**NuChart-II: The road to a fast and scalable tool for Hi-C data analysis**

Postprint copy of F. Tordini, M. Drocco, C. Misale, L. Milanesi, P. LiÒ, I. Merelli, M. Torquati, and M. Aldinucci, "NuChart-II: the road to a fast and scalable tool for Hi-C data analysis," International Journal of High Performance Computing Applications (IJHPCA), 2016. doi: 10.1177/1094342016668567


# NuChart-II: the road to a fast and scalable tool for Hi-C data analysis

Fabio Tordini*, Maurizio Drocco*, Claudia Misale*, Luciano Milanesi‡, Pietro Liò†, Ivan Merelli‡, Massimo Torquati§, Marco Aldinucci*

*Computer Science Department, University of Torino, Italy.
†Computer Laboratory, University of Cambridge, UK.
‡Institute for Biomedical Technologies - Italian National Research Council, Segrate (Mi), Italy.
§Computer Science Department, University of Pisa, Italy.

**Abstract**

Recent advances in molecular biology and bioinformatics techniques brought to an explosion of the information about the spatial organisation of the DNA in the nucleus of a cell. High-throughput molecular biology techniques provide a genome-wide capture of the spatial organization of chromosomes at unprecedented scales, which permit to identify physical interactions between genetic elements located throughout a genome. This important information is however hampered by the lack of biologists-friendly analysis and visualisation software: these disciplines 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 and integrated, with substantial requirements in speed (in terms of execution time), application scalability and data representation. In this work we present NuChart-II, an efficient and highly optimized tool for genomic data analysis that provides a gene-centric, graph-based representation of genomic information, and proposes an *ex-post* normalisation technique for Hi-C data. While designing NuChart-II we addressed several common issues in the parallelisation of memory bound algorithms for shared-memory systems.

**Index Terms**

High-Performance Computing, Bioinformatics, Hi-C data analysis, Parallel Computing, Memory-bound algorithms

## I. Introduction

A huge amount of information is daily produced in molecular biology laboratories all around the world, but the interpretation of this data in an effective way is a complex and challenging task. Also, an increasing number of experiments highlight the importance of studying the spatial organisation of the DNA in the nucleus, in order to gather insights on the processes ongoing within a cell: there is an undeniable need for 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, advances in molecular biology techniques have been developed to study chromatin interactions and the three-dimensional chromosome folding on a larger scale [1]. In 2002, Dekker et al. developed a strategy called "Chromosome Conformation Capture" (**3C**) [2], that put 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 3D organization of the chromosome in the nucleus of the cell, by detecting the frequencies of interaction between any two genomic *loci* in order to reveal their spatial disposition. Among 3C-based methods, *Hi*-C uses Next-Generation Sequencing (**NGS**) techniques to interrogate the 3C ligation library more comprehensively and with an increased throughput [3]. The "Hi" thus stands for "High-throughput", and sometimes it is also written as *HT-3C*.

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[1], which produces millions of paired-end *reads* (i.e., short DNA sequences with start/end

**Corresponding author:**
*Fabio Tordini, Computer Science Dept., University of Torino, Italy – e-mail: tordini@di.unito.it*
[1]http://www.bioinformatics.babraham.ac.uk/projects/hicup/

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 each phase of 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 II-A), but this approach flattens the possibilities for data interpretation uncovered by Hi-C experiments to a mere frequency count, while the genomic information is blurred beneath. We propose an approach that overcomes these limitations: based on an early R prototype [4] we designed NuChart-II, a C++ application that uses advanced parallel computing techniques (such as non-blocking synchronisation and algorithmic skeletons) and applies memory optimisations to provide a gene-centric, graph-based representation of the chromosome organisation.

The first prototype 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, while it could not be used for highly precise data analysis (which requires many iterations of graph building process). Moreover, the above limitations dramatically limited its usability, causing the application to crash frequently due to memory overflows.

NuChart-II has been designed according to a structured parallel programming approach [5], [6]; in particular, it has been designed on top of the *FastFlow* parallel programming framework, that provides high-level parallel programming patterns for the C++ language.

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 of results. Two main phases 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 to the 80% of the whole execution time.

This paper is organized as follows: section II presents the background from which this work has emerged, giving an overview of the Bioinformatics tools for Hi-C data analysis, together with a briefing on parallel programming techniques. Section III describes the graph construction algorithm and the memory optimisations applied. Section IV focuses on the normalisation step, which is of utmost importance when dealing with raw sequenced data. We will show how this normalisation is employed to obtain the weight of the edges in the graph. Section V describes the experiments conducted to test NuChart-II. Section VI presents a discussion about performance issues and results obtained by applying our approach. Results are compared against other widely used parallel computing frameworks, such as OpenMP and Intel TBB. Section VII concludes this paper.

## II. Related works

In this section we provide a comprehensive list of the available tools designed to explore and visualize Hi-C data, followed by an overview of the most commonly used and studied parallel computing frameworks.

### A. Omic *Tools*

3C-based techniques used to characterize the nuclear organization of genomes and cell types have widespread among scientific communities, and a number of systems biology methods designed to analyse such data have been proposed. Particular attention is given to the detection and normalisation of systematic biases: the raw outputs of many genomic technologies are affected both by technical biases, arising from sequencing and mapping, and biological factors, resulting from intrinsic physical properties of distinct chromatin states. This makes difficult to evaluate their outcomes, as it may result in false-positives or false-negatives.

Considering general software for the interpretation of Hi-C data, an interesting package is *HOMER* [7], 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, while we are writing this paper, it only works at the chromosome level.

*HiTC* [8] has been designed to facilitate the exploration of 3C-based data. It allows users to transform, normalize and visualize interaction maps. An interaction map is a two-dimensional *heat-map* representation of the matrix of Hi-C counts, whose entries correspond to the number of times two restriction fragments in a given genomic region have been ligated in 3C 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.

**TABLE I** – *Software tools for Hi-C data analysis*

| 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 |

*Fit-Hi-C* [9] assigns statistical confidence estimates to mid-range, intra-chromosomal contacts by jointly modelling the random polymer looping effect and previously observed technical biases in Hi-C data sets.

If Fluorescence *in situ* hybridization experiments are available, a good normalization solution is represented by *FisHiCal* [10]. 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 this 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 [11] 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.

Hu et al. proposed HiCNorm [12], which 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. The first NuChart prototype [4] solved this issue by exploiting Hu et al. solution to estimate a score to each read, identifying half of the Hi-C contact instead of normalizing the contact map, thus preserving the sequence information. NuChart-II 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.

Table I 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 I 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 visualization and exploration of Hi-C data assumes a dramatic importance when analysing Hi-C data. 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, with a normalisation technique that evaluates the actual probability of physical proximity.

### *B. Parallel Computing Tools*

Over the years, research on loop parallelism has been carried on using different approaches and techniques that vary from automatic parallelisation to iterations scheduling.

Intel Threading Building Blocks (*TBB*) [13] is a library that enables support for scalable parallel programming using standard C++. It provides high-level abstractions to exploit task-based parallelism, independently from the underlying platform details and threading mechanisms. The TBB `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.

*OpenMP* [14] uses a directive based approach, where the source code is annotated with pragmas (`#pragma omp`) that instruct the compiler about the parallelism to be used in the program. In OpenMP, two constructs are used to parallelise a loop: the *parallel* and the *loop* construct. The *parallel* construct, introduced by the `parallel` directive, declares a parallel region which will be executed in parallel by a pool of threads. The *loop* construct, introduced by the `for` directive, is placed within the parallel region to distribute the loop iterations to the threads executing the parallel region. The two constructs are often fused together into the `#pragma omp parallel for` directive. OpenMP supports several strategies for distributing loop iterations among threads. The scheduling strategy may be specified via the `schedule(type[,chunk size])` clause, which is appended to the `for` directive. The `type` of scheduling policy can be one among *static*, *dynamic*, *guided*, *auto*, *runtime*, each one providing a different thread scheduling policy.

*FastFlow* is a parallel programming environment originally designed to support efficient streaming on cache-coherent multi-core platforms and distributed systems [15], [16]. It is realised as a C++, pattern-based, parallel programming framework, aimed at simplifying the development of applications for multi-core and GPGPUs platforms. It provides developers with a set of high-level, parallel programming patterns (aka *algorithmic skeletons*), obtained by the composition of two basic algorithmic skeletons: a *farm* skeleton, and a *pipeline* skeleton. Leveraging the *farm* skeleton, FastFlow exposes a `ParallelFor` pattern [17], 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.

The FastFlow library has been our first choice among available algorithmic skeleton frameworks: despite many others provide support for shared-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.

## III. 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. With these premises, our neighbourhood graph represents a topological map of the specific nucleus region to which a gene belongs.

### A. 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 [4], 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 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 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 1, row 10). For each discovered read, the algorithm searches for a gene on the second pair of the active read (Listing 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 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.

```
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 1** – *Sequential graph construction*

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 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 1, row 12).

*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. High-level parallelisation of graph exploration has been treated, among others, in [18].

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.

### *B. 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.

The basic BFS implementation of Listing 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

```
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      }
23      // level synchronisation
24      Γ := BuildPartialGraph(V[th],E[th])
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 2** – *Parallel BFS Graph Construction*

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 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. $\Gamma$ is used at every level synchronisation to store partial graphs (Listing 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 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 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`.

### *C. 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. In our implementation we tried to minimise the number of dynamic memory allocations performed by each thread, mostly using pre-allocated containers

```
NormaliseEdge(e, τ, NTH) {
   LenM = GCcM = MapM = CMap := ∅  // genomic features matrices
   X = Y = β := ∅
   Conv := false

   // all matrices have the same size
   CMap := ContactMap(e.Chr1, e.Chr2)
   LenM := BuildLengthMatrix(e)
   GCcM := BuildGCcontentMatrix(e)
   MapM := BuildMappabilityMatrix(e)

   X := Matrix(LenM^, GCcM^)
   Y := Cmap^

   while (not Conv) {
      ApplyLinkFunction(Y)
      β := ApplyGLM(Y, X, MapM^)
      Conv := CheckConvergence(β)
   }
   e.Weight := f(β)
}
```

**Listing 3** – *Normalisation*

(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.

## IV. Normalisation

Particular attention is given to the detection and normalisation of systematic biases: several sequence-dependent features substantially bias Hi-C readouts. These biases 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 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.

Yaffe and Tanay were the first to discover such factors [11] 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 [12]. Based on these achievements, NuChart-II applies an *ex-post* normalisation to each edge of the resulting neighbourhood graph.

In our vision, an edge identifies the existence of a Hi-C read that encompasses two connected 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.

For each edge, a contact map ($Y$) is constructed directly modelling the read count data at a resolution level of 1 Mb (mega base). Hi-C data matrix is symmetric, thus we consider only its upper triangular part, where each point of $Y_{i,j}$ denotes the intensity of the interaction between positions $i$ and $j$. 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. The model is given by the formula:

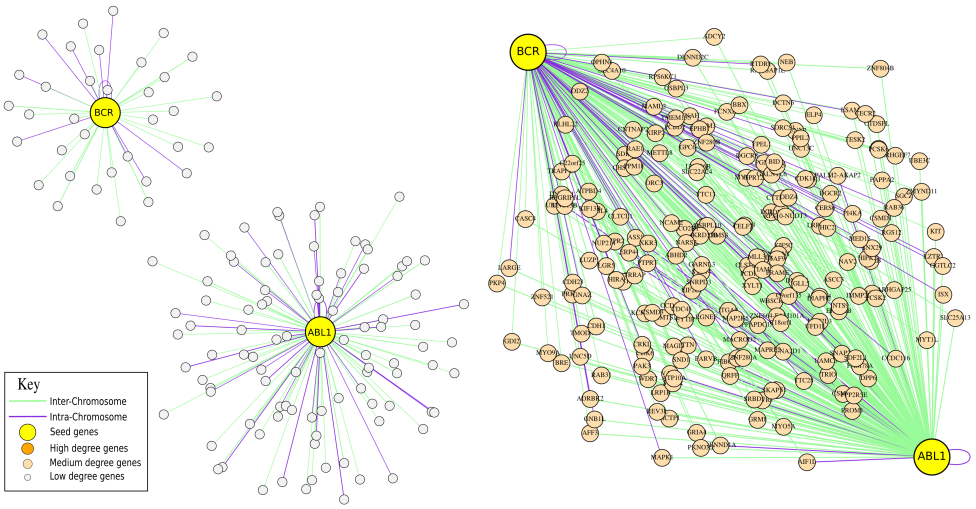


**Fig. 1** – *Neighbourhood graph with genes ABL1 and BCR, according to LiebermanAiden's SRA:SRR027956 (*left*) and SRA:SRR027962 (*right*) experiments*

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

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. $\beta$ 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 estimate for each edge is computed using the *Iteratively Weighted Least Squares* algorithm (IWLS), proposed by Nelder and Wedderburn [19]. 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 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, EBI). Model components are built using arrays containing such upper-triangular values (excluding the diagonal: diagonal values are all zeros, because a chromosome locus does 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 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 $\chi^2$ (*chi-squared*) difference at each iteration is less than a certain threshold $\tau$:

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

In Listing 3, the function `ApplyGLM` writes the best-fit parameters in vector $\beta$, 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.

## V. 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

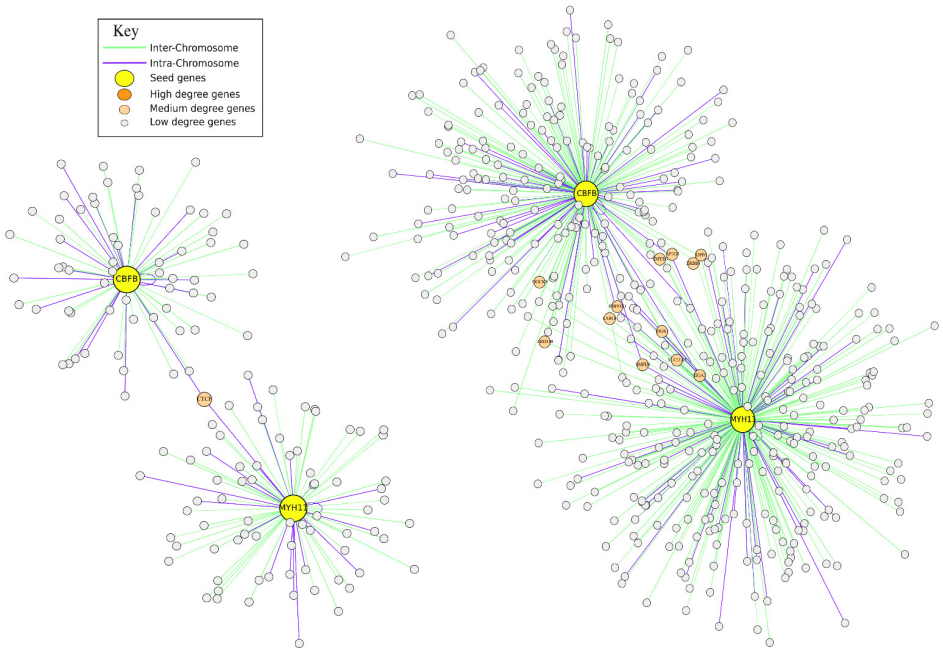


**Fig. 2** – *Neighbourhood graph with genes CBF$\beta$ and MYH11 according to LiebermanAiden's SRA:SRR027959 (left) and SRA:SRR027963 (right) experiments*

the tool. This novel implementation addresses these weaknesses, making possible a genome-wide exploration of Hi-C contacts – thanks to the optimal memory management and data structure design – 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 [4] 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.

We performed several executions involving the creation of neighbourhood graphs for relevant genes or gene clusters, at different levels of iterations, in order to verify how Hi-C can be used for citogenetics studies. In particular, we focused on *Philadelphia* translocation, which is a specific chromosomal abnormality that is associated with chronic myelogenous leukaemia (CML). The result of this translocation is that a fusion gene is created from the juxtaposition of the ABL1 gene on chromosome 9 to part of the *Breakpoint Cluster Region* (BRC) gene on chromosome 22. This is a reciprocal translocation, creating an elongated chromosome 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 translocations in the above context. In particular, our analysis relies on data from LiebermanAiden experiments [3], which consist in normal human lymphoblastoid cells (GM06990) sequenced with Illumina Genome Analyzer, compared with an erythroleukemia cell line (K562) with an aberrant karyotype. Starting from well-established data related to the cytogenetic experiments, we tried to understand if the Hi-C technology can successfully be applied to verify 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 (Figure 1) are distant 2 or 3 contacts in sequencing runs concerning GM06990, while they are in close contact in sequencing runs related to K562 with digestion enzyme HindIII. Therefore, there is a perfect agreement between the positive and the negative presence of Hi-C contacts and FISH data.

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 showed 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, only few genes are shared, thus justifying a lower probability for a translocation to happen. Considering the translocation CBF$\beta$-MYH11 (Figure 2), they are distant 2 or 3 contacts in GM06990, while they are proximal in K562.

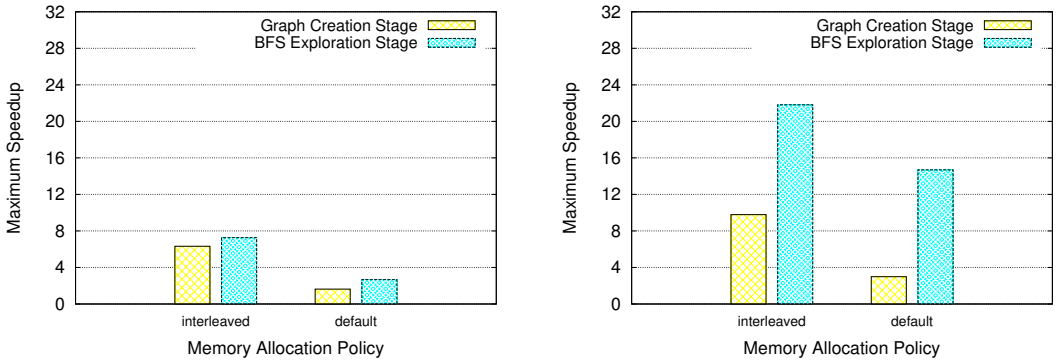


**Fig. 3** – *Execution on NuChart-II with no optimisations (*left*) and using memory optimisations (*right*). The maximum speedup obtained by the parallel execution of the graph creation and BFS exploration phases is reported*

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.

A drawback of this visualization 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).

## VI. Discussion

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 the chromosome spatial conformation as a graph changes the mindset, and permits to focus on interactions occurring among genes, which can in turn be interpreted by applying network analysis over the resulting graph and study graph metrics to reduce structural properties of a network to real number, facilitating the comparison of different networks[2]. For instance, 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 [20]. We have noticed that the degree distribution of common graphs resulting from DNA exploration through Hi-C data analysis leads to consider the genes network as a scale-free network, which suggests further investigations towards detecting the presence of community structures. This network perspective permits to shed some new light over the genes interaction.

### A. Performance

Both the graph construction and the edges weighing phase are bounded to the memory size required to hold the data. 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 target architecture is a NUMA Intel workstation equipped with 4 eight-core E7-4820 Nehalem 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 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. Considering that the application is also tested on a NUMA platform, we executed NuChart-II also using an interleaved memory allocation policy via *numactl* utility, which can be used to control NUMA policy for processes or shared memory. Finally, we relies on to the internal structure of FastFlow *ParallelFor*, which allows to use all physical cores while thread pinning is automatically

[2]While *network* and *graph* are two distinct, though related, concepts, here they are used interchangeably.

**TABLE II** – *Execution times (*in seconds*) for NuChart-II without optimisations (*left*) and with memory optimisations (*right*)*

| **NuChart-II with no optimisations** | | | | |
|---|---|---|---|---|
| | *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 |

| **NuChart-II with memory optimisations** | | | | |
|---|---|---|---|---|
| | *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 |
| 32 | 28 | 38 | 10 | 26 |

managed by the FastFlow library. As a performance metric, together with the overall execution time, we also used the speedup, calculated as $S = T(seq)/T(n)$ where $T(seq)$ is the sequential execution time measured during a plain sequential execution, and $T(n)$ is the parallel execution time using $n$ worker threads, keeping the problem size fixed.

*Graph construction:* Figure 3 reports performance results. On the left we show the maximum speedup obtained executing NuChart-II without optimisations, 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. 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 criterion (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 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. These aspects related to memory layouts have been discussed in Section III. By applying the mentioned optimisations, we obtained an evident performance gain as reported on right side of Figure 3.

Table II (*left*) 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, taking to have 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[3] 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 II (*right*) 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 steps 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 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, starting from a speedup of $\sim 7$ 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. It was possible to achieve this result since, as previously explained, a good filtering of data structures local to each thread, helped in improving execution times by reducing memory traffic. Further experiments with the PerfStat tool showed that, by the working set reduction and memory bandwidth utilization optimisation, it was possible to reduce cache misses number as well, improving also the CPU utilisation. Results are not reported due to limited space. This result led to increasing the throughput in terms of edges visited per time unit, thus dramatically reducing the total execution time.

*Normalisation:* The edges weighing phase is an embarrassingly parallel application: any arbitrary subset of the edges can be processed independently from each other by mean of a parallel loop pattern. With Fastflow `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 features. These data are used to build all matrices needed to

[3]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 non-static members of pointer-to-member type

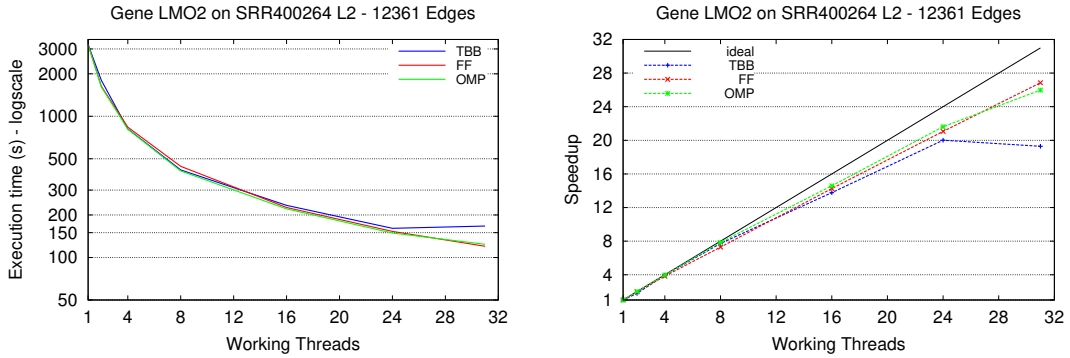


**Fig. 4** – *Execution time (left) and speedup (right) of the normalisation phase for 12361 edges, resulting from a neighbourhood graph for the gene LMO2 with genes distant up to 2 hops from the root, according to the Dixon et al. SRA:SRR400264 experiment*

construct the regression workspace. The task involves tight loops doing Floating Point arithmetic calculations on data that fit the L3 cache and can 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 with FastFlow shows 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). Intel TBB begins to suffer for the dynamic memory allocations when the number of threads is greater than 24, causing its performance 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*, as it attempts to perform some automatic cache optimizations, although it did not bring substantial improvements with respect to the default *auto partitioner*. With FastFlow's `ParallelFor` we found that the best performances were reached when using the scheduler as thread. OpenMP `parallel for` produced the best performance with dynamic scheduling.

We have conducted our tests with a maximum of 31 cores in our machine with 32 cores (excluding two-ways hyper-threading): when using FastFlow's `ParallelFor` 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.

## VII. 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. In the short-term, it might help in focusing on the interactions of a gene with its neighbourhood in the three-dimensional space thus to study correlation of phenomena due to phyisical proximity of genes. In the longer-term, it might enable the study of long-distance and thus more abstract relationships among genes or sub-graphs of thems.

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 to address many of the issues that burdened the original R prototype, obtaining performances that would have been inconceivable with the original R prototype.

As for the future works, we are investigating practical and convenient solutions to address the visualisation problem: starting from community structures detection, the graph visualization can be ameliorated by providing a representation at different scales: when the scale is small only communities and their connections are shown, while as the scale grows genes within a zooming window must become visible and readable. We aim at building dynamic, interactive graphs where all the physical, chemical and statistical information are easily accessible by direct interaction with the graph. We are investigating the benefits obtainable using a scalable, lock-free memory allocator that fosters memory affinity in a NUMA system, hence to get the best memory latency in accessing effectively used data structures.

*Targeting Heterogeneous Devices*

We also evaluated the usage of GPUs for the BFS graph construction, implementing a preliminary GPU version. This porting is not straightforward due to the dynamic nature of the application (e.g., loops that *break* when a matching position is found): GPUs execution model is similar to SIMD and it is better suited for data-parallel applications, while has limited options when it comes to more dynamic workflows. One possible approach is to divide the search space into smaller parts and then offload each chunk to the GPU: for each chunk, if a match is found the search is stopped and the next chunk is offloaded. All in all, after a proper tuning of the chunk size we obtained a performance gain of $10\%$ on a high-end NVidia K40 GPU (compared to a sequential CPU version). We are also considering a possible GPU implementation of the weighing phase, that strongly relies on heavy computations over matrices: here the huge computational power exposed by modern GPU devices can be used to reduce the execution time. The issue here comes from the memory management on device side, since the involved matrices reach considerable sizes and they often need to be dynamically allocated. This feature is available only on modern GPU devices and it affects performance by a non negligible factor. Beside matrices, a number of back up data structures are needed to set up regression and correctness tests (e.g, p-value), forcing frequent accesses to device's global memory, which notoriously has high access latency.

It is a recent trend to exploit the vast variety of heterogeneous platforms in the domain of bioinformatics [21]. Since FastFlow recently introduced the support for offloading to hardware accelerators such as GPU, FPGA, DSP, we plan to explore the usage of such platforms according to an approach analogous to what we discussed above for GPUs.

## ACKNOWLEDGEMENT

This work has been partially supported by the EC-FP7 STREP project "REPARA" (no. 609666), the EU H2020 "RePhrase" project (no. 644235), and the Italian Ministry of Education and Research Flagship (PB05) "InterOmics".

## REFERENCES

[1] E. de Wit and W. de Laat, "A decade of 3C technologies: insights into nuclear organization," *Genes & Development*, vol. 26, no. 1, pp. 11–24, Jan. 2012.
[2] J. Dekker, K. Rippe, M. Dekker, and N. Kleckner, "Capturing Chromosome Conformation," *Science*, vol. 295, no. 5558, pp. 1306–1311, 2002.
[3] E. Lieberman-Aiden, N. L. van Berkum, L. Williams, M. Imakaev, and T. e. a. Ragoczy, "Comprehensive mapping of long-range interactions reveals folding principles of the human genome," *Science*, vol. 326, no. 5950, pp. 289–293, 2009.
[4] I. Merelli, P. Liò, and L. Milanesi, "NuChart: An R Package to Study Gene Spatial Neighbourhoods with Multi-Omics Annotations," *PLoS ONE*, vol. 8, no. 9, Sep. 2013.
[5] 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*, vol. 52, no. 10, pp. 56–67, 2009.
[6] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Design patterns percolating to parallel programming framework implementation," *International Journal of Parallel Programming*, vol. 42, no. 6, pp. 1012–1031, 2014.
[7] 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*, vol. 23, no. 12, pp. 2066–2077, Dec. 2013.
[8] 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*, vol. 28, no. 21, pp. 2843–2844, Nov. 2012.
[9] F. Ay, T. Bailey, and W. Noble, "Statistical confidence estimation for Hi-C data reveals regulatory chromatin contacts," *Genome Research*, 2014.
[10] Y. Shavit, F. Hamey, and P. Liò, "FisHiCal: an R package for iterative FISH-based calibration of Hi-C data," *Bioinformatics*, vol. 30, no. 18, Sep. 2014.
[11] A. T. Eitan Yaffe, "Probabilistic modeling of Hi-C contact maps eliminates systematic biases to characterize global chromosomal architecture," pp. 1059–1065, 2011.
[12] 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 (Oxford, England)*, vol. 28, no. 23, pp. 3131–3133, Dec. 2012.
[13] "Intel Threading Building Blocks, project site," 2013, http://threadingbuildingblocks.org.
[14] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.
[15] M. Aldinucci, M. Meneghin, and M. Torquati, "Efficient Smith-Waterman on multi-core with fastflow," in *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, M. Danelutto, T. Gross, and J. Bourgeois, Eds. Pisa, Italy: IEEE, Feb. 2010, pp. 195–199.
[16] 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*, ser. LNCS, vol. 7640. Springer, 2013, pp. 47–56.
[17] M. Danelutto and M. Torquati, "Loop parallelism: a new skeleton perspective on data parallel patterns," in *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, M. Aldinucci, D. D'Agostino, and P. Kilpatrick, Eds. Torino, Italy: IEEE, 2014.
[18] 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*, ser. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 78–88.
[19] J. A. Nelder and R. W. M. Wedderburn, "Generalized linear models," *Journal of the Royal Statistical Society, Series A, General*, vol. 135, pp. 370–384, 1972.
[20] W. Winterbach, P. V. Mieghem, M. J. T. Reinders, H. Wang, and D. de Ridder, "Topology of molecular interaction networks." *BMC Systems Biology*, vol. 7, p. 90, 2013.
[21] S. Soroushnia, M. Daneshtalab, J. Plosila, T. Pahikkala, and P. Liljeberg, "High performance pattern matching on heterogeneous platform," *J. Integrative Bioinformatics*, vol. 11, no. 3, 2014.