High-Performance Haplotype Assembly
Marco Aldinucci, Andrea Bracciali, Tobias Marschall, Murray Patterson, Nadia Pisanti, Massimo Torquati

To cite this version:
Marco Aldinucci, Andrea Bracciali, Tobias Marschall, Murray Patterson, Nadia Pisanti, et al.. High-Performance Haplotype Assembly. Computational Intelligence Methods for Bioinformatics and Biostatistics (CIBB’15), 2015, Cambridge, United Kingdom. pp.281 - 258, 10.1016/j.compbiolchem.2005.05.001. hal-01526652

HAL Id: hal-01526652
https://hal.inria.fr/hal-01526652
Submitted on 23 May 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract. The problem of Haplotyping Assembly is an essential step in human genome analysis. It is typically formalised as the Minimum Error Correction (MEC) problem which is NP-hard. MEC has been approached using heuristics, integer linear programming, and fixed-parameter tractability (FPT), including approaches whose runtime is exponential in the length of the DNA fragments obtained by the sequencing process. Technological improvements are currently increasing fragment length, which drastically elevates computational costs for such methods. We present pWHS, a multi-core parallelisation of WHS, a recent FPT optimal approach to MEC. WHS moves complexity from fragment length to fragment overlap and is hence of particular interest when considering sequencing technology’s current trends. pWHS further improves the efficiency in solving the MEC problem, as shown by experiments performed on datasets with high coverage.

1 Introduction

The differences among genomes of distinct individuals of the same species are called polymorphisms. Given two DNA sequences, a Single Nucleotide Polymorphism (SNP) is a variation of a single nucleotide occurring at a specific position in the two sequences. SNPs may occur in genomes of different individuals of the same species or in different copies of chromosomes of the same individual. The different forms that a chromosome may exhibit are called alleles. The Human genome consists of two copies of each chromosome, i.e. it is diploid. Each copy comes from one of the two parents.

Genomic data obtained from a sequencing experiment of a human genome is a mixture of the two copies of the chromosomes in the form of many DNA fragments, called reads, which may exhibit one of the forms, i.e. alleles, of parental chromosomes. Haplotyping is the task of phasing the SNPs, i.e., determining which one of the two alleles they come from.

Haplotyping is an essential task for genome annotation and for several kinds of downstream (comparative) genome analyses, such as finding patterns in human genetic variations for population genomics, or associating genetic variants to diseases, response to drugs, and environmental effects.
When SNP phasing is performed directly on raw sequencing reads, it is referred to as haplotype assembly or read-based phasing. In this case, reads are first mapped to a reference genome and are then assigned to one of the two haplotypes based on the SNPs they cover. For each SNP position, reads that indicate different alleles must be assigned to different haplotypes. The result is a partition of the reads in two classes according to their originating haplotype. Unfortunately, in real data, such a partition may not exist, due to sequencing errors and also due to reads being misplaced in the mapping phase. For this reason, the task of haplotype assembly becomes a computational optimisation problem where one has to minimise the number of adjustments to the data needed to define a bipartition that is then a candidate to represent the correct haplotypes. In the literature, several optimisation problems that formalise haplotype assembly are considered. Minimum Fragment Removal (MFR) removes the minimum number of conflicting fragments and hence focuses on mapping errors (that is, misplaced reads in the mapping phase). Minimum Error Correction (MEC), asks for the minimum number of characters (nucleotides) to be corrected in the input reads. Minimum Error Removal (MER) removes a minimum number of characters from the reads, where removed characters are handled as if the read would not cover these positions at all. MEC and MER have been proved to be equivalent [11], and, since they can be reduced from MAX-CUT [9], are NP-hard. Published algorithms to solve MEC include statistical/heuristic approaches, integer linear programming, or are exact fixed-parameter tractable algorithms [10], whose complexity is exponential in the number of SNPs per read. Due to ever-increasing read lengths, leading to more SNPs per read, provided by evolving sequencing biotechnologies, methods that are exponential in the read length will perform worse with future-generation longer reads.

In this paper, we present an optimised, parallel implementation of WHATSHAP, which was introduced by some of the authors in [20]. WHATSHAP focuses on solving wMEC, a weighted version of MEC. Our choice of WHATSHAP is due to the fact that, remarkably, it is the first exact fixed-parameter tractable algorithm for solving wMEC which, instead of being exponential in read length, is instead exponential only in the sequencing coverage, i.e. the maximum number of different reads that cover a single SNP position. This makes WHATSHAP particularly appealing with respect to the other currently available proposals, in the light of developments of future generation sequencing techniques, which will provide longer reads.

In wMEC, each SNP value comes with an associated confidence degree, which can be set to a combination of the confidence of the base call for that specific position, i.e. which allele the read comes from, and of the confidence of the mapping of the whole read within the chromosome. The confidence degree associated to each SNP is used as the cost of flipping/ignoring that SNP value in order to remove errors. In this way, by minimising the total weight of corrected SNPs, the optimisation problem corrects the most probable sequencing and mapping errors and can be viewed as a maximum likelihood approach. This improves the accuracy of WHATSHAP in comparison to methods that solve the unweighted
MEC problem. The weighted variant of MEC was first suggested by [12], and in [21] the authors proposed a heuristic for a special case of wMEC where they also present experiments show that wMEC is more accurate than MEC.

WhatsHap is still a computationally demanding algorithm. Experimental results show that single-chromosome datasets with a coverage up to 20 can be treated in about 2 hours on a single core of an Intel Xeon E5-2620 CPU. The analysis of a whole genome may require the solution of several independent instances of the haplotype assembly problem. In this context, the possibility a high-performance parallel WhatsHap appeared worth exploring.

The main contribution of this paper is to introduce pWhatsHap, an optimised parallel version of WhatsHap. We will focus on the parallelisation of the single chromosome haplotype assembly instance on a multi-core machine. Whole-genome approaches can be built on top of “embarrassingly parallel” instances of pWhatsHap on a range of different architectures.

pWhatsHap has been engineered by relying upon skeletons and parallel design patterns provided by the FastFlow framework [2], a methodological approach that allows WhatsHap to be parallelised with minimal changes to the original sequential code, while minimising the usage of typically slow classical mutual exclusion mechanisms.

Obtained results show a clear performance increase, allowing us to handle larger data sets, with bigger coverage, such as the ones that will be provided by future generation sequencing technologies.

Haplotype assembly and WhatsHap will be recapped in the next two sections, then pWhatsHap and obtained performance results will be presented.

2 wMEC model for Haplotype Assembly

The input dataset for this problem is a set of reads mapped to a reference genome. Arbitrarily re-labelling the alleles to 0 and 1 for each SNP position, the input data is represented as a matrix, having a row for each read and a column for each SNP position. Each element of the matrix reports the value of a given SNP in a read.

More formally, the input dataset is represented as an \( n \times m \) matrix \( F \), with \( n \) the number of reads and \( m \) the number of SNP sites. The elements \( f_{i,j} \) of \( F \) take values from the set \( \{0, 1, \neg\} \), telling whether, at position \( j \), the read \( i \) has the SNP value of the allele 0, or of the allele 1. A value of “\( \neg\)" indicates that the respective read does not cover the SNP position. In this case, we say that the read is not active at that position. In addition, a confidence value (or weight) \( v_{i,j} \) is associated to each active \( f_{i,j} \) as part of the input to the problem. The weight \( v_{i,j} \) is the confidence degree of the correctness of the value of \( f_{i,j} \), and in the optimisation problem wMEC, it represents the cost of flipping \( f_{i,j} \).

A conflict between two reads \( r_p \) and \( r_q \) is a SNP position where the two reads are active and have different values. In the absence of errors, a conflict between two reads implies that the two reads come from different alleles. In this framework, a correct haplotype assembly consists of a bipartition of the rows of
into two conflict free sets $R$ and $S$. Each conflict free set contains the complete set of reads assigned to the same haplotype. Unfortunately, such a bipartition into conflict free sets usually does not exist in real data sets due to sequencing and mapping errors. The problem thus becomes that of detecting a minimum-weight set of error corrections that allow for a conflict free bipartition. For instance, without correcting errors, no conflict-free bipartition exists for (the rows of) the following $3 \times 2$ matrix $F$ of coverage 3 (see column 2), where subscripts are the costs $v_{i,j}$:

$$F = \begin{pmatrix}
1 & 1 & 0 & 3 \\
9 & -8 \\
1 & 8 & -8
\end{pmatrix}$$

However, the minimum cost, conflict-free bipartition $R = \{1, 2\}$, $S = \{3\}$ can be obtained by correcting $f_{2,1}$, i.e. flipping it to 1 at a cost of 3.

Several heuristic approaches to solve MEC have been put forward in the last ten years, such as the greedy approaches of [19, 15] to assemble the haplotype of a genome, a method to sample a set of likely haplotypes under the MEC model [7], and the much faster follow-up to [7], based on the definition of a graph, analogously to [9], and an iterative greedy heuristic to optimise the MAX-CUT of that graph [6]. The latter outperforms [19, 15] while showing similar accuracy to [7]. In [18], reducing MEC to MAX-SAT and using a (heuristic) MAX-SAT solver has been proposed.

All of the abovementioned tools are heuristics – they provide no guarantee on the quality of the solution. To solve the MEC problem to optimality, several exact algorithms have been proposed. To this end, integer linear programming techniques have been developed [11, 8]. Fixed-parameter tractable (FPT) algorithms are another way of approaching the MEC problem and have been employed in [13]. However, as noted, these approaches have an exponential complexity in the number of SNPs per read or in the read length, which is going to increase soon and fast with emerging sequencing technologies. Most recently, this problem has been overcome by *WhatsHap* [20] which is a FPT algorithm with coverage as the cost parameter. It is thus better suited to the current development trends of sequencing technologies. Shortly after *WhatsHap*, an equivalent algorithm formulated in terms of belief propagation was independently proposed by Kuleshov [14]. The next section recaps *WhatsHap*.

### 3 *WhatsHap*

*WhatsHap* is a sequential algorithm, based on dynamic programming, that takes as input the fragment matrix $F$ (one row per read, one column per SNP position, and values in $\{0, 1, -\}$) and a set of confidence values associated to the active positions of the reads, as described in the previous section. It computes, with a dynamic programming method, a minimum-cost conflict-free bipartition of the set of reads.
WhatsHap builds a cost matrix $C$ with as many columns as $F$ (i.e., one column for each SNP). $C$ is constructed incrementally, one column at a time. Let $F_j$ be the set of all reads that are active in the $j$-th column, let $(R, S)$ be one of the possible bipartitions of $F_j$, and let $C(j, (R, S))$ be the entry for $(R, S)$ in the $j$-th column of $C$. Then,WhatsHap computes the minimum cost $C(j, (R, S))$ of making $(R, S)$ conflict free, for all possible $(R, S)$.

In general, a read spanning several consecutive positions will induce dependencies across columns, because a single read must be consistently assigned to the same allele throughout all the positions at which it is active (see read 2 in the example of section 2). Therefore, when computing the cost of the bipartitions of the same allele throughout all the positions at which it is active (see read 2 in the example of section 2). Therefore, when computing the cost of the bipartitions of $F_j$ for the construction of the $j$-th column of $C$, WhatsHap also needs to consider the (minimum) cost inherited by the construction of compatible partitions in $F_{j-1}$.

Entries $C(1, (R, S))$ in the first column of $C$, with $(R, S)$ a bipartition of $F_1$, only depend on the cost of making $R$ and $S$ conflict free (clearly, no inheritance from previous columns). $R \subseteq F_1$ can be made conflict free by flipping all 0s in $f_{1,k}$, with $r_k \in R$, 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 $W(1)_R^0$. Alternatively, $R$ can be made conflict free by flipping all 1s into 0s, paying $W(1)_R^1$. That is, taking the most advantageous alternative,

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

When considering the $j$-th column, both the contribution of the column itself (computed in the same way as for the first column), and the cost of a compatible bipartition inherited 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 minimum 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 preceding column $j - 1$ summarises all the corrections made in columns 1 to $j - 1$ for keeping $(R', S')$ conflict free. Summing up,

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

The schema of the generic $j$-th step of the algorithm consists of defining all the possible $(R, S)$ at $j$ and then performing the following three steps:

(a) determine the minimum local cost for making the $j$-th column conflict free by flipping some bits on the column according to their weights and the various correction possibilities;
(b) select the minimum-cost partition amongst those computed at step/column
\( j - 1 \) which are compatible with the current partition;

(c) fill in entry \( C(j, (R, S)) \) with the sum of the outcomes of (a) and (b).

Once the whole matrix \( C \) is computed, the result of the wMEC problem
is identified by the conflict-free partition \((R^*, S^*)\) of minimum cost in the last
column. The actual solution will also comprise all the minimum-cost corrections
made throughout the construction of the matrix, which have assigned each read
in \( F \) to partitions compatible with \((R^*, S^*)\).

The complexity of \textsc{WhatsHap} algorithm is dominated by the maximum
number of bipartitions that must be taken into account at a column. The num-
ber of possible bipartitions of the reads at column \( j \) is \( 2^{|F_j|} \), and therefore the
complexity is exponential in the amount of active reads that can be found at
a position. This critical (i.e. exponential) parameter is therefore the sequencing
coverage (see [20] for details).

We conclude this section with a few implementation details of \textsc{WhatsHap}
that are relevant for its parallelisation.

In the construction of the \( j \)-th column of \( C \), the possible bipartitions of \( F_j \)
are considered according to a Gray code enumeration, i.e. their binary encodings
are ordered in a way that the next entry differs from the previous one by only
one bit, e.g. 0001 and 0011, where 0 and 1 indicate the assignment of an active
read to either \( R \) or \( S \). This implies that two subsequent partitions differ in the
position of a single read \( r \) that moves from set \( R \) to set \( S \) (or vice versa). This
allows for an efficient incremental computation, since, accounting only for the
impact of moving \( r \), the computation of the cost of the next partition can be
obtained in constant time from the cost of the previous one because updating
the values of \( W(j)_{1_R}, W(j)_{0_R}, W(j)_{1_S}, W(j)_{0_S} \) requires constant time.

4 \textsc{pWhatsHap: high-performance haplotype assembly}

The development of a parallel solution for a given problem can be addressed
either by developing a parallel algorithm from scratch, or by parallelising an
existing sequential algorithm. Our work follows the second approach, where the
time complexity of \textsc{WhatsHap} is a strong motivation for choosing this path. In-
deed, \textsc{WhatsHap} is the first algorithm solving wMEC with a complexity which is
exponential only in the sequencing coverage. As explained, solving the weighted
version of the problem caters to its accuracy and exhibiting a complexity in-
dependent of the length of the fragments makes it particularly suitable for the
current trends in sequencing technology, which will provide fragments of increas-
ing length.

We will here focus on \textsc{pWhatsHap}, a parallel version of \textsc{WhatsHap} for a
single chromosome. The Multiple instances of haplotype assembly needed for a
whole genome are fully independent. Such independent runs can be executed concurrently in an embarrassingly parallel fashion, exhibiting best scalability when executed on truly independent platforms (e.g. clusters or cloud resources) where there is no performance degradation due to the concurrent usage of resources,
which instead may happen on multi-core architectures. For instance, multiple
instances of pWhatSHAP could be supported by cloud infrastructures, rightly
considered enabling technologies for bioinformatics and computational biology
that provide a large amount of computing power and storage in an elastic and
on-demand fashion.

Our pWhatSHAP targets multi-core architectures and relies upon the Fast-
Flow parallel programming framework [2].

4.1 Technological background

After decades of increasing clock-frequency and instruction-level parallelism in
single core architectures, the current trends for providing high-end performances
have steadily focused on increasing the number of cores per chip. Since current
multi-core architectures are de-facto small-scale on-chip parallel machines, the
most effective way to increase their performance is to use thread-level parallelism.
However, legacy sequential code does not necessarily benefit from multi-core ar-
chitectures, where single-core complexity and clock are typically lower than tra-
ditional single-core, and sequential code may perform even worse. Furthermore,
parallel programs are inherently more difficult to write than sequential ones due
to concurrency issues. Developers, including bioinformatics scientists, are then
facing the challenge of achieving a trade-off between high-end performances and
time to solution in developing applications and algorithms on current and forth-
coming multi-core platforms.

Parallel software engineering addressed this challenge via high-level language
extensions and coding patterns aimed at simplifying the porting of sequential
codes to parallel architecture, while guaranteeing the efficient exploitation of
concurrency [5]. Parallel design patterns (PDP) [16] have been recognised to
have the potential to induce a radical change in the parallel programming sce-
nario, allowing parallel programs to fully exploit the high parallelism provided by
hardware vendors, simplifying programmer’s tasks, and making whole applica-
tion development more efficient. PDPs provide tested and efficient parallel pat-
terns as composable building blocks, which eliminate the need of implementing,
tuning and maintaining ad-hoc solutions. The machinery available to application
developers is then at a higher level of abstraction with respect to traditional ap-
proaches, such as the Message Passing Interface (MPI), where the programmer
is fully responsible for the parallel behaviour of an application.

The FastFlow parallel framework provides algorithm skeletons and parallel
design patterns, enabling a good trade-off between performance, sequential code
reuse and time to solution.

Our multi-core parallelisation of WhatSHAP is based on FastFlow and ex-
plots the physical shared memory of the underlying architecture, making it unnec-
necessary to move data between threads, a typical source of overhead. However,
if this greatly simplifies the parallelisation, it also introduces data sharing and
concurrent access related problems. Parallel patterns provided by the FastFlow
framework solve these problems by defining clear dependencies among different
parts of the computations, hence avoiding costly synchronisations.
FastFlow has been demonstrated to be effective in parallelising and redesigning several sequential and concurrent applications, e.g. [1, 3, 17], and offers an important methodological approach for the parallelisation of WHATSHAP with minimal changes to the original sequential code.

4.2 Parallel WHATSHAP

WHATSHAP follows a dynamic programming approach based on recording so-far computed results in the incremental construction of the solution, i.e. the matrix of all the possible conflict-free partitions of minimum cost. In seeking for a possible decomposition of the algorithm into sub-problems to be solved in parallel by different executors, two obvious alternatives are possible: a vertical decomposition, where each executor builds a number of columns, i.e. they solve different parts of the genome, and a horizontal decomposition, where each executor builds some of the entries of the current column (combinations of the two alternatives could also be considered). The former would constitute a substantial departure from the original structure of WHATSHAP, whose incremental approach induces linear dependencies on columns: each one depends on the results of the previous one, i.e., the minimum-cost compatible partitions of the previous step (see p. 5). Such dependencies make a vertical decomposition difficult, left for future work.

This paper focusses on a horizontal decomposition: each executor evaluates a subset of the possible bipartitions \((R, S)\) of the set \(F_j\) of reads that are active in column \(j\).

The first step in the design of pWHATSHAP has been to profile the performance of WHATSHAP by measuring the time cost of generating the \(j\)-th column in the minimum cost matrix \(C\) (see p. 5). The time required in the construction of a typical column of a given dimension, i.e. the number of possible bipartitions of the column, depends on the coverage, i.e. the number of active fragments of that column (there are \(\sim 2^c\) bipartitions for a coverage \(c\)). The cost of building columns with a coverage less than 15 is minimal (< 1ms), and does not justify the overhead of a parallel construction. The situation is different for \(c > 15\), where the cost of building columns varies from a few milliseconds to a few seconds. In these cases it may be worth adopting an adaptive partitioning, varying the number of executors according to the dimension of the column.

Fig. 1 illustrates the first steps of the parallel construction of the columns of a minimum cost matrix \(C\) (Fig. 1(b)) for a fragment matrix \(F\) (Fig. 1(a)), with, e.g., read \(f_1\) being 0 in SNP 1 with confidence 5 and read \(f_2\) covering SNPs 1 and 2. In Fig. 1(b), \(C(1, (R, S))\) (left matrix) is built by considering all the possible bipartitions \((R, S)\) of the reads active on SNP 1, i.e. \(f_1, f_2\) and \(f_3\), which are represented as binary strings and Gray-code ordered (first three columns). The set of all possible bipartitions is split between two executors (thick horizontal line). Each executor builds \(C\) starting from the respective entry points (marked by \(A\)) and, in order to maintain as much as possible the original structure of the sequential algorithm, processes bipartitions in Gray code order (see p. 6). A bit of care was necessary to properly identify the entry points \(A\) in the Gray code sequence.
Fig. 1. (a): First two columns of a fragment matrix \( F \) with associated weights. (b): The (parallel) construction of the cost matrix \( C(j,(R,S)) \)

The costs in \( C(1, (R,S)) \) (column \( c1 \)) only depend on making the current partitions conflict free, e.g. for \( \{f_1, f_2, f_3\}, \emptyset \) (first row) by flipping \( f_1 \) to 1 (last three columns) at a cost of 5 (column \( c1 \)), so that \( R \) is conflict free and \( S \) empty. In general, for the construction of \( C_j \), the \( j \)-th column of \( C \) with coverage \( c_j \) and \( k \) executors, each executor processes approximately \( 2^o/k \) possible bipartitions \((R,S)\) of \( F_j \), with \( k \) dynamically depending on \( c_j \) (and on the hardware configuration). In this phase, each executor computes its own bipartitions in parallel with all other \( k-1 \) executors (map phase, see Fig. 2).

The construction of \( C_j \) depends on the minimum costs of the bipartitions in \( C_{j-1} \) which are “compatible” with those in \( C_j \), i.e. all those partitions in \( C_{j-1} \) which “agree” on the values of common reads, i.e. \( f_2 \) and \( f_3 \) in Fig. 1(b). This information is recorded on a suitable table (central matrix), where each executor over-writes the currently discovered best cost. This may induce write conflicts (\( Ws \) in the figure), which have been addressed by constructing local copies of the table for each executor, and then managing their merging by means of a sequential reduction phase, executed in pipeline with the map phase (Fig. 2). The information recorded in the table is then used to determine the costs in the next column (right matrix) as the sum (\( \Sigma \)) of the minimum cost of compatible bipartitions (\( m(c1) \)) and minimal corrections on the current bipartition (\( c2 \)). Concurrent read accesses (\( Rs \)) are of no particular concern.

Interestingly, if more than one minimum exists, the interplay of relative execution speed among parallel executors, may cause non-determinism in the last over-written minimum, thus providing different solutions of equal minimum cost over different executions. Comparison of different optimal solutions is left as future work.

4.3 Implementation details

The parallel construction of each column of the minimum cost matrix \( C \) has been implemented by using two FastFlow patterns pipeline and task-farm-with-feedback (Fig.2). The pipeline pattern consists of a 2-stage pipeline whose first stage is a task-farm pattern, with workers (\( Ws \)) connected both to the scheduler thread (\( S \)) and the second stage of the pipeline (\( R \)). The first stage implements

\[
\begin{array}{cccccc}
1 & 2 & \text{c1} & (R,S) & m(c1) & \Sigma \\
& & f_1 & f_2 & f_3 & f_1 & f_2 & f_3 & \min ((R,S),1) & \\
& & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0 \cdots 0 \cdots 0 \cdots 0 \\
& & f_2 & 1 & 3 & 0 & 2 & 0 & \min ((R,S),1) & 0.0 \cdots 0 \cdots 0 \cdots 0 \\
& & f_3 & 1 & 1 & - & 1 & - & \min ((R,S),1) & 0.0 \cdots 0 \cdots 0 \cdots 0 \\
& & f_4 & - & 0 & - & 0 & - & \min ((R,S),1) & 0.0 \cdots 0 \cdots 0 \cdots 0 \\
\end{array}
\]
Fig. 2. The FastFlow skeleton used in pWhatSHP. Each entity is a concurrent thread. The Emitter (S) produces and schedules tasks towards a pool of Workers (Ws). Each Worker sends results to the Reducer (R) and asks for new tasks from S.

the map phase of the proposed parallelisation, where a given number (chunksize) of the bipartitions of the fragment set F are computed by each worker in parallel.

The second stage of the pipeline consists of a simple sequential node called Reducer (R), which receives tasks from all workers (i.e. locally produced results) and then updates the matrix C with the minimum cost found (reduction phase on all inputs received). By using these patterns it is possible to exploit: i) Emitter–Workers pipeline parallelism: the Emitter computes all possible bipartitions sending disjoint sub-partitions to Workers using a dynamic scheduling policy; ii) parallelism among Workers: computation of local minimum costs in parallel; and iii) Workers–Reducer pipeline parallelisms: the Reducer receives multiple results in chunks from each worker.

The proposed parallelisation is quite direct and, importantly, requires minimal changes to the original sequential WhatSHP code. Furthermore, a high degree of parallelisation is involved due to the many entries of the large fragment table F corresponding to many (small) tasks that can be executed in parallel on the available cores.

5 Experimental results

In this section we report the results of experiments showing how effective the proposed parallelisation is. All the experiments were run using a workstation hosting two E5-2695 Ivy Bridge Intel Xeon processors, each with 12 cores, 64 Gbytes of main memory, Linux Red Hat 4.4.7 with kernel 2.6.32. Each core has two hardware contexts and thus a total of 48 threads are directly supported in hardware. The compiler used was gcc 4.8.2 with optimisation level –O3. For each experiment, CPU frequency was set to the maximum value possible for the considered platform (2.40 GHz, no turbo boost). The parallel version was executed using the
Table 1. Overall speedup and columns considered for each data sets.

<table>
<thead>
<tr>
<th>max. cov.</th>
<th>n. col.</th>
<th>Time (S)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>23,000</td>
<td>534</td>
<td>171</td>
</tr>
<tr>
<td>22</td>
<td>9,000</td>
<td>568</td>
<td>160</td>
</tr>
<tr>
<td>24</td>
<td>2,600</td>
<td>581</td>
<td>188</td>
</tr>
<tr>
<td>26</td>
<td>600</td>
<td>523</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 2. Speedup of the single column and % of columns with coverage ≥ 18.

<table>
<thead>
<tr>
<th>max. cov.</th>
<th>col. with &gt; 18</th>
<th>Time (S)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>77%</td>
<td>439</td>
<td>88</td>
</tr>
<tr>
<td>22</td>
<td>86%</td>
<td>485</td>
<td>87</td>
</tr>
<tr>
<td>24</td>
<td>83%</td>
<td>503</td>
<td>96</td>
</tr>
<tr>
<td>26</td>
<td>80%</td>
<td>422</td>
<td>70</td>
</tr>
</tbody>
</table>

shell command `numactl --interleave=all` to exploit all the available memory bandwidth of the 2 NUMA nodes of the hardware platform.

Since this paper aims at long reads, synthetic data sets with maximum coverage of 20, 22, 24 and 26 have been generated and used in the presented experiments. Such coverages correspond to quite large data sets. These data sets were produced by generating a single data set with an average coverage of 30 mapped to human genome, and then pruned to smaller coverage data sets (see [20] for details on the construction). Time performance was evaluated by measuring the time elapsed in the computation of subsets (i.e. a given number of columns) of each data set. The dimension of each subset was chosen to guarantee that the entire produced output could be stored in main memory.

First, we ran a set of tests aimed at determining the time spent when computing columns of different coverage. We found that, on the considered platform, it is worth parallelising only columns with a coverage ≥ 18, which we call higher coverage. Columns with coverage of 18 have an average computation time of about 7.4ms. Columns with a coverage of less than 18 (lower coverage) are processed in less than 1ms on average.

For higher coverage columns, we found that the best execution time was obtained by using all the cores of the platform (24), in particular by using 23 worker threads for the map phase and 1 thread for the reduction phase. Conversely, columns with lower coverage were computed using the same parallel pattern but with just 1 worker thread for the entire map phase (see Fig. 2). In this case the parallel skeleton is reduced to a pipeline of 2 sequential stages.

The experimental results obtained from running both the original sequential WHATS\textsc{Hap} and the new parallel pWHATS\textsc{Hap} are summarised in Table 1. For instance, for the data with maximum coverage of 20 (column max. cov.), we considered a subset of 23,000 contiguous columns (n. col.). The WHATS\textsc{Hap} execution time was 534s (Tseq) while the pWHATS\textsc{Hap} time was 171s (Tpar), thus obtaining an overall improvement of 3.12 (Speedup). For all coverages, the amount of main memory used was fixed to ~ 63 GB in all the tested cases. The overall obtained improvement ranges from 3 to 3.5 (see also Fig. 3. Left).

Since it has been observed that the fraction of sequential computation (including both the construction of columns with lower coverage and inherently sequential parts of the application) amounts to about 20% of the overall com-
putation time (see Table 2), from Amdahl’s law [4] it follows that the maximum possible speedup would be at most 5.0\,^7.

The results obtained considering only the columns with higher coverage (i.e. the ones we compute in parallel) are summarised in Table 2. In this case, the overall speedup ranges from 5 to 6 times.

The average execution time for computing a single column for several different coverages is reported in Fig. 3. Right (logarithmic scale). The per-column gain obtained, is always in the range 5 - 6.1, both for the smallest coverage (18), which has a very small computation time (∼7ms), and for the biggest coverage (26), which requires more than 2s of sequential execution for each column. The fact that we obtained almost the same speedup for very different computation granularities, clearly demonstrates that the limited scalability is not due to the overhead introduced by the parallel run-time code. Instead, we found that the limiting factor is mainly the extensive and non-regular memory access pattern exhibited by pWhaSHap, which does not allow the memory hierarchy of the chosen platform to be fully exploited in concurrent executions. This seems to be connected to the fact that the computation for higher-coverage columns is memory bound. However, further investigation is needed in order to clearly understand how, and if, it is possible to suitably re-organise pWhaSHap’s data structures to overcome this issue.

6 Final considerations

pWhaSHap aims at further stretching the capabilities of the computational analysis of DNA sequences, targeting high-coverage data sets of long fragments. The performed experiments clearly demonstrated the validity of the proposed

\footnote{Let $f$ be the fraction of the algorithm that is strictly sequential, 1/5 here, then the theoretical maximum speedup that can be obtained with $n$ threads is $S(n) = \frac{1}{f + \frac{1}{n}(1-f)}$, i.e. 5 with $n \to \infty$.}
parallelisation of WHATSHP, with an overall speedup of more than $3\times$ for such a fine-grained parallelism problem. Thanks to the design pattern methodology adopted for the parallelisation, such results have been obtained with minimal modifications to the original sequential code. Critical parts of the sequential algorithm amenable to further optimisation have been identified in the process, paving the way for future enhancements of the parallel algorithm. An extensive experimentation on human data sets is planned for future work.

References