IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Memory-Optimised Parallel Processing of Hi-C Data

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1521910 since 2016-11-19T16:37:57Z

Publisher:

IEEE Computer Society

Published version:

DOI:10.1109/PDP.2015.63

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

02 May 2026

Memory-Optimised Parallel Processing of Hi-C Data

Maurizio Drocco, Claudia Misale, Guilherme Peretti Pezzi, Fabio Tordini and Marco Aldinucci
Computer Science Department, University of Turin, Italy
[drocco, misale, peretti, tordini, aldinuc]@di.unito.it

***Abstract*—This paper presents the optimisation efforts on the creation of a graph-based mapping representation of gene adjacency. The method is based on the Hi-C process, starting from Next Generation Sequencing data, and it analyses a huge amount of static data in order to produce maps for one or more genes. Straightforward parallelisation of this scheme does not yield acceptable performance on multicore architectures since the scalability is rather limited due to the memory bound nature of the problem. This work focuses on the memory optimisations that can be applied to the graph construction algorithm and its (complex) data structures to derive a cache-oblivious algorithm and eventually to improve the memory bandwidth utilisation. We used as running example NuChart-II, a tool for annotation and statistic analysis of Hi-C data that creates a gene-centric neighborhood graph. The proposed approach, which is exemplified for Hi-C, addresses several common issue in the parallelisation of memory bound algorithms for multicore. Results show that the proposed approach is able to increase the parallel speedup from 7x to 22x (on a 32-core platform). Finally, the proposed C++ implementation outperforms the first R NuChart prototype, by which it was not possible to complete the graph generation because of strong memory-saturation problems.**

I. INTRODUCTION

The Hi-C method for genome-wide chromatin 3D conformation is widely used to understand the genome physical conformation and allows the investigation of the interaction of genomic elements [1]. The basic Hi-C process, starting from Next Generation Sequencing data, creates a list of pairs of locations along the chromosome which is represented by a matrix, called *contact map*. By this representation it is easy to identify interactions between two chromosomes, but the description of their physical adjacency or, more in general, the spatial neighborhood of two or more genes is not easily obtainable. Hi-C data, by providing a three-dimensional information of the whole examined genome, naturally takes to the representation of such adjacency by a graph-based mapping and visualisation. By having a graphical visualisation, it is possible not only to have a visual characterisation of the different spatially associated domains of other omics-data, but also it permits to have a more impressive and immediate view about how genes are related and connected among each other. In addition to state-of-art statistical analysis and contact map creation, more statistics can be applied to the obtained graph, such as social network analysis for interactions within genes discovery, clustering or cliques detections.

The construction of such graphs and, in general, contact maps, is based on the exploration of static data. By static data we refer to the genomic DB represented in fig. 1, that is, data that can be considered constant in different graphs creation. In contrast, Hi-C reads files (fig. 1) can be considered as dynamic data, that is, data that can change while keeping the same genomic DB to fulfill different analysis. Accordingly, static data structures, possibly common to all stages of a hypothetical analysis tool, would be entirely loaded at the beginning of the analysis for sake of simplicity during the implementation process. The possibility of simplifying the implementation process and reduce at minimum the effort, can lead a programmer to use the same data structure across the whole pipeline, increasing dramatically the amount of used memory and inducing an artificial memory-bound nature that can be avoided. 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 dramatically improve overall performance.

Given this rationale, in this work we will show how to apply these optimisations by having as a use case *NuChart-II* [2], a tool for Hi-C data analysis that provides a gene-centric view of the chromosomal neighbourhood in a graph-based manner. Starting from the first C++ implementation of this tool, we realised an efficient and optimised version that can also exploit parallelism in the graph exploration. We also propose and discuss issues about the first GPGPU implementation of a kernel for finding chromosome fragments enclosing a single gene.

This paper is organised as follows: Sec. II presents the related work on Hi-C data analysis tool. Sec. III gives a background on graph-based exploration of Hi-C data and presents the parallel algorithm. Sec. IV presents performance issues of this process and an approach to overcome them. Sec. V presents results and performances obtained by different levels of optimisations applied and describes the GPGPU implementation. Finally, in Sec. VI we show final thoughts and also some future work.

II. RELATED WORK

3C-based techniques used to characterise the nuclear organisation of genomes and cell types took the scientific community to the designing of a number of systems biology methods to analyse such data. Particular attention is given to the detection and normalisation of systematic biases. The first step consists generally of Hi-C data processing from raw sequence to contact matrices, in order to detect and normalise biases coming from sequencing and mapping. The second step consists of interpretation and visualisation of corrected data. In this section, a description of tools for Hi-C data analysis is provided.

HOMER [3] is a suite of tools for Hi-C, ChIP-Seq, GRO-Seq, RNA-Seq, DNase-Seq data analysis, based on the creation of contact maps and exploiting Principal Component Analysis and hierarchical clustering. HiTC [4] is a R/Bioconductor package that facilitates the exploration of high-throughput 3C-based data. It provides a set of features to manipulate high-

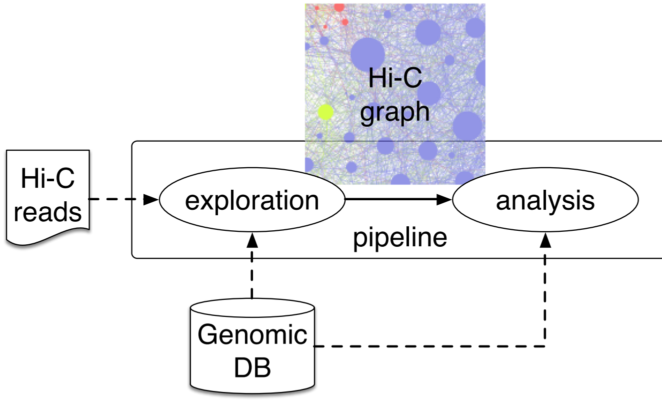

Fig. 1 – *A pipeline schema for Hi-C data processing.*

Algorithm 1 *Sequential BFS procedure*

```
1  BFS(GeneDB : Array<Gene>, HiCReads : Array<Read>, root : Gene) {
2      Set<Gene> V
3      Queue<Gene> Q
4      V <− Q <− {root}
5      while (not Q.empty()) {
6          g <− Q.pop()
7          // find reads ...
8          for_each ({r ∈ HiCReads | r.chr1 = g.chr}) {
9              // find genes ...
10             for_each({g′ ∈ GeneDB | g′.chr = r.chr2}) {
11                 if (not g′.visited) {
12                     Q.push(g′)
13                     V = V ∪ {g′}
14                     g′.visited = true
15                 }
16                 E = E ∪ {(g,g′)}
17             }
18         }
19     }
20 }
```

throughput C data and is suited for visualisation and basic transformations of 5C and Hi-C data. HiBrowse [5] is a web-based analysis server for 3D genome statistical analysis, providing a set of tools based on state-of-the-art statistical methods utilising Monte Carlo and analytic methods, in addition to a range of tools for visualisation and hypothesis-generating investigations. FisHiCal [6] is an R package that performs an iterative FISH-based Hi-C calibration by exploiting 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.

NuChart [7] is an R/Bioconductor package that allows the annotation and statistical analysis of a list of input genes with information relying on Hi-C data, integrating knowledge about genomic features that are involved in the chromosome spatial organisation. NuChart is the first analysis tool that works with the aim of creating gene-centric neighbourhood graphs on which multi-omics features can be mapped. We recall that NuChart-II, the first C++ implementation, is the case study used within this paper.

III. Graph-based Hi-C data analysis

The process of working on Hi-C data for interpretation and visualisation purposes, can be seen as a two-stage pipeline, as shown in fig. 1.

The first stage (the *exploration* phase) takes as input a static database containing genomic data (e.g. human genome) and a file containing Hi-C double-ended reads. Hi-C reads expose the spatial information exploited by the process in order to build topological-oriented structures, such as contact maps and gene-centric graphs. The basic mechanism in the exploration stage consists of looping over Hi-C reads and, for each input read r, searching for couples of genes connected by r. The graph exploration is iterated in a BFS fashion, starting from a gene (the starting node of the graph) until all the nodes have been visited (i.e. fix-point is reached). The output of the exploration stage is a structure representing a relation between genes. In such case, the first stage can be basically regarded as a Breadth-First Search (BFS) and the output is a graph $G = (V, E)$ where V is a set of genes and an edge $e = (g, g') \in E$ is an unordered couple of neighbour genes. An edge exists if there is a single Hi-C read encompassing g and g', thus putting these neighbour genes in a (binary symmetric) relation of spatial neighborhood.

The second stage (the *analysis* phase) takes as input the graph produced by the exploration stage and again the static database of genomic data. In this phase, some statistical processing is applied to the edges, in order to assign a weight to each edge and filter away the noisy ones. Finally, various types of processing can be performed on the graph, ranging from simple visualisation to classical statistical and graph analysis (e.g. connected components and cliques detection), up to clustering methods (e.g. k-means and Quality Threshold) or social network analysis techniques.

A. Memory-intensive parallel BFS graph exploration

The pseudo-code of the BFS is reported in Algorithm 1. Each iteration of the outermost while loop pops an unvisited gene g and explores all the edges departing from g. For each discovered edge, if it connects g to an unvisited gene g', then g' is pushed into the working queue Q, as in any typical graph exploration procedure. Notice that each iteration of the while loop accesses a subset of the Hi-C reads file – namely the reads which first end-point falls in the same chromosome as the one enclosing gene g. Then, for each read r, a subset of (the in-memory copy of) the genomic DB is accessed – namely the genes enclosed in the same chromosome as the one enclosing the second end-point of the read r.

Exploiting the computational capability provided by widespread multicore platforms, possibly paying low programming effort, is an attractive approach for speeding up the pipeline execution. Since each read in the BFS can be processed independently from each other, the graph exploration results into a typical data-parallel procedure, in which any arbitrary subset of reads can be processed independently from each other. Ideally, it can be parallelised in a seamless way just by taking the kernel of the procedure (lines 9–17 in the pseudo-code) and putting it into a *ParallelFor* loop pattern, which semantics amounts to execute in parallel the iterations

Algorithm 2 *Parallel BFS procedure*

```
1  BFS(GeneDB : Array<Gene>, HiCReads : Array<Read>, root : Gene) {
2      int level <− 0
3      Set<Gene> V
4      Queue<Gene> Q
5      V <− Q <− {root}
6      while (not Q.empty()) {
7          // explore reads
8          T <− {} // set of tasks
9          while(not Q.empty()) {
10             g <− Q.pop()
11             // find reads ...
12             for_each ({r ∈ HiCReads | r.chr1 = g.chr})
13                 T.push((g,r))
14         }
15
16         //works on each read in parallel
17         ParallelFor({(g,r) ∈ T}, NTHREADS) {
18             for_each({g' ∈ GeneDB | g'.chr = r.chr2}) {
19                 if(not Bitmap[g']) {
20                     localQ[tid].push(g')
21                     localV[tid] = V[tid] ∪ {g'}
22                     localBitmap[tid][g'] = level
23                 }
24                 E[tid] = E[tid] ∪ {(g,g')}
25             }
26         }
27
28         //reduce
29         for_each({0 ≤ tid < NTHREADS}) {
30             Q = Q ∪ localQ[tid]
31             V = V ∪ localV[tid]
32             Bitmap.merge(localBitmap[tid])
33         }
34
35         // reset local structures
36         ++level
37     }
38 }
```

of the loop, provided they are independent from each other. High-level parallelisation of BFS exploration has been treated, among others, in [8].

Nevertheless, following this high-level approach requires some adjustment to the structure of the BFS procedure. As discussed in [8], BFS exploration should be organised in a level-synchronised way. Input genes at level i correspond to the set genes discovered at level $i-1$, taking the root gene as the input at level 0. Moreover, 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.

The pseudo-code of a parallel BFS is reported in Algorithm 2. Here we applied a first simple optimisation by splitting the reads exploration phase (lines 8–14) and the core ParallelFor (lines 17–26). This approach aims to avoid mixing the working sets of the two phases, thus minimising cache thrashing. The reduce phase (lines 29–33) merges thread-local structures into per-level global ones, including a Bitmap array containing, for each gene in the DB, the level at which the gene has been visited by the BFS procedure. Notice that in the proposed pseudo-code the reduce phase is executed sequentially, after the end of each per-level ParallelFor instance. Obviously a further optimisation could consist in designing a parallel reduce and overlapping its execution with the ParallelFor phase.

Under the performance analysis perspective, we remark that, in the proposed schema, each iteration of the ParallelFor works on a subset of the DB in order to integrate information related to genes encompassed by each read end-point. Since the proposed approach works with an in-memory copy of the DB, this amounts to transfer all the interaction traffic to the memory system. This interaction schema, further exacerbated when running the application in a multithread fashion on a shared memory platform, can clearly induce heavy traffic on memory system, thus obtaining a memory bound behaviour. These aspects will be discussed in section IV.

B. Running Example: NuChart-II

In this work, we use as test-bed the NuChart-II tool, which is basically a C++ implementation of NuChart, a R/Bioconductor package working on Hi-C data with a gene-centric graph-based approach. NuChart-II, in the first C++ implementation, showed inevitable problems of scalability while working genome-wide on large datasets [2].

The pipeline in fig. 1 can be considered as representative of the NuChart approach to the Hi-C data analysis. In this work we consider the optimisation related only to the first stage, that is, the graph construction and exploration. We implemented the ideas discussed above, producing a naive-parallel version of NuChart-II, based on the *ParallelFor* pattern provided by the FastFlow skeleton library [9], [10].

FastFlow is a C++ based parallel programming framework built on top of POSIX threads aimed at providing the parallel programmer with a set of pre-defined algorithmic skeletons modelling the main stream-parallel patterns. It provides a set of high-level, parallel programming patterns, called *algorithmic skeletons*, obtained by the composition of two basic algorithmic skeletons: a *farm* skeleton, and a *pipeline* skeleton. FastFlow provides also a *ParallelFor* skeleton [11] aimed at filling the usability and expressiveness gap between the classical data parallel skeleton approach and the loop parallelisation facilities offered by frameworks such as OpenMP [12] and TBB [13].

IV. BFS: A MEMORY-OPTIMISED APPROACH

User-defined structures (or classes) are widely used for describing complex data and nowadays it is very common to gather several (related) elements in a single data type. This logical organisation also reflects on how these elements will be mapped into the physical memory and this - ideally - should not affect the data access performance. However, current architectures are highly optimised for contiguous memory access and, therefore, extra care should be taken especially when dealing with arrays of user-defined structures.

The initial BFS implementation relied on existing data structures, containing additional fields that were used in other phases but that were not accessed in this specific stage. At first glance, this approach might not seem to harm the overall performance - as the extra data is not explicitly loaded - but the

Algorithm 3 *Standard struct definition for a Fragment*

```
1 struct Fragment{
2   long start , end;
3   string id;
4   long r1, r3, r5;
5 };
```

Algorithm 4 *Struct definition for describing Fragment positions*

```
1 struct FragmentPosition{
2   long start , end;
3 };
```

actual results show clearly a performance degradation. What really happens in the background is the loading of (lots of) unaccessed data into the cache due to proximity. This overhead can actually saturate the memory bus making it nearly impossible to exploit multiple processors in a multicore system, even in the case of an embarrassingly parallel application.

This memory problem is very difficult to deal when the access pattern is completely random, however, in the BFS we are dealing mostly with linear access, and in this case it is possible to optimise the memory bandwidth utilisation.

The proposed approach in this work consists in: (1) identifying the memory intensive parts of the program, (2) creating structures that define the subset of variables that are used in each stage and (3) duplicating the data using a specific data structure for each stage. Reducing the working set is the key for improving the memory bandwidth usage - less unused data loaded into the cache translates into less cache misses. The choice of data duplication is preferred in this case because it can be easily implemented without breaking the application logic: since the duplicated data has read-only semantics, the original data structure can still be used in other parts of the code. In the case where the duplication is not affordable, it is also possible to optimise data structures but at the price of making the software more complex and modifying all the source code where the original structure is used.

In order to illustrate this approach when dealing with a memory-optimised implementation of the NuChart-II tool, the original struct definition for a chromosome fragment is given in Algorithm 3. Suppose we have a huge array of elements of type Fragments and we wish to retrieve some information of those who match a given criteria (for example based on the `start` and `end` variables). The most straightforward way to implement this is using a `for loop` that will access the variables `start` and `end` of all array elements and, for the elements matching the criteria, the program will perform further accesses on the other struct variables.

In the case where most of the fragments match the criteria, this approach might yield an acceptable performance. However, in cases were only a few elements match, many values will be loaded into the cache even though they are not going to be accessed by the program. In most of the current architectures, the cached memory access is optimised to contiguous consecutive access and the smallest amount of data that can be fetched is usually larger than one primitive variable. In our example, explicitly reading `start` typically translates to automatically loading into the cache some (or maybe all) consecutive variables from the same element.

There are several possible ways to avoid this waste of memory bandwidth, here we propose a solution that minimises the modifications on the existing code, by creating an additional structure that describes the fragment positions (Algorithm 4). Since all the data in this struct are contiguous and 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).

This kind of optimisations can have a huge impact on the overall application performance, but, currently, it has to be applied manually and it may require domain-specific understanding of the problem.

V. Experimental Results

In this section, we will compare performances obtained by the different levels of optimisations applied to the parallel BFS graph exploration, part of the first stage (the exploration phase) of the pipeline shown in fig. 1, running the application using up to 32 threads. NuChart-II has been tested on an Intel workstation, equipped with 4 eight-core E7-4820 Nehalem (64 HyperThreads) @2.0GHz with 18MB L3 cache and 64 GBytes of main memory running Linux CentOS 6.5 x86_64. The Nehalem processor uses HyperThreading 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. Thanks to the internal structure of the FastFlow *ParallelFor*, it is possible to use all physical cores while thread pinning is automatically managed by the FastFlow library. The used dataset is LiebermanAiden et al. Hi-C data (SRA:SRR027963) [1] with TP53 as starting gene for the graph exploration. Considering that the application is tested on a NUMA (non-uniform memory access) platform, we executed NuChart-II also using an interleaved memory allocation policy via *numactl* linux utility, which can be used to control NUMA policy for processes or shared memory.

A. Naive approach

Results here exposed are related to performances obtained with the implementation having no working set optimisations, that is, without the duplication of only accessed data for the BFS exploration phase. In fig. 2 it is reported the maximum speedup obtained by the execution of NuChart-II with interleaved and default memory allocation policy. The memory-intensive nature induced by this implementation, invalidates all attempts to gain performances by exploiting parallelism. One of the easiest optimisation techniques that could be useful to increase the memory bandwidth utilisation, consists in applying an interleaved allocation policy. The interleaved policy permits to allocate memory pages in a Round-Robin fashion over all nodes in the system. This allocation strategy usually leads to some advantage in terms of performances because of spreading the memory load across memory nodes, thus preventing a single memory node to become the bottleneck for the memory traffic. As a first result, we can state that despite in this case the gain is minimal and negligible, more

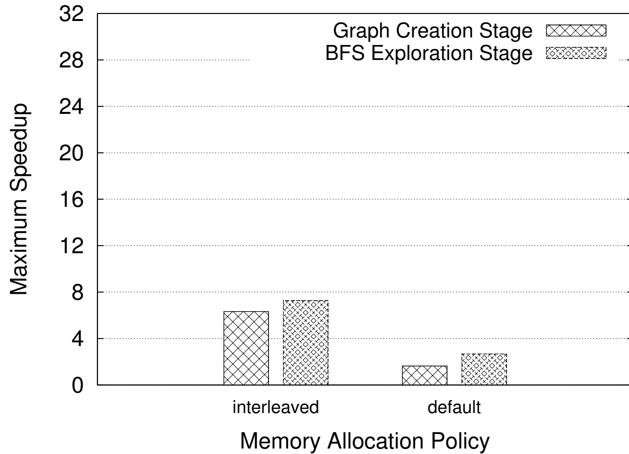

Fig. 2 – *Naive implementation of NuChart. The maximum speedup obtained by the parallel execution of the graph creation and BFS exploration phases is reported.*

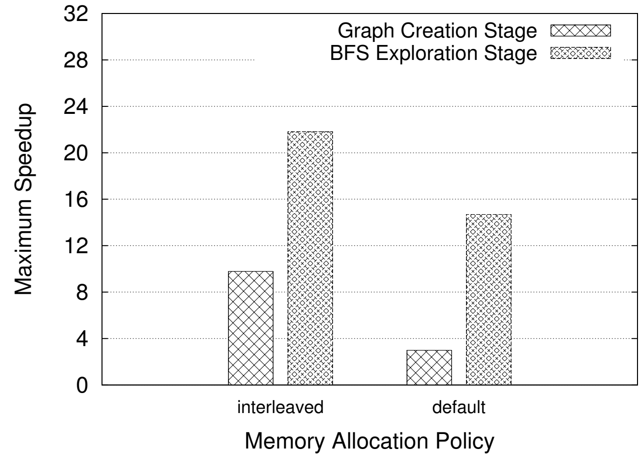

Fig. 3 – *Memory-optimised implementation of NuChart. The maximum speedup obtained by the parallel execution of the graph creation and BFS exploration phases is reported.*

effort should be applied on the memory-optimised approach discussed above.

TABLE I – *Execution times (seconds) for the naive NuChart-II implementation*

	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 I refers to execution times obtained by the naive 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, until the fix point is reached, as explained in section III-A. At each level, the number of nodes reached is highly unbalanced, taking to have very different execution times during the BFS exploration.

B. Memory-optimised approach

In this paragraph, we show performances obtained by the execution of the optimised implementation. We recall that the working set utilised in the graph exploration has been reduced by the replication of the subset of data needed by this stage.

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 and, thus, speeding up the computation.

Table II refers to execution times obtained by 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 twice faster compared to the sequential execution of the naive implementation (1006 seconds vs 570 and 962 seconds vs 566), 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.

TABLE II – *Execution times (seconds) for the memory-optimised NuChart implementation*

	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

C. Implementation on GPGPUs

The usage of GPGPUs for implementing the BFS is possible, however it is not straightforward due to the dynamic nature of this application (graph creation and `for` loops containing `break` statement). GPGPUs can handle very well data-parallel applications, but have limited options when it comes to dynamic memory allocation on the device.

As a slight variant of the memory-optimised NuChart-II tool, a preliminary GPGPU version has been designed by implementing a kernel for finding chromosome fragments enclosing a single gene. The original CPU code consists in a `for` that breaks when the fragment position matches. On GPUs, however, it is not possible to implement the very same behaviour as a parallel kernel: there is no guarantee on the order which the kernels are scheduled on the available cores and also there is no efficient mechanism to stop remaining threads when one thread finds a positive match.

One possible solution to address this problem is to divide the search space into smaller parts and to offload sequentially each chunk to the GPU: if a match is found, the research can be stopped making it possible to avoid exploring all the search space (minimising the waste of GPU resources). Yet, there is a clear trade off concerning the parallel performance when using this solution: the chunk size has to be carefully chosen as it impacts on the number of synchronous calls.

This mechanism has been implemented and, after tuning the chunk size, a gain of 10% on the overall performance has been observed on a NVidia GPU K40 (compared to a sequential CPU version). This is an ongoing work, the results show the feasibility of a GPGPU version and, even if the performance gain is quite limited, it represents only a porting of a small part of the program and there are still a great portion of code that can be implemented using GPGPUs.

VI. Conclusion

In this work, we addressed the problem of the optimisation of data structures and memory allocation in order to exploit parallelism in a Hi-C data analysis application. In particular, we used as running example the NuChart-II tool, a C++ tool working on Hi-C data with a gene-centric graph-based approach. NuChart-II applies this approach by generating and visiting a graph in which the exploration stage consists of looping over Hi-C reads and searching for couples of genes connected by them. The graph exploration is iterated in a BFS fashion, starting from a gene until all the nodes have been visited. We started from a naive implementation of NuChart-II, in which the BSF exploration was based on existing data structures, containing fields not useful for the exploration. By keeping these fields during the exploration, we showed that such extra data lead to overall performance degradation. This is due to the loading of not accessed data into the cache, that can saturate the memory bus, thus making it clearly difficult to exploit multicore systems. Given this rationale, we succeeded to minimise this performance degradation by reducing the working set: ad-hoc data structures were created, in which only the data effectively used in the BFS stage phase are copied. The consequence of this optimisation is that there is an immediate improvement of the memory bandwidth and cache utilisation. In addition to that, we used also an interleaved allocation policy of all data structures, thus incrementing the memory bandwidth utilisation. Comparing to the naive implementation of NuChart-II and the optimised implementation, we obtained a maximum speedup of ~ 22 starting from a speedup of ~ 7. As a future optimisation, it could be helpful to try to allocate data structures in order to provide threads with memory affinity and, hence, to get the best memory latency in accessing effectively used data structures.

VII. Acknowledgments

This work has been partially supported by the EC-FP7 STREP project Paraphrase (no. 288570), the EC-FP7 STREP project REPARA (no. 609666) and the Fondazione San Paolo IMPACT project (ID. ORTO11TPXK).

References

[1] 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.

[2] F. Tordini, M. Drocco, M. Aldinucci, P. Liò, L. Milanesi, and I. Merelli, "NuChart-II: a graph-based approach for the analysis and interpretation of Hi-C data," in *Proc. of the 11th Intl. meeting on Computational Intelligence methods for Bionformatics and Biostatistics (CIBB)*, Cambridge, UK, Jun. 2014.

[3] 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, Dec. 2013.

[4] 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, Nov. 2012.

[5] J. Paulsen, G. K. Sandve, S. Gundersen, T. G. Lien, K. Trengereid, and E. Hovig, "Hibrowse: Multi-purpose statistical analysis of genome-wide chromatin 3d organization." *Bioinformatics*, 2014.

[6] Y. Shavit, F. Hamey, and P. Lió, "FisHiCal: an R package for iterative FISH-based calibration of Hi-C data," *Bioinformatics*, vol. 30, no. 18, 2014.

[7] I. Merelli, P. Li, and L. Milanesi, "NuChart: An R Package to Study Gene Spatial Neighbourhoods with Multi-Omics Annotations," *PLoS ONE*, vol. 8, no. 9, 2013.

[8] 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. IEEE Computer Society, 2011, pp. 78–88.

[9] 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*. Pisa, Italy: IEEE, feb 2010, pp. 195–199.

[10] 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.

[11] 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.

[12] 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.

[13] "Intel Threading Building Blocks, project site," 2014, http://threadingbuildingblocks.org.