expression. TopHat uses Bowtie for read mapping and alignment, but it aligns RNA-Seq reads to a reference genome in order to identify exon-exon splice junctions: by first mapping RNA-Seq reads to the genome, TopHat identifies potential exons, since many RNA-Seq reads will contiguously align to the genome. Using this initial mapping information, TopHat builds a database of possible splice junctions and then maps the reads against these junctions to confirm them. This specific phase of TopHat's processing is very time- (and memory-) consuming, and there is little room to speed it up because it does not benefit from multi-threaded execution, which is instead exploited when aligning reads with Bowtie. When

	RNA-Seq			
	SRR207094 (S)		SRR207095 (C)	
step	**time (s)**	**input (GB)**	**time (s)**	**input (GB)**
fastq-dump	171	0.45	240	0.53
alignment	4871	5.5	4006	6.2
cufflinks	3156	0.6	3421	0.8
cuffmerge	287	1.2	–	1.3
cuffdiff	18840	2.5	–	–

Table 6.5: *The table shows execution times (in seconds) and size of the input file (in MB) for each step of the RNA-Seq pipeline, measuring both the control and the sample file*

we split `.fastq` files we can distribute the workload over several working machines, attempting reducing overall execution time — on average, splitting the file into 4 parts and aligning each part on separate instances, halves the wall-clock execution time, with respect to processing the full dataset — but the overhead of merging partial outputs and sorting the definitive one is not negligible. All in all, performance strongly depend on the underlying computing capabilities.

In order to reconstruct the transcriptome, in the subsequent steps the Cufflinks suite is used: sample and control transcriptome libraries are assembled to quantify their expression. Cuffilnks reports the estimated gene-level expression values in a file named `genes.fpkm_tracking`, and the assembled isoforms in a file called `transcripts.gtf`. Assembled isoforms from sample and control file are then merged together into a master transcriptome assembly, and then used to analyse both mapped reads with Cuffdiff, that compares expression levels of genes and transcripts in both control and sample RNA-Seq experiments. The differential expression at the gene level is reported in a file named `gene_exp.diff`.

The Hi-C pipeline does not use a control file, but operates on paired-end reads: conversion of raw Hi-C data contained in the `.sra` file yields two `.fastq` files with forward and reverse spot, respectively. HiCup maps `.fastq` data against a reference genome and filters out frequently encountered experimental artefacts. It produces paired-read files in `.sam` (or `.bam`) format, each read pair corresponding to a putative Hi-C di-tag.

	HiC			
	SRR501780_1		SRR501780_2	
step	**time (s)**	**input (GB)**	**time (s)**	**input (GB)**
fastq-dump	1517	3.0	1624	3.2
alignment	13606	15+15	–	–
NuchaRt	387	∼8	–	–

Table 6.6: *The table shows execution times (in seconds) and size of the input file (in MB) for each step of the Hi-C pipeline*

The Hi-C pipeline proceeds by building a neighbourhood graph upon the `.sam` file produced by HiCup, starting from the genes of interest. On the graph, we map the genomic information resulting from other pipelines: CTCF binding sites described in the `.bed` files yielded by MACS and differential expression values listed in `gene_exp.diff` yielded by Cuffdiff. Once the graph is constructed, we can conduct statistical analysis over the graph (such as ERGM), visualise data either in textual or tabular format, and eventually plot resulting graphs using Nucha*R*t. The execution time for the Nucha*R*t phase is relative to the ran for gene SOX2 on SRR501780 experiment: we normally reported graph construction and normalisation performance only, because we focused on the parallel implementation. The whole application's execution time is the sum of 5 distinct steps (as described in Section 5.1.1), the others being mostly involved in I/O sequential operations, whose execution time depends on data size.

Most of the benefits in terms of performance clearly derive from the partitioned alignment, which is a valuable mean for reducing alignment timings — compared to the processing of a whole dataset, maintaining the same hardware configuration. With our solution we achieved to distribute the workload over several computing instances, making effective use of the IaaS cloud paradigm. Nevertheless, performance is still heavily dependent on the underlying computing capabilities and physical memory available. Also, we used basic configurations when launching alignment tools, but the number of options varies for each tool, which permit to customise the process and obtain more accurate and detailed outcomes, at the cost of higher memory consumption.

Nevertheless, a clear improvement that this solution brings is the possibility of concurrently executing the analysis pipelines on several sample/control files

couples, provided a sufficient number of services is available, which is obviously related to the (virtual) hardware resources available. An increase in the number of "worker" services must be supported by a proper WMS, which will permit to efficiently orchestrate the whole process.

Minor considerations

Notably, long-lasting tasks are more likely to incur in platforms or connection errors, causing the abortion of the failing job and which would force a job resubmission, increasing the total execution time. Our task farm can handle failures by trying to resubmit a task in a different working instance, but has little automated control over platform errors: misconfiguration of user software or missing libraries can impede an application to start; huge datasets can saturate physical memory, causing an application to immediately abort. As soon as we upgrade our infrastructure with a state-of-the-art WMS, we foresee better handling of failures and errors.

VMs have all been cloned from a fully equipped and tested one, but failures can still occur. In this case the user must manually intervene. At the same time, if some required datasets are missing (such as genome annotations, digest files, genome indexes, etc.) they must be re-generated using the proper tool, and this requires the user to provide for such needs.

It might also be the case that some stages' execution time largely exceed the others, causing a pipeline to halt, waiting for a output data to be ready. In our case, this happened with the Hi-C pipeline, where the NuchaRt stage was waiting for the outputs of both ChIP-Seq and RNA-Seq before it could build graphs with mapped multi-omic information.

6.6 Concluding remarks

The amount of information that can be integrated relying on each single Hi-C map is huge, in the order of tens of GB of raw data. The explosion of experimental datasets available for genome folding analysis and multi-omic integration will pose difficult challenges of data management, which should be carefully

considered by computer scientists working in the field. Cloud Computing can be very useful for this kind of analysis, since the on-demand paradigm is well suited with the possibility (and necessity) of remapping available expression and epigenetic data on new Hi-C experiments. In this Chapter we have shown how the re-interpretation of omics data can be automatically performed using a cartographic representation of the nucleus, as soon as experimental data become available. The presented system attempts to provide an optimal exploitation of the available resources, while good scalability is foreseeable for what it concerns the split/merge of sequenced reads, as the number of virtual instances increases (see Section 6.4.1.2). The whole system permits to produce results in a standardized and reproducible way. Many different protocols and tools can be used in this sense, but the principle is to provide researchers better possibilities to analyse their data, in the view of creating a cloud solution able to fully consider also spatial component in the complexity of biological systems.

Chapter 7

Conclusions

In this thesis we walked through the steps of a high-performance pipeline, whose objective is to promote the usability and availability of novel solutions for the analysis and interpretation of heterogeneous genomic information. Starting from NuChart-II, an application for Hi-C data analysis built using high-level parallel programming paradigms, we embedded our tool into the R environment, which is rich on statistical features and is widely appreciated in most scientific communities for being highly usable and accurate. We eventually integrated Nucha*R*t in larger genomic analysis pipelines, and described their implementation by means of a tailored cloud approach, fostering both efficiency and availability. We tried to address performance and usability in parallel and distributed systems, providing a pragmatic approach linked to Bioinformatics. We believe that our work can be the base for further adaptations and developments, also considering the appreciable results we obtained.

In Chapter 4 we focused on NuChart-II, a C++ application that uses advanced parallel computing techniques (such as lock-free synchronisation and algorithmic skeletons) and applies memory optimisations to provide a gene-centric, graph-based representation of the chromosome spatial organisation, by working on the three-dimensional chromosome information obtainable from Hi-C data. With NuChart-II we have suggested an approach that helps to deal with memory-bound algorithms, which are not unusual in scientific fields: forcing a run-time, working set reduction, we obtained substantial improvements

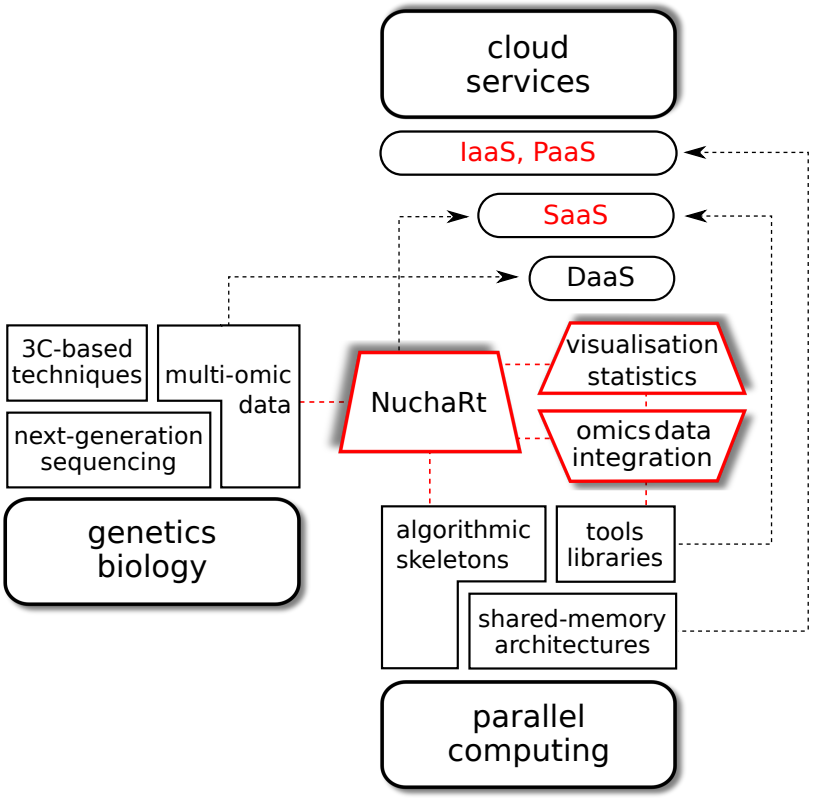


Figure 7.1: *A high-level view of our approach to heterogeneous genomic information processing and interpretation*

in memory bandwidth and cache utilisation, taking full advantage of multi-core architectures. We also elaborated a normalisation approach for biased Hi-C data, that permits to assign a probability to the spatial proximity of two genes, and map this probability on the graph: we described our solution that employs a well-known normalisation algorithm to our graph-based representation, and we discussed its implementation using high-level parallel programming patterns, which achieved outstanding performance and quasi-linear scalability.

In Chapter 5 we focused on usability, and decided to embed NuChart-II into the R environment: R has a rich set of existing statistical packages that facilitates the integration of exploratory data analysis, and permits to easily move through the steps of model development, from data analysis to implementation and visualisation. We designed an R package, Nucha*R*t, that embeds our C++ software as a parallel engine: parallel memory-intensive tasks can be offloaded

to C++, while we rely on R for setting up a usable working environment. Performances are consistently similar to the C++ implementation (for what it concerns parallel computing phases), while there are clear benefits in the possibility of conducing a step-by-step analysis of genomic data.

The availability requirement has been addressed proposing a cloud-based solution that permits to deliver computation and storage as dynamically allocated virtual resources: our gene-centric, graph-based representation of genomic data has been used as the the ground for the integration and analysis of multi-omic information, facilitating the identification of functional patterns arranged in the nucleus of a cell.

Our cloud infrastructure deploys a task farm, where software for data processing is provided as a service, and a task scheduler is responsible for maintaining a coarse-grain parallelism while services execute their tasks over different sorts of data. Every task can fully exploit the underlying computing infrastructure it lands on, while different stages of the data analysis pipeline can be executed in parallel, eventually dealing with different parallelism, memory- and data-access requirements. By partitioning datasets we also managed to ameliorate the performance of a time- (and resource-) consuming task, like sequence alignment can be.

The work discussed in this thesis is a small, though remarkable, step towards our final purpose: we envision a Cloud platform that provides Bioinformatics scientists with computing facilities, software tools and easy access to public data repositories. What we expect is a complete solution for dealing with complex scientific workflows modelled as pipelines, where software services are made available in an fully equipped environment that can embrace novel data processing and visualisation techniques, as novel discoveries spring out from Bioinformatics and Systems Biology communities.

7.1 Open issues and future works

Throughout this discussion, we bumped into some issues that are still open, which we tried to address by studying possible solutions, and upon which we

are currently working on.

Going back to NuChart-II in Chapter 4, the memory management matter still presents a number of open issues: currently, our efforts are focusing on memory allocation for memory-bound parallel applications: as mentioned in section 2.2, we believe that data movement and memory affinity represent two of the cornerstones of performance and efficiency in code production for multicores and heterogeneous architectures. Particularly for memory-bound problems, taking full control of the memory allocation strategy can lead to substantial performance improvement and overall program efficiency: we are investigating a solution that would alleviate the memory burden, by balancing good throughput and good memory utilization, while possibly exploiting memory-mapped I/O for huge read-only datasets. We believe that handling the problem with a pattern-based approach will help to describe data structures that can be effectively coupled with the implicit data path exposed by the computation.

A porting on GPUs is also under evaluation, both for the graph construction phase and the normalisation step. However, this porting is not straightforward, due to the dynamic nature of the application (e.g., loops that *break* when a matching position is found): it seems to be unsuitable for a SIMD design, that notoriously suffers from dynamic memory allocation and conditional branching.

The Nucha*R*t package presented in Chapter 5 is not yet publicly available from R's on-line package repository (CRAN[41]), but we are currently working on making it compliant with CRAN's requirements. The usability requirement encompasses the visualisation of processing results: we have so far managed to draw neighbourhood graphs of small-to-medium size, obtaining a comprehensive representation of the analysed data. However, static visualisation hinders the readability of the graphs, particularly when the size of the graph increases. We are still investigating novel approaches that operates over probabilistic graphs and that would help us to implement an interactive, navigable map of the genome.

Visualisation of genomic data can not overlook experiments resolution: recent works on Hi-C data have shown that more information about chromatin

[41] `https://cran.r-project.org/`

folding can be captured by increasing experiments resolutions [128, 141], which clearly reveals TADs organisation into sub-compartments, where genomic elements have high interaction frequencies. We are currently studying how to incorporate this information in our neighbourhood graphs, making it available as we zoom in into the graph.

For what it concerns our Cloud solutions, it has been tested on a small-scale virtual environment with very basic configurations. The whole infrastructure has to be enriched with a full-featured workflow manager, in order to adhere to standard task scheduling mechanisms. This requirement becomes crucial once we manage to enlarge our virtual infrastructure with proper storage solutions and computing nodes: by widening the number of computing nodes, grows the number of available services, which would reflect in a greater number of samples upon which data analysis pipelines can be executed, concurrently.

Core decomposition and probabilistic graphs

Being the visualisation requirement a serious issue, we have been studying possible solutions to optain a multi-scale, interactive representation of Hi-C data, that permits to variate the amount of rendered details as we zoom closer (or further) into the graph.

One step toward this achievement is to compute the *core decomposition* on a genome-wide graph: this technique is based on the notion of *k-core*, a graph-analysis technique commonly used to find subgraphs that can be computed in linear time [23]. The k-core of a graph is defined as the maximal subgraph in which every vertex has at least degree k within that subgraph. The set of all k-cores of a graph forms its core decomposition. We can employ this decomposition to create a sort of multi-scale visualisation mechanism: when zooming out of the graph we only keep high k cores, while the k decreases as we zoom in, revealing more and more details. However, we cannot keep edges out of this game, because they contain valuable genomic information plus a score resulting from the normalisation, that tells us whether we are actually looking at two likely adjacent genes.

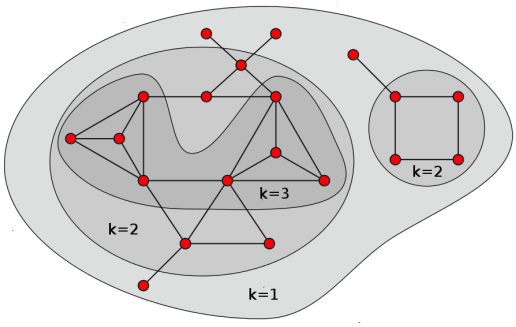


Figure 7.2: *Core decomposition of a (deterministic) graph, adapted from Batagelj V. and Zaveršnik M, 2011 [23]*

In this regard, we could also describe a neighbourhood graph built from Hi-C data as an *uncertain* (or probabilistic) graph, where the uncertainty comes from the biased, noisy experiments. Indeed, we already manage such uncertainty assigning a score to each edge of a graph, according to the genomic features that characterise the relationship between each pair of linked genes (see Section 4.3). Since the range of values of scores varies widely, we can rescale such score so that each value falls in the range $(0, 1]$. What we get is a probability that the relation between the two linked nodes holds; in other worlds, a probability that a given edge exists.

An uncertain graph is defined as $\mathcal{G} = (V, E, p)$, where $p : E \rightarrow (0, 1]$ is a function that assigns a probability of existence to each edge. The connectivity of such network is a complex probabilistic function of the network topology and edge uncertainty. A *possible* graph of an uncertain graph is a deterministic graph which is one of the possible outcome of the random variables representing the edges of the probabilistic graph $\mathcal{G}$. The possible graph is usually denoted as $G = (V_G, E_G)$, with $G \subseteq \mathcal{G}$ and $E_G \subseteq \mathcal{G}$. The total number of possible graphs obtainable from an uncertain graph is $2^{|E|}$, because for each edge we have two cases as to whether or not that edge is present in the graph. The probability of sampling a possible graph G from the uncertain graph $\mathcal{G}$ is:

$$Pr[G] = \prod_{e \in E_G} p(e) \prod_{e \in E \setminus E_G} (1 - p(e))$$

A drawback of probabilistic graphs is that, due to the exponential number of possible worlds, even the simplest analysis problem that would be computed in linear time in a deterministic graph, becomes a #P-complete problem in the uncertain scenario [155]. However, knowledge discovery and data mining communities have long been investigating such problems, proposing solutions that lead to the extraction of representative instances of an uncertain graph, where the representative instances preserve a number of important graph metrics.

Among these approaches, we found two of them that could be interesting for our purposes: the Highly Reliable Subgraphs discovery proposed by Jin et al. [85], and the core decomposition of uncertain graphs proposed by Bonchi et al. [26]. The first solution is described "at least as intractable" as a #P-complete network reliability problem could be, while the latter uses dynamic programming techniques to obtain a core decomposition algorithm in polynomial time.

Both solutions take into account the network topology and edges probability, and use a probability threshold below which graphs elements are dropped, though maintaining the consistency of the information contained in the graph. We are still investigating whether (an adaptation of) these approaches might be of valuable use for our purposes: we believe that by leveraging the probability threshold we can augment or reduce the "details" of a graph, producing a sort of navigable map of the DNA. This would be a first step towards our project of building an interactive, navigable genome browser.

Bibliography

[1] R. Admiraal and M. S. Handcock. `networksis`: a package to simulate bipartite graphs with fixed marginals through sequential importance sampling. *Journal of Statistical Software*, 24(8):1–21, may 2008.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[3] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Targeting distributed systems in fastflow. In *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing*, volume 7640 of *LNCS*, pages 47–56. Springer, 2013.

[4] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Design patterns percolating to parallel programming framework implementation. *International Journal of Parallel Programming*, 42(6):1012–1031, 2014.

[5] M. Aldinucci and M. Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7):449–462, 2006.

[6] M. Aldinucci and M. Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, Oct. 2007.

[7] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Skeletons for multi/many-core systems. In B. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. Peters, and T. Priol, editors, *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009, Lyon, France)*, volume 19 of *Advances in Parallel Computing*, pages 265–272, Lyon, France, 2010. IOS press.

[8] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, Oct. 2014.

[9] M. Aldinucci and M. Torquati. *FastFlow website*, 2009. `http://mc-fastflow.sourceforge.net/`.

[10] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[11] S. Angiuoli, M. Matalka, G. Gussman, K. Galens, M. Vangala, D. Riley, C. Arze, J. White, O. White, and W. Fricke. Clovr: A virtual machine for automated and portable sequence analysis from the desktop using cloud computing. *BMC Bioinformatics*, 356(12), Aug 2011.

[12] J. Antony, P. P. Janes, and A. P. Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/Hyper-Transport. In *Proceedings of the 13th international conference on High Performance Computing*, HiPC'06, pages 338–352, Berlin, Heidelberg, 2006. Springer-Verlag.

[13] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[14] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[15] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[16] T. Asbury, M. Mitman, J. Tang, and W. J. Zheng. Genome3d: A viewer-model framework for integrating and visualizing multi-scale epigenomic information within a three-dimensional genome. *BMC Bioinformatics*, 11(1):444, 2010.

[17] F. Ay, T. Bailey, and W. Noble. Statistical confidence estimation for Hi-C data reveals regulatory chromatin contacts. *Genome Research*, 2014.

[18] F. Ay and W. Noble. Analysis methods for studying the 3D architecture of the genome. *Genome Biology*, 16(1):183+, Sept. 2015.

[19] Babraham Bioinformatics. Available from: http://www.bioinformatics.babraham.ac.uk/projects/hicup/, 2012. Accessed: 2015-11-20.

[20] T. Bailey, P. Krajewski, I. Ladunga, C. Lefebvre, Q. Li, T. Liu, P. Madrigal, C. Taslim, and J. Zhang. Practical guidelines for the comprehensive analysis of chip-seq data. *PLoS Comput Biol*, 9(11):e1003326, 11 2013.

[21] A. Barabási and Z. Oltvai. Network Biology: Understanding the Cell's Functional Organization. *Nature Genetics*, 5:101–114, 2004.

[22] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An Open Source Software for Exploring and Manipulating Networks. In *International AAAI Conference on Weblogs and Social Media*, 2009.

[23] V. Batagelj and M. Zaveršnik. Fast algorithms for determining (generalized) core groups in social networks. *Adv. Data Anal. Classif.*, 5(2):129–145, July 2011.

[24] J.-M. Belton, R. P. McCord, J. H. Gibcus, N. Naumova, Y. Zhan, and J. Dekker. Hic: A comprehensive technique to capture the conformation of genomes. *Methods*, 58(3):268 – 276, 2012. 3D chromatin architecture.

[25] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In L. Rudolph and A. Gupta, editors, *ASPLOS*, pages 117–128. ACM Press, 2000.

[26] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1316–1325, New York, NY, USA, 2014. ACM.

[27] J. Bonwick. The Slab Allocator : An Object-Caching Kernel Memory Allocator, 1994.

[28] G. Boss, P. Malladi, D. Quan, L. Legregni, and H. Hall. Cloud Computing. Version 1.0, IBM White Paper, October 2007.

[29] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, Washington, DC, USA, 2010. IEEE Computer Society.

[30] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[31] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[32] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[33] S. Campa, M. Danelutto, M. Goli, H. González-Vélez, A. M. Popescu, and M. Torquati. Parallel patterns for heterogeneous cpu/gpu architectures: Structured parallelism from cluster to cloud. *Future Generation Computer Systems*, 37:354–366, 2014.

[34] S. Campa, M. Danelutto, M. Torquati, H. González-Vélez, and A. M. Popescu. Towards the deployment of fastflow on distributed virtual architectures. In *ECMS*, pages 518–524, 2013.

[35] Y. Chu and D. Corey. RNA Sequencing: Platform Selection, Experimental Design, and Data Interpretation. *Nucleic Acid Therapeutics*, 22(4):271–274, July 2012.

[36] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Munster skeleton library Muesli – a comprehensive overview. In *ERCIS Working paper*, number 7. ERCIS – European Research Center for Information Systems, 2009.

[37] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[38] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[39] L. Dai, X. Gao, Y. Guo, J. Xiao, and Z. Zhang. Bioinformatics clouds for big data manipulation. *Biology Direct*, 7(1), 2012.

[40] M. Danelutto, R. D. Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Compututer Systems*, 8(1-3):205–220, 1992.

[41] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors,

Proc. of 6th Intl. Euro-Par 2000 Parallel Processing, volume 1900 of *LNCS*, pages 1175–1184, Munich, Germany, Aug. 2000. Springer.

[42] M. Danelutto and M. Torquati. Loop parallelism: a new skeleton perspective on data parallel patterns. In M. Aldinucci, D. D'Agostino, and P. Kilpatrick, editors, *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, Torino, Italy, 2014. IEEE.

[43] J. Darlington, Y.-k. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 19–28, New York, NY, USA, 1995. ACM.

[44] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on numa systems. *SIGARCH Comput. Archit. News*, 41(1):381–394, Mar. 2013.

[45] E. de Wit and W. de Laat. A decade of 3C technologies: insights into nuclear organization. *Genes & Development*, 26(1):11–24, Jan. 2012.

[46] J. Dekker, K. Rippe, M. Dekker, and N. Kleckner. Capturing Chromosome Conformation. *Science*, 295(5558):1306–1311, Feb. 2002.

[47] M. Diener, E. H. M. da Cruz, and P. O. A. Navaux. Locality vs. balance: Exploring data mapping policies on NUMA systems. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*, pages 9–16, 2015.

[48] J. Dixon, S. Selvaraj, F. Yue, A. Kim, Y. Li, Y. Shen, M. Hu, J. Liu, and B. Ren. Topological domains in mammalian genomes identified by analysis of chromatin interactions. *Nature*, 485(5):376–80, 2012.

[49] J. Dostie, T. A. Richmond, R. A. Arnaout, R. R. Selzer, W. L. Lee, T. A. Honan, E. D. Rubio, A. Krumm, J. Lamb, C. Nusbaum, R. D. Green, and J. Dekker. Chromosome Conformation Capture Carbon Copy (5C): a massively parallel solution for mapping interactions between genomic elements. *Genome Research*, 16(10):1299–1309, Oct. 2006.

[50] M. Drocco, C. Misale, G. Peretti Pezzi, F. Tordini, and M. Aldinucci. Memory-optimised parallel processing of Hi-C data. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*, pages 1–8. IEEE, Mar. 2015.

[51] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *Software Engineering, IEEE Transactions on*, 16(6):660–673, Jun 1990.

[52] D. Eddelbuettel. *Seamless R and C++ Integration with Rcpp*. Springer, New York, 2013. ISBN 978-1-4614-6867-7.

[53] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[54] J. Enmyren and C. W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[55] X. Feng, R. Grossman, and L. Stein. Peakranger: A cloud-enabled peak caller for chip-seq data. *BMC Bioinformatics*, 12:139, 2011.

[56] C. Ferrai, I. J. de Castro, L. Lavitas, M. Chotalia, and A. Pombo. Gene positioning. *Cold Spring Harbor perspectives in biology*, 2(6):a000588, 2010.

[57] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[58] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid - Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15:2001, 2001.

[59] J. Fraser, I. Williamson, W. A. Bickmore, and J. Dostie. An overview of genome organization and how we got there: from FISH to Hi-C. *Microbiology and Molecular Biology Reviews*, 79(3):347–372, 2015.

[60] M. J. Fullwood, M. H. Liu, Y. F. Pan, J. Liu, H. Xu, Y. B. Mohamed, Y. L. Orlov, S. Velkov, A. Ho, P. H. Mei, E. G. Y. Chew, P. Y. Huang, W.-J. Welboren, Y. Han, H. S. Ooi, P. N. Ariyaratne, V. B. Vega, Y. Luo, P. Y. Tan, P. Y. Choy, Wansa, B. Zhao, K. S. Lim, S. C. Leow, J. S. Yow, R. Joseph, H. Li, K. V. Desai, J. S. Thomsen, Y. K. Lee, Karuturi, T. Herve, G. Bourque, H. G. Stunnenberg, X. Ruan, V. Cacheux-Rataboul, W.-K. Sung, E. T. Liu, C.-L. Wei, E. Cheung, and Y. Ruan. An oestrogen-receptor-alpha-bound human chromatin interactome. *Nature*, 462(7269):58–64, Nov. 2009.

[61] V. A. Fusaro, P. Patil, E. Gafni, D. P. Wall, and P. J. Tonellato. Biomedical cloud computing with amazon web services. *PLoS Comput Biol*, 7(8):1–6, 08 2011.

[62] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[63] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(2SI):15–26, May 1990.

[64] J. Gibcus and J. Dekker. The Hierarchy of the 3D Genome. *Molecular Cell*, 49(5):773–782, 2013.

[65] A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Nbmalloc: Allocating memory in a lock-free manner. *Algorithmica*, 58(2):304–338, 2010.

[66] A. Gogol-Dring and W. Chen. An overview of the analysis of next generation sequencing data. In J. Wang, A. C. Tan, and T. Tian, editors, *Next Generation Microarray Bioinformatics*, volume 802 of *Methods in Molecular Biology*, pages 249–257. Humana Press, 2012.

[67] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw., Pract. Exper.*, 40(12):1135–1160, 2010.

[68] J. L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[69] M. S. Handcock, D. R. Hunter, C. T. Butts, S. M. Goodreau, and M. Morris. `statnet`*: Software tools for the Statistical Modeling of Network Data*. Seattle, WA, 2003.

[70] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[71] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

[72] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[73] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 276–290, New York, NY, USA, 1988. ACM.

[74] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[75] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 78–88, Washington, DC, USA, 2011. IEEE Computer Society.

[76] M. Hu, K. Deng, S. Selvaraj, Z. Qin, B. Ren, and J. S. Liu. HiCNorm: removing biases in Hi-C data via poisson regression. *Bioinformatics*, 28(23):3131–3133, 2012.

[77] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1134–1139, Washington, DC, USA, 2010. IEEE Computer Society.

[78] T. Hubbard, D. Andrews, M. Caccamo, G. Cameron, Y. Chen, M. Clamp, L. Clarke, G. Coates, T. Cox, F. Cunningham, V. Curwen, T. Cutts, T. Down, R. Durbin, X. M. Fernandez-Suarez, J. Gilbert, M. Hammond, J. Herrero, H. Hotz, K. Howe, V. Iyer, K. Jekosch, A. Kahari, A. Kasprzyk, D. Keefe, S. Keenan, F. Kokocinsci, D. London, I. Longden, G. McVicker, C. Melsopp, P. Meidl, S. Potter, G. Proctor, M. Rae, D. Rios, M. Schuster, S. Searle, J. Severin, G. Slater, D. Smedley, J. Smith, W. Spooner, A. Stabenau, J. Stalker, R. Storey, S. Trevanion, A. Ureta-Vidal, J. Vogel, S. White, C. Woodwark, and E. Birney. Ensembl 2005. *Nucleic Acids Research*, 33(suppl 1):D447–D453, 2005.

[79] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. Mcrtmalloc: a scalable transactional memory allocator. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 74–83, New York, NY, USA, 2006. ACM.

[80] A. Hunter, D. Schibeci, H. L. Hiew, and M. I. Bellgard. Grendel: A bioinformatics Web Service-based architecture for accessing HPC resources. In

R. Buyya, P. D. Coddington, P. Montague, R. Safavi-Naini, N. P. Sheppard, and A. L. Wendelborn, editors, *ACSW Frontiers*, volume 44 of *CRPIT*, pages 29–32. Australian Computer Society, 2005.

[81] T. Ideker and D. Lauffenburger. Building with a scaffold: emerging strategies for high- to low-level cellular modeling. *Trends in Biotechnology*, 21(6):255–262, Aug. 2003.

[82] Intel Corp. *Threading Building Blocks*, 2011.

[83] J.A. Lorenzo and J.C. Pichel and F.F. Rivera and T.F. Pena and J.C. Cabaleiro. A flexible and dynamic page migration infrastructure based on hardware counters. *Journal of Supercomputing*, 65(2):930–948, 2013.

[84] V. Jackson. Formaldehyde cross-linking for studying nucleosomal dynamics. *Methods*, 17(2):125 – 139, 1999.

[85] R. Jin, L. Liu, and C. C. Aggarwal. Discovering highly reliable subgraphs in uncertain graphs. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 992–1000, New York, NY, USA, 2011. ACM.

[86] D. Johnson, A. Mortazavi, R. Myers, and B. Wold. Genome-wide mapping of in vivo protein-dna interactions. *Science*, 316(5830):1497–1502, June 2007.

[87] D. Karolchik, G. P. Barber, J. Casper, H. Clawson, M. S. Cline, M. Diekhans, T. R. Dreszer, P. A. Fujita, L. Guruvadoo, M. Haeussler, R. A. Harte, S. Heitner, A. S. Hinrichs, K. Learned, B. T. Lee, C. H. Li, B. J. Raney, B. Rhead, K. R. Rosenbloom, C. A. Sloan, M. L. Speir, A. S. Zweig, D. Haussler, R. M. Kuhn, and W. J. Kent. The UCSC genome browser database: 2014 update. *Nucleic Acids Research*, 42(D1):D764–D770, 2014.

[88] Khronos Compute Working Group. *OpenCL*, Nov. 2009. `http://www.khronos.org/opencl/`.

[89] A. Kleen. A NUMA API for LINUX. Technical report, SUSE Labs, Apr. 2005.

[90] A. Kleen. Scaling existing lock-based applications with lock elision. *Commun. ACM*, 57(3):52–56, Mar. 2014.

[91] C. Knief. Analysis of plant microbe interactions in the era of next generation sequencing technologies. *Frontiers in Plant Science*, 5(216), 2014.

[92] K. Krampis, T. Booth, B. Chapman, B. Tiwari, M. Bicak, D. Field, and K. E. Nelson. Cloud biolinux: pre-configured and on-demand bioinformatics computing for the genomics community. *BMC Bioinformatics*, 13(1):1–8, 2012.

[93] A. Kukanov and M. J. Voss. The Foundations for Scalable Multicore Software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(04), Nov. 2007.

[94] B. R. Lajoie, J. Dekker, and N. Kaplan. The hitchhikers guide to Hi-C analysis: Practical guidelines. *Methods*, 72:65 – 75, 2015. (Epi)Genomics approaches and their applications.

[95] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[96] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(11):1–10, 2009.

[97] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[98] C. E. Leiserson and I. B. Mirman. How to Survive the Multicore Software Revolution (or at Least Survive the Hype) . *Cilk ARTS*, 2008.

[99] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *SIGARCH Comput. Archit. News*, 18(2SI):148–159, May 1990.

[100] M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 289–296, Washington, DC, USA, 2010. IEEE Computer Society.

[101] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078, 2009.

[102] E. Lieberman-Aiden, N. L. van Berkum, L. Williams, M. Imakaev, T. Ragoczy, A. Telling, I. Amit, B. R. Lajoie, P. J. Sabo, M. O. Dorschner, R. Sandstrom, B. Bernstein, M. A. Bender, M. Groudine, A. Gnirke, J. Stamatoyannopoulos, L. A. Mirny, E. S. Lander, and J. Dekker. Comprehensive Mapping of Long-Range Interactions Reveals Folding Principles of the Human Genome. *Science*, 326(5950):289–293, Oct. 2009.

[103] L. Liu, Y. Li, S. Li, N. Hu, Y. He, R. Pong, D. Lin, L. Lu, and M. Law. Comparison of Next-Generation Sequencing Systems. *Journal of Biomedicine and Biotechnology*, 2012:1–11, 2012.

[104] P. Mahadevan, D. Krioukov, M. Fomenkov, X. Dimitropoulos, K. C. Claffy, and A. Vahdat. The Internet AS-level Topology: Three Data Sources and One Definitive Metric. *SIGCOMM Comput. Commun. Rev.*, 36(1):17–26, Jan. 2006.

[105] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccnuma systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 90–99, New York, NY, USA, 2006. ACM.

[106] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proc. of the 1st Inter. conference on Scalable information systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[107] I. Merelli, P. Liò, and L. Milanesi. Nuchart: An r package to study gene spatial neighbourhoods with multi-omics annotations. *PLoS ONE*, 8(9):e75146, 09 2013.

[108] I. Merelli, F. Tordini, M. Drocco, M. Aldinucci, P. Liò, and L. Milanesi. Integrating multi-omic features exploiting chromosome conformation capture data. *Frontiers in Genetics*, 6(40), 2015.

[109] M. Michael and M. Scott. *Concurrent Update on Multiprogrammed Shared Memory Multiprocessors*. Technical report. University of Rochester, Department of Computer Science, 1996.

[110] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, June 2004.

[111] T. Mikkelsen, M. Ku, D. Jaffe, B. Issac, E. Lieberman, G. Giannoukos, P. Alvarez, W. Brockman, T.-K. Kim, R. Koche, W. Lee, E. Mendenhall, A. O'Donovan, A. Presser, C. Russ, X. Xie, A. Meissner, M. Wernig, R. Jaenisch, C. Nusbaum, E. Lander, and B. Bernstein. Genome-wide maps of chromatin state in pluripotent and lineage-committed cells. *Nature*, 448(7153):553–560, Aug. 2007.

[112] M. Morey, A. Fernndez-Marmiesse, D. Castieiras, J. M. Fraga, M. L. Couce, and J. A. Cocho. A glimpse into past, present, and future {DNA}

sequencing. *Molecular Genetics and Metabolism*, 110(12):3 – 24, 2013. Special Issue: Diagnosis.

[113] S. Mukherjee, H. Erickson, and D. Bastia. Enhancer-origin interaction in plasmid R6K involves a DNA loop mediated by initiator protein. *Cell*, 52(3):375 – 383, 1988.

[114] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing*, 68(9):1186 – 1200, 2008.

[115] U. Nagalakshmi, Z. Wang, K. Waern, C. Shou, D. Raha, M. Gerstein, and M. Snyder. The transcriptional landscape of the yeast genome defined by RNA Sequencing. *Science*, 320:1344–1349, 2008.

[116] J. A. Nelder and R. W. M. Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society, Series A, General*, 135:370–384, 1972.

[117] T. Nguyen, W. Shi, and D. Ruden. Cloudaligner: A fast and full-featured mapreduce based tool for sequence mapping. *BMC Research Notes*, 4(1):1–7, 2011.

[118] J. W. Nicol, G. A. Helt, S. G. Blanchard, A. Raja, and A. E. Loraine. The integrated genome browser: free software for distribution and exploration of genome-scale datasets. *Bioinformatics*, 25(20):2730–2731, 2009.

[119] Oaks, Scott and Wong, Henry. *Java Threads, 3rd Edition*. O'Reilly Media, 2004.

[120] K. Okonechnikov, O. Golosova, M. Fursov, and the UGENE team. Unipro UGENE: a unified bioinformatics toolkit. *Bioinformatics*, 28(8):1166–1167, Apr. 2012.

[121] D. A. Orozco, E. Garcia, R. Khan, K. Livingston, and G. R. Gao. Toward high-throughput algorithms on many-core architectures. *TACO*, 8(4):49, 2012.

[122] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[123] C. S. Pareek, R. Smoczynski, and A. Tretyn. Sequencing technologies and genome sequencing. *Journal of Applied Genetics*, 52(4):413–435, 2011.

[124] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann. Parallel programming environment for OpenMP. *Scientific Programming*, 9:143–161, 2001.

[125] E. Pettersson, J. Lundeberg, and A. Ahmadian. Generations of sequencing technologies. *Genomics*, 93(2):105 – 111, 2009.

[126] C. Pousa Ribeiro and J.-F. Méhaut. Minas: Memory Affinity Management Framework. Technical report RR-7051, INRIA, 2009.

[127] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[128] S. S. P. Rao, M. H. Huntley, N. C. Durand, E. K. Stamenova, I. D. Bochkov, J. T. Robinson, A. L. Sanborn, I. Machol, A. D. Omer, E. S. Lander, and E. L. Aiden. A 3D Map of the Human Genome at Kilobase Resolution Reveals Principles of Chromatin Looping. *Cell*, 159(7):1665–1680, Dec. 2014.

[129] C. Ravishankar and J. Goodman. Cache implementation for multiple microprocessors. Jan 1983.

[130] R. Reagans and B. McEvily. Network structure and knowledge transfer: The effects of cohesion and range. *Administrative Science Quarterly*, 48(2):pp. 240–267, 2003.

[131] D. Reed. *High-Performance Computing: Where'd The Abstractions Go?* BLOG@CACM, May 2009.

[132] A. Réka. Network Inference, Analysis, and Modeling in Systems Biology. *The Plant Cell Online*, 19(11):3327–3338, Nov. 2007.

[133] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, and G. P. Nolan. Cloud and heterogeneous computing solutions exist today for the emerging big data problems in biology. *Nature reviews. Genetics*, 12(3):224, Mar. 2011.

[134] M. C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[135] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 84–94, New York, NY, USA, 2006. ACM.

[136] V. C. Seitan, A. J. Faure, Y. Zhan, R. P. P. McCord, B. R. Lajoie, E. Ing-Simmons, B. Lenhard, L. Giorgetti, E. Heard, A. G. Fisher, P. Flicek,

J. Dekker, and M. Merkenschlager. Cohesin-based chromatin interactions enable regulated gene expression within preexisting architectural compartments. *Genome research*, 23(12):2066–2077, Dec. 2013.

[137] Sequence Read Archive Submissions Staff. Using the SRA Toolkit to convert .sra files into other formats. Available from: http://www.ncbi.nlm.nih.gov/books/NBK158900/, 2011. Accessed: 2015-07-29.

[138] N. Servant, B. R. Lajoie, E. P. Nora, L. Giorgetti, C.-J. Chen, E. Heard, J. Dekker, and E. Barillot. HiTC: exploration of high-throughput 'C' experiments. *Bioinformatics*, 28(21):2843–2844, Nov. 2012.

[139] Y. Shavit, F. Hamey, and P. Liò. FisHiCal: an R package for iterative FISH-based calibration of Hi-C data. *Bioinformatics*, 30(18), Sept. 2014.

[140] Y. Shavit and P. Liò. Combining a wavelet change point and the bayes factor for analysing chromosomal interaction data. *Mol. BioSyst.*, 10:1576–1585, 2014.

[141] Y. Shavit, B. J. Walker, and P. Lio. Hierarchical block matrices as efficient representations of chromosome topologies and their application for 3c data integration. *Bioinformatics*, 2015.

[142] Y. Shen, F. Yue, D. F. McCleary, Z. Ye, L. Edsall, S. Kuan, U. Wagner, J. Dixon, L. Lee, V. V. Lobanenkov, and B. Ren. A map of the cis-regulatory sequences in the mouse genome. *Nature*, 488(7409):116–120, Aug. 2012.

[143] M. Simonis, P. Klous, D. Splinter, Y. Moshkin, R. Willemsen, E. de Wit, B. van Steensel, and W. de Laat. Nuclear organization of active and inactive chromatin domains uncovered by chromosome conformation capture-on-chip (4c). *Nature Genetics*, 38(11):1348–1354, Nov. 2006.

[144] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

[145] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[146] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In *Proceedings of InPar Conference*, 2012.

[147] M. Steuwer, P. Kegel, and S. Gorlatch. Skelcl - a portable skeleton library for high-level gpu programming. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 0:1176–1182, 2011.

[148] F. Tordini, M. Drocco, I. Merelli, L. Milanesi, P. Liò, and M. Aldinucci. NuChart-II: a graph-based approach for the analysis and interpretation of Hi-C data. In *Post-Conference proceedings of the 11th Intl. meeting on Computational Intelligence methods for Bioinformatics and Biostatistics (CIBB 2014)*, volume 8623 of *LNBI*, Cambridge, UK, 2015. Springer.

[149] F. Tordini, M. Drocco, C. Misale, L. Milanesi, P. Liò, I. Merelli, and M. Aldinucci. Parallel exploration of the nuclear chromosome conformation with NuChart-II. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*. IEEE, Mar. 2015.

[150] F. Tordini, I. Merelli, L. Milanesi, P. Liò, and M. Aldinucci. NuchaRt: embedding high performance computing in R for augmented DNA exploration. In *Post-proceedings of the 12th Intl. meeting on Computational Intelligence methods for Bioinformatics and Biostatistics (CIBB 2015)*, LNBI, Naples, Italy, 2016. Springer. To appear.

[151] C. Trapnell, L. Pachter, and S. L. Salzberg. Tophat: discovering splice junctions with rna-seq. *Bioinformatics*, 25(9):1105–1111, 2009.

[152] C. Trapnell, B. A. Williams, G. Pertea, A. Mortazavi, G. Kwan, M. J. van Baren, S. L. Salzberg, B. J. Wold, and L. Pachter. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*, 28(5):511–515, May 2010.

[153] H. L. Truong and S. Dustdar. On analyzing and specifying concerns for data as a service. In M. Kirchberg, P. C. K. Hung, B. Carminati, C. Chi, R. Kanagasabai, E. D. Valle, K. Lan, and L. Chen, editors, *4th IEEE Asia-Pacific Services Computing Conference, IEEE APSCC 2009, Singapore, December 7-11 2009, Proceedings*, pages 87–94. IEEE, 2009.

[154] University of California, Santa Cruz. Ucsc genome browser. http://genome.ucsc.edu/, 2012.

[155] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[156] J. C. e. a. Venter. The Sequence of the Human Genome. *Science*, 291(5507):1304–1351, 2001.

[157] Z. Wang, M. Gerstein, and M. Snyder. RNA-Seq: a revolutionary tool for transcriptomics. *Nature reviews. Genetics*, 10(1):57–63, Jan. 2009.

[158] K. Wetterstrand. DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP). Available at: `www.genome.gov/sequencingcosts`. Accessed: 2015-09-02.

[159] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[160] H. Wickham. *Advanced R*. Chapman and Hall/CRC, 1 edition, Oct. 2014.

[161] S. Widmer, D. Wodniok, N. Weber, and M. Goesele. Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 120–126, New York, NY, USA, 2013. ACM.

[162] W. Winterbach, P. V. Mieghem, M. J. T. Reinders, H. Wang, and D. de Ridder. Topology of molecular interaction networks. *BMC Systems Biology*, 7:90, 2013.

[163] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[164] E. Yaffe and A. Tanay. Probabilistic modeling of Hi-C contact maps eliminates systematic biases to characterize global chromosomal architecture. *Nat Genet*, 43(11):1059–1065, Nov. 2011.

[165] L. Zhang, S. Gu, Y. Liu, B. Wang, and F. Azuaje. Gene set analysis in the cloud. *Bioinformatics (Oxford, England)*, 28(2):294–295, Jan. 2012.

[166] Y. Zhang, T. Liu, C. A. Meyer, J. Eeckhoute, D. S. Johnson, B. E. Bernstein, C. Nusbaum, R. M. Myers, M. Brown, W. Li, and X. S. Liu. Model-based Analysis of ChIP-Seq (MACS). *Genome Biology*, 9(9):R137+, Sept. 2008.

[167] Z. Zhao, G. Tavoosidana, M. Sjolinder, A. Gondor, P. Mariano, S. Wang, C. Kanduri, M. Lezcano, K. Singh Sandhu, U. Singh, V. Pant, V. Tiwari, S. Kurukuti, and R. Ohlsson. Circular chromosome conformation capture (4C) uncovers extensive networks of epigenetically regulated intra- and interchromosomal interactions. *Nat Genet*, 38(11):1341–1347, Nov. 2006.