TOWARDS HIGH-PERFORMANCE HAPLOTYPE ASSEMBLY FOR FUTURE SEQUENCING

Marco Aldinucci[(1)], Andrea Bracciali[(2)], Nadia Pisanti[(3)], Massimo Torquati[(3)]

(1) Turin University
Computer Science Department, aldinuc@di.unito.it

(2) Stirling University
Computer Science and Mathematics, abb@cs.stir.ac.uk

(3) Pisa University
Computer Science Department, {pisanti,torquati}@di.unipi.it

Keywords: Haplotype, minimum error correction, multi-core, parallel programming.

Abstract. The problem of *Haplotype Assembly* is an essential step in human genome analysis. Being the well known MEC model for its solution NP-hard, it is currently addressed by using algorithms that grow exponentially with the length of DNA fragments obtained by the sequencing process. Technological improvements will reduce fragmentation, increase fragment length and make such computational costs worst. WHATSHAP is a recently proposed novel approach which moves complexity from fragment length to fragment sovrapposition, improving the perspective of computational costs, but Haplotype Assembly still remains a demanding computational problem. Directions towards high-performance computing Haplotype Assembly for future sequencing, based on parallel WHATSHAP, are discussed in this paper.

1 Scientific Background

Human genome is *diploid*, i.e. each chromosome comes in two copies, each of which is a *haploid* chromosome coming from one of the two parents (one *allele* per parent). *Single Nucleotide Polymorphisms* (SNPs) are single DNA positions in a chromosome where the nucleotide can differ in distinct individuals, e.g. in parents, and therefore be different in the two DNA copies of a single individual.

Haplotyping is the task of phasing the SNPs, i.e., assigning their values to either of the two DNA copies (alleles) inherited from parents. Genomic data obtained from a sequencing experiment is a mixture of the two copies of the chromosomes in the form of many DNA fragments coming from either of the two alleles, called *reads*, which have not been assembled yet into contiguous sequences of whole chromosomes. When SNPs phasing is performed directly on such raw sequencing reads, we talk about *haplotype assembly*: each read is assigned to one of the two alleles. Therefore, reads that exhibit different values on the same SNP position must necessarily belong to different alleles.

Arbitrarily re-labelling the alleles with 0 and 1, the input data can be represented as a $n \times m$ matrix F, with n the number of reads and m the number of SNPs sites. The i-th read (i-th row of F) is represented with a string in the alphabet $\{0, 1, -\}$ where a value 0 (resp. 1) at column j tells that the i-th read has the value of allele 0 (resp. 1). The value $-$ means that the read does not cover the j^{th} SNP position.

A *conflict* is an SNP position where two reads r_p and r_q have different values (that is, a 0 and a 1). Reads that have distinct allele values at a common SNP are assumed to come from different chromosome copies. A correct haplotype assembly corresponds to a bipartition of the rows of F into two sets F_0 and F_1 such that each one of these two sets is *conflict free*. Unfortunately, due to sequencing errors, in real data such bipartition

does not exist. The problem thus becomes that of detecting a minimal amount of errors to be corrected (or removed) in order to have a conflict free bipartition.

In literature, there are several models for the haplotype assembly problem corresponding to different optimization problems: *Minimal Error Correction* (MEC) corrects the minimum number of errors, by turning 0s into 1s or viceversa; *Minimal Error Removal* (MER) removes the minimum number of errors, by turning 0s or 1s into $-$s; *Minimal Fragment Removal* (MFR) removes the minimum number of conflicting fragments. We focus on detecting sequencing errors (and not mapping errors, i.e. erroneous assignments of a read to a position in the genome), thus concentrating on MEC and MER. The two have been proved to be equivalent, and actually both can be reduced to finding a MAX-CUT in a graph and therefore are NP-hard.

Several proposals and tools have been put forward to solve MEC in the last ten years, such as [11, 8] based on a greedy heuristic to assemble the haplotype of a genome, [5] a method to sample a set of likely haplotypes under the MEC model, and faster follow-up based on the definition of a graph [6], and an iterative greedy heuristic to optimize the MAX-CUT of that graph [4]. The latter outperforms [11, 8, 5] and shows similar accuracy to [5]. Other reductions of MEC to MAX-SAT are in [10, 7].

Since the problem is NP-hard, all practical solutions to MEC are either statistical/heuristics approaches, or are *exact fixed-parameter tractable algorithms*, in which case complexity turns out to be exponential in the number of SNPs per read or in the read length. Due to the way in which sequencing biotechnologies evolve providing ever-increasing read length, methods with fixed parameter tractability exponentially linked to read length (or to the number of SNPs per read, which also grows with read length) will perform worse and worse with future-generation longer reads.

In [12], some of the authors introduced WHATSHAP, the first exact fixed-parameter tractable algorithm for solving MEC that, importantly, is exponential in the *sequencing coverage*. This parameter is the maximum number of different reads that cover a single SNP position. WHATSHAP results to be quite accurate, due to the fact that it actually solves wMEC, a weighted generalisation of MEC in which a *confidence degree* is associated to each 0 and 1 and less confident values are the most likely to be corrected.

WHATSHAP is still a computationally demanding algorithm. For instance, experiments with a coverage limited to up to $20\times$ can be managed with a time cost in the order of the hour on a *single* core of a standard desktop machine. It is worth remarking that higher coverages may occur in practice and even small increases may have substantial impact. Interestingly for future perspectives, datasets with higher coverage are desirable since they could further improve the accuracy of WHATSHAP.

Considering also that the analysis of a whole genome may require the solution of several (a few tens) independent instances of haplotype assembly, it is clearly worth exploring the possibility of a parallel version of WHATSHAP.

2 Materials and Methods

2.1 WHATSHAP

WHATSHAP takes as input the *fragment table* F, one row per *read* (a DNA *fragment*) and one column per SNP position with values in $\{0, 1, -\}$, and computes minimum-cost conflict-free partitions of the set of reads F. WHATSHAP, in a dynamic programming style, incrementally builds a *cost matrix* C with as many columns as F (i.e., again, one column per SNP position). Column j of C will contain all the possible bipartitions of the reads and their associated (minimal) cost for making them conflict-free. Such a cost will be determined by considering the information available up to column j of F. More precisely, given the set F_j of all *active* reads that covers the j^{th} position in F (that is, with either a 0 or a 1 but not a $-$), WHATSHAP computes the minimum cost of all the

posible bipartitions (R, S) of F_j and the costs associated to make such a bipartitions conflict-free, written $C(j, (R, S))$.

In general, a read spanning over several consecutive positions will induce constraints on the possible partitions, as it must be consistently assigned to the same allele throughout all the positions on which it is active in F, and this may require corrections. Therefore, when computing the cost of the partitions of F_j at column j, the costs paid for "compatible" partitions of F_{j-1} must be taken into account.

Entries in the first column of C have the form $C(1, (R, S))$, with (R, S) a bipartition of F_1. In this case, the cost of (R, S) only depends on making R and S conflict free with respect to the first column of F. $R \subseteq F_1$ can be made conflict free by flipping all 0s into 1s, at a cost that is equal to the sum of all the weights associated to the 0s that must be flipped, denoted as $W(1)^1_R$, or by flipping all 1s into 0s, paying $W(1)^0_R$. Summing up (taking the most advantageous alternative),

$$C(1, (R, S)) = min\{W(1)^1_R, W(1)^0_R\} + min\{W(1)^1_S, W(1)^0_S\}.$$

When considering the j^{th} column, both the contribution of the column itself (computed in the same way as in the first column), and the cost that any specific bipartition *inherites* from previous columns must be taken into account.

Consider, for instance, $C(j, (R, S))$, with $j > 1$ and (R, S) a bipartition of F_j. The local contribution of column j is, again, just the cost of the best way to make R and S conflict free over the column j of F (first row in the formula below).

To this cost, the cost of keeping (R, S) consistent on all the columns $i < j$ has to be added. This cost is the minimal cost of $C(j-1, (R', S'))$, for any (R', S') which is "compatible" with (R, S). A partition (R, S) defined at j and one (R', S') defined at $j-1$ are *compatible*, written $(R, S) \cong (R', S')$, if each element in $F_j \cap F_{j-1}$, i.e. the reads active in both j and $j-1$, is assigned to the same subset in both (R, S) and (R', S'). It is important to note that, because of the incremental way of proceeding, the cost in the immediately preceeding column $j-1$ summarises all the corrections made in columns 1 to $j-1$ for keeping (R', S') error free. Summing up,

$$\begin{aligned} C(j, (R, S)) \;\; = \;\; & min\{W(j)^1_R, W(j)^0_R\} + min\{W(j)^1_S, W(j)^0_S\} + \\ & min_{(R',S') \cong (R,S)} \;\; C(j-1, (R, S)) \end{aligned}$$

The schema of the generic j^{th} step of the algorithm consists in *defining all the possible* (R, S) at j and then performing the following three steps:

(a) determine the minimal "local" cost for making the j^{th} column conflict-free by flipping some bits on the column according to their weights and possible corrections;

(b) select the minimal-cost partition amongst those computed at step/column $j-1$ which are compatible with the chosen partition making conflict-free the j^{th} column;

(c) determine the total cost for the current partition as the sum of the two mimima from the previous two points.

2.2 High Performance Computing: methodological aspects

After a long period of hardware evolution based on boosting single-core chip performances by increasing clock-frequency and instruction-level parallelism, the focus now has shifted to increasing cores per chip, while preserving power consumption. However, sequential code gets no or very small performance benefit from new multi-core chips, which typically have lower single-core complexity and clock. Since new multi-core platforms are de-facto small-scale on-chip parallel machines, performances can only be increased by using thread-level parallelism, but writing parallel programs is inherently more difficult than writing sequential code, and developers, including bioinformatics scientists, need to mange the trade-off between high-end performances and time to solution. Parallel software engineering adopted high-level sequential language extensions

and coding patterns aimed at simplifying the porting of sequential code to parallel architecture and guaranteeing performances [3]. *Parallel design patterns* [9] have been recognised to be able to support the exploitation of current highly parallel architectures and make programming simpler and more efficient. They provide tested and efficient parallelism exploitation patterns as composable building blocks, which are at an higher level of abstraction than traditional approaches, MPI say, where the programmer is fully responsible for parallelism.

The FastFlow framework [1, 2] provides parallel design patterns suitable to support performance and code reuse for a parallel implementation of WHATSHAP on a multi-core architecture. Importantly from the methodological viewpoint, minimal changes to the original sequential code are required. Technically, FastFlow allows *read-after-write* dependencies to be synchronised in shared data structures, hence avoiding the usage of the typically much slower classical mutual exclusion mechanisms. In these settings, a thread that receives a task is allowed to write the data of that specific task, e.g., the data of the current column, and read all other shared data structures, e.g. the data of the previous column. A FastFlow implementation of WHATSHAP will be discussed in the next section. It is interesting to mention that FastFlow can uniformly be used to support the parallel execution of the multiple instances of haplotype assembly needed for a whole genome, which are fully independent and can be best concurrently executed in an *embarrassingly parallel* fashion on truly independent platforms, such as, distributed clusters or cloud infrastructures, with no performance degradation due to the concurrent usage of resources, which instead may happen on multi-core architectures. The FastFlow *task-farm* parallel pattern, which will be discussed for multi-core WHATSHAP in the next section, can also be used to support multiple instances of parallel WHATSHAP on a cloud infrastructure, which is considered to be an enabling technology for bioinformatics and computational biology, since it can provide a large amount of computing power and storage in an elastic and on-demand fashion.

3 Results

Several aspects must be considered when attempting to solve Haplotype Assembly by means of a *parallel* WHATSHAP. We will here focus on considering the reads of a single chromosome, recalling that different chromosome induce completely independent instances of the problem, whose solution can be trivially computed in parallel. We consider a multi-core architecture, where a parallel WHATSHAP can exploit the physical shared memory and avoid to move data between threads, a typical source of overhead. This, greatly simplifies the porting of the sequential to the parallel version, but also introduces new problems related to data sharing.

Which tasks carried out by WHATSHAP can be and are worth of being parallelised in these settings? WHATSHAP *traverses* a big matrix by sequentially exploring its columns, and for each column *generates* a cost matrix through a costly subset construction. In order to parallelise the algorithm, it is important to clearly understand data dependencies.

Parallelising the traversal of the matrix, i.e. assigning to different executors the traversal of a subsets of the columns of the matrix, would be a natural and potentially very effective parallelisation, but the data dependency of each column on the previous one makes this parallelisation not so straightforward. This is currently under study.

On the other hand, the cost matrix generation phase clearly consists of independent activities which, within the same column, do not require synchronisation. Both the computation of the minimal local cost for making the j^{th} column of the F table non-conflicting (see step (a) on page 3) and the selection of the minimal-cost among those computed at the previous $(j-1)^{th}$ column (step (b)) are independent over all possible bipartitions (R, S) of F_j. Each bipartition (R, S) can theoretically be run in parallel.

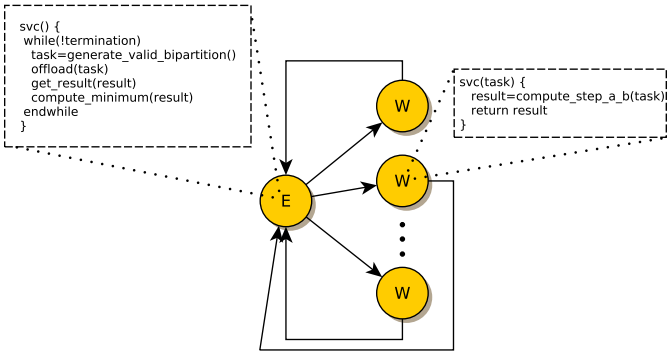

Figure 1: The *task-farm-with-feedback* skeleton of FastFlow. Each entity is a concurrent thread. The Emitter thread (*E*) produces and schedules tasks towards a pool of Workers threads (*W*).

In order to ascertain the actual viability of such *fine-grained* parallelisation, some profiling of the application has been carried out on a modern multi-core workstation (2 CPUs Intel Xeon E5-2695 @2.40GHz, 64GB RAM) running Linux CentOS 6.5 x86_64 and using a version of Venter's *chromosome one* data, where coverage has been limited to $20\times$. WHATSHAP terminated in about half an hour.

Exponential times in the coverage of a column (the number of active reads on that column) have been observed quite regularly. Columns with a coverage up to $16\times$ require a maximum time of $1.8ms$ to be processed, columns with a coverage of $16-18\times$ require a maximum of $7.5ms$, columns with a coverage of $19-20\times$ can be processed with times in the interval $14.5-31ms$. This profiling clearly shows the exponential complexity of the algorithm. It is worth remarking that $20\times$ is an imposed constraint on the dataset. Due to overheads, only columns with a coverage of $19-20\times$ and times of about $20-30ms$ are worth being parallelised. It is interesting to observe the distribution of the above classes of columns: $1-16\times$ columns are about 17% of the total, $16-18\times$ are 15% and $19-20\times$ are 68%. These data shows that about 70% of the computational cost of WHATSHAP can anyway benefit from the discussed fine-grained parallelisation. Importantly, the gain will improve with larger coverages, which are expected following the current technological trends, and desirable in order to improve accuracy.

The FastFlow *task-farm-with-feedback* pattern (Fig.1) provides the needed abstraction for implementing the proposed parallelisation. It consists of a sequential stage called Emitter (*E*), which emits and schedule the tasks, and a stage containing the pool of Worker threads (*W*), each one executing the tasks provided by the Emitter. In the *task-farm-with-feedback*, the Workers send the results back to the Emitter. It is possible to exploit both Emitter–Workers *pipeline* parallelism and parallelism among Workers. The sequential WHATSHAP is so split in three concurrent parts:

1. the computation of all possible bipartitions for each given column, which is computed by the Emitter thread;

2. the computation of the local minimum-cost in both the step (a) and (b) (see page 3), computed by the Worker threads;

3. the computation of the overall minimum cost for each bipartition, computed by the Emitter.

As soon as one possible bipartition has been computed, a task containing the reference to that bipartition is sent (offloaded) to one of the available workers using a suitable scheduling policy (the framework allows us to use one of the predefined scheduling policies or to define a new one). The worker W_i receiving the task (containing one or even more possible bipartitions just computed by the Emitter) computes the local

minimal costs and sends them back to the Emitter. The Emitter after having received all results from the workers, executes a local *reduce*, a result collection phase, to find the overall minimum cost for the partition under consideration.

The proposed parallelisation is quite direct and, importantly, requires minimal changes to the sequential code. Furthermore, this is a typical case in which one can obtain the best results because of the high-degree of the parallelisation, which involves the many entries of the large *fragment table* F, corresponding to many (small) tasks that can be executed in parallel on the available cores. As a further enhancement, it is also possible to (partially) overlap in a three stage pipeline *i)* the time spent for generating all possible valid bipartitions, *ii)* the time spent for computing both the steps (a) and (b) and *iii)* the time for computing the total cost of the current partition (step (c)).

4 Conclusions

WHATSHAP is an interesting approach for the solution of the haplotype assembly problem in genome sequencing. This algorithms appears particularly interesting in the light of future-generation sequencing technologies capable of managing longer DNA fragments, which will cause the complexity of traditionally used algorithms to explode.

However, the intrinsic complexity of the problem and the increasingly high demand for sequencing call for more efficient apporaches. We believe that these will be supported by the provision of multi-core programming. We have identified FastFlow as a particularly suitable framework for a parallel WHATSHAP and proposed and analysed a possible parallel implementation.

References

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, March 2014.

[2] M. Aldinucci, M. Torquati, C. Spampinato, M. Drocco, C. Misale, C. Calcagno, and M. Coppo. Parallel stochastic systems biology in the cloud. *Briefings in Bioinformatics*, June 2013.

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

[4] V. Bansal and V. Bafna. HapCUT: an efficient and accurate algorithm for the haplotype assembly problem. *Bioinformatics*, 24(16):i153–159, 2008.

[5] V. Bansal, A.L. Halpern, N. Axelrod, and V. Bafna. An MCMC algorithm for haplotype assembly from whole-genome sequence data. *Genome Research*, 18(8):1336–1346, 2008.

[6] R. Cilibrasi, L. van Iersel, S. Kelk, and J. Tromp. On the complexity of several haplotyping problems. In R. Casadio and G. Myers, editors, *Proceedings of the Fifth International Workshop on Algorithms in Bioinformatics (WABI)*, volume 3692 of *Lecture Notes in Computer Science*, pages 128–139, Berlin, 2005. Springer.

[7] D. He, A. Choi, K. Pipatsrisawat, A. Darwiche, and E. Eskin. Optimal algorithms for haplotype assembly from whole-genome sequence data. *Bioinformatics*, 26(12):i183–i190, 2010.

[8] S. Levy *et al.* The diploid genome sequence of an individual human. *PLoS Bio*, 2007.

[9] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[10] S.R. Mousavi, M. Mirabolghasemi, N. Bargesteh, and M. Talebi. Effective haplotype assembly via maximum Boolean satisfiablility. *Biochemical and biophysical research communications*, 404(2):593–598, 2011.

[11] A. Panconesi and M. Sozio. Fast hare: a fast heuristic for the single individual SNP haplotype reconstruction. In I. Jonassen and J. Kim, editors, *Proceedings of the Fourth International Workshop on Algorithms in Bioinformatics (WABI)*, volume 3240 of *Lecture Notes in Computer Science*, pages 266–277, Berlin, 2004. Springer.

[12] M. Patterson, T. Marschall, N. Pisanti, L. van Iersel, L. Stougie, G. W. Klau, and A. Schönhuth. Whatshap: Haplotype assembly for future-generation sequencing reads. In *RECOMB*, pages 237–249, 2014.