The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-05939-8

PARamrfinder: detecting allele-specific DNA methylation on multicore clusters

Alejandro Fernández-Fraga[1] · **Jorge González-Domínguez**[1] · **María J. Martín**[1]

Accepted: 27 January 2024

Abstract
The discovery of Allele-Specific Methylation (ASM) is an important research field in biology as it regulates genomic imprinting, which has been identified as the cause of some genetic diseases. Nevertheless, the high computational cost of the bioinformatic tools developed for this purpose prevents their application to large-scale datasets. Hence, much faster tools are required to further progress in this research field. In this work we present *PARamrfinder*, a parallel tool that applies a statistical model to identify ASM in data from high-throughput short-read bisulfite sequencing. It is based on the state-of-the-art sequential tool *amrfinder*, which is able to detect ASM at regional level from Bisulfite Sequencing (BS-Seq) experiments in the absence of Single Nucleotide Polymorphism information. *PARamrfinder* provides the same Allelically Methylated Regions as *amrfinder* but at significantly reduced runtime thanks to exploiting the compute capabilities of common multicore CPU clusters and MPI RMA operations to attain an efficient dynamic workload balance. As an example, our tool is up to 567 times faster for real data experiments on a cluster with 8 nodes, each one containing two 16-core processors. The source code of PARamrfinder, as well as a reference manual, is available at https://github.com/UDC-GAC/PARamrfinder.

Keywords Allele-specific methylation · High performance computing · MPI · OpenMP · RMA · Dynamic load balancing

✉ Alejandro Fernández-Fraga
a.fernandez3@udc.es

Jorge González-Domínguez
jgonzalezd@udc.com

María J. Martín
mariam@udc.com

[1] Computer Architecture Group, CITIC, Universidade da Coruña, A Coruña, Spain

Published online: 21 March 2024

1 Introduction

Methylation is an epigenetic procedure that modifies the DNA by adding a methyl group (an alkyl derived from methane) to a DNA nucleotide. Methylation analysis is key for biologists as it is associated with different biological functions, and abnormal methylation levels can indicate the presence of certain diseases. One of these biological functions is genomic imprinting, a process by which only one copy of a gene in an individual is expressed; while, the other copy is suppressed. Most genes require a biparental contribution for their correct development; thus, the occurrence of genetic imprinting has been identified as the cause of some genetic diseases, such as the Prader–Willi [1], Angelman [2] or Beckwith–Wiedemann [3] syndromes.

Imprinted gene expression is regulated by Alele-Specific Methylation (ASM), a particular type of methylation that occurs when DNA methylation patterns are asymmetrical between alleles. Because of this, the identification of Allelically Methylated Regions (AMRs) has gained attention in the last years, as it provides interesting biological insights.

amrfinder [4] is a cutting-edge tool to discover ASM based on data from Bisulfite Sequencing (BS-Seq) experiments. It has shown high sensitivity, specificity and control of type I errors, while it is, up to our knowledge, the only freely available software that is capable of detecting ASM at regional level without Single Nucleotide Polymorphism (SNP) information for an individual sample. In addition, *amrfinder* is part of *MethPipe* [5], a pipeline of tools for methylation analysis that is highly referenced in the literature. This made it the choice to perform interesting biological studies in fields such as early human development [6], genetic–epigenetic interactions [7] or the building of allele-specific epigenome maps [8]. On top of that, *amrfinder* has been successfully utilized in several recent studies, demonstrating its continued relevance and effectiveness in the field of epigenetics [9–11].

Although these analyses that identify AMRs can obtain interesting biological insights for understanding the role of DNA methylation in genomic imprinting, they come with a steep computational cost. This is the main drawback of *amrfinder*, as it requires a high runtime to process large input datasets. In this paper we present *PARamrfinder*, a tool that is able to accelerate the identification of AMRs on modern multicore clusters. The high performance of the tool has been achieved by implementing a series of optimization, including:

- A profiling of the sequential tool which lead to a sequential optimization of its main bottleneck to half its cost.
- A identification of the load balancing issue of the application by spotting the different causes.
- The implementation of different workload distribution algorithms.
- A comparison of the efficiency of these algorithms in dealing with the load balancing problem.
- The development of parallel I/O algorithms.
- The elimination of the two new performance bottlenecks that arose by applying several parallelization techniques.

Thanks to these contributions, the novel tool obtains the same highly accurate biological results as *amrfinder*, but in a significantly shorter time. It uses an efficient hybrid approach that combines Message Passing Interface (MPI) [12] processes and OpenMP [13] threads. The rationale behind this hybrid approach is that these analyses are mostly performed on HPC multicore clusters, a kind of system where MPI and OpenMP usually obtain the best performance [14]. Each MPI process launches multiple threads to efficiently exploit the cores available on each node and take advantage of the Hyperthreading technology supported by many CPU architectures. In addition, MPI Remote Memory Access (RMA) operations are used to build a dynamically balanced workload at the process level.

The rest of the paper is organized as follows. Section 2 presents the state of the art. Section 3 introduces some background concepts about the original *amrfinder* tool that are necessary to understand the goal of this work and the implementation of our method. Section 4 describes the parallel implementation of *PARamrfinder*. Section 5 provides the experimental evaluation in terms of runtime and scalability. Finally, concluding remarks are presented in Sect. 6.

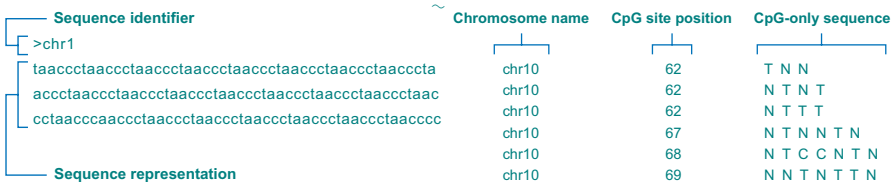
2 Related work

There has been a great effort for many years in the development of tools to detect ASM regions for datasets obtained from different types of biological technologies, such as microarrays that genotype bisulfite-converted DNA [15], lower resolution capture technologies such as Methyl-Binding Domain (MBD) sequencing [16] or Methylated DNA ImmunoPrecipitation (MeDIP) sequencing [17]. However, currently the most popular and widely used technology is high-throughput short-read BS-Seq [18], as it has the ability to detect ASM at the single nucleotide level.

Even though, most of the tools that detect ASM based on BS-Seq do so by associating this data with heterozygous SNPs. Some examples are Bis-SNP [19], Allelonome.PRO [20] or CGmapTools [21]. These approaches that depend on genotypic data present an important limitation: They are blind to some portions of ASM, since imprinted methylation is not necessarily associated with genotypic variation.

Only a few tools overcome this limitation. *amrfinder* [4] is one of them, as its model is genotype-independent, and so it is widely applicable to the identification of ASM in the context of imprinting. *allelicmeth* [4] is another tool from the same authors which does not rely on SNP data, and differs from *amrfinder* in that it provides an ASM score for each CpG site, instead of identifying AMR. Another tool that stands out is *DAMEfinder* [22], as it can run in two modes, SNP-based and tuple-based, hence does not necessarily depend on SNP data. However, the purpose of this tool is not to identify ASM over a sample, but to discover different patterns of ASM over samples of two conditions (treatments, diseases,...). Due to its purpose, this method depends on the availability of data from multiple samples of the two conditions.

Up to our knowledge, there is no previous work focused on accelerating the identification of ASM with High Performance Computing (HPC) techniques. Nevertheless, we can find in the literature other bioinformatics tools that are



(a) Example of *FASTA* file (b) Example of *epiread* file

Fig. 1 Example of *amrfinder* input file formats

able to efficiently exploit the computational capabilities of multicore clusters. Some examples are *pIQPNNI* [23], that uses MPI to infer maximum-likelihood phylogenetic trees from DNA or protein data with a large number of sequences; *MPIGeneNet* [24], that includes MPI routines and OpenMP directives to construct genetic networks; or *ClustalW-MPI* [25], which applies MPI to reduce the execution time for aligning multiple protein or nucleotide sequences. These previous tools have been used by biologists to complete experiments on large real-world datasets in reasonable time, proving that a tool as *PARamrfinder* can be attractive for the scientific community.

3 Background: amrfinder

amrfinder [4] is a publicly available software (as part of the *MethPipe* pipeline[1]) for identifying AMRs in mammals, in which methylation occurs mainly in CpG sites (fragments of DNA where a cytosine (C) nucleotide is followed by a guanine (G)). The tool identifies those regions from BS-Seq data, a method by which genomic DNA is treated with sodium bisulfite and then sequenced, providing single-base resolution of methylated cytosines in the genome. Upon sequencing, unmethylated cytosines are converted to thymidines (T); while, methylated cytosines remain as cytosines (C). The tool is able to achieve high accuracy and low false discovery rate (less than 0.01 in all the tested cases) by applying both a single-allele and a allele-specific model to the data and comparing the likelihood of these models to determine the presence of ASM. The single-allele model assumes that the set of reads mapped to an interval represent one single methylation pattern; while, the allele-specific model assumes that these reads represent two distinct methylation patterns, both of them constituted by roughly the same proportion of the reads (as the alleles themselves are present in equal proportions). The parameters of these models are iteratively adjusted to fit the data.

amrfinder is a command-line tool written in C++ that admits several configuration parameters (e.g., the maximum number of iterations used for fitting the models) and obtains the biological input data from text files. Concretely, the inputs are:

[1] http://smithlabresearch.org/software/methpipe/

- A reference genome, contained in a series of *FASTA* files, to which the results must be aligned. Figure 1a shows as example a fragment of a dataset that follows the *FASTA* format. This format is used to store text-based sequences of nucleotides or amino acids represented using single-letter codes. Each sequence begins with a greater-than character (">") followed by a description of the sequence, all in a single line. The next lines contain the sequence itself, which is represented by a series of characters, each representing a single nucleotide. In the case of *amrfinder*, these sequences are the chromosomes of the reference genome.
- An *epiread* file, which contains the input reads in a compressed format, indicating only the methylation state for each CpG site. Figure 1b shows an example of an *epiread* file, where each row is dedicated to one read and consists of three columns. The first column is the chromosome of the read, the second is the numbering order of the first CpG in the read, and the last is the CpG-only sequence of the read.

Algorithm 1 *amrfinder*'s workflow

Input: The path to a file, $epiread\_file$, containing the input reads
The path to a directory, $fasta\_files$, containing the reference genome
Output: A file, $amrs\_file$, containing the identified allele-specific methylated regions

```
1  amrs ← CreateEmptyList()
2  /* Identify AMRs on one chromosome at a time */
3  foreach Chromosome chrom in the epireads_file do
4  |  chrom_epireads ← ReadEpireadsFromChromosome(chrom, epireads_file)
5  |  amrs ← amrs + IdentifyAMRs(chrom_epireads)
   end
6  /* Post-processing to ensure the accuracy of the results */
7  amrs ← PostProcessAMRs(amrs)
8  MapAMRsToReferenceGenome(amrs, fasta_files)
9  /* Write the final list of AMRs to a file, one AMR per line */
10 WriteAMRsToOutputFile(amrs, amrs_file)
```

Algorithm 2 Pseudocode of the amrfinder's processing phase

```
1  amrs ← CreateEmptyList()
2  foreach Chromosome chrom in the epireads_file do
3  |  chrom_epireads ← ReadEpireadsFromChromosome(chrom, epireads_file)
4  |  foreach Window win in chrom do
5  |  |  /* Get only the reads containing CpGs within the current
   |  |     window limits */
6  |  |  window_epireads ← GetCurrentEpireads(win, chrom_epireads)
7  |  |  /* Only if the window contains a minimum of information
   |  |     it's worth to test */
8  |  |  if ContainsEnoughInformation (window_epireads) then
9  |  |  |  single_allele_model_result ←
   |  |  |   FitSingleAlleleModel(window_epireads)
10 |  |  |  two_allele_model_result ← FitTwoAlleleModel(window_epireads)
11 |  |  |  is_significant ← EvaluateWindow(single_allele_model_result,
   |  |  |   two_allele_model_result)
12 |  |  |  /* If the test proved the candidate window to be
   |  |  |     allele-specific methylated, append it to the output
   |  |  |     list */
13 |  |  |  if is_significant then
14 |  |  |  |  AddAMR(amrs, window_epireads)
   |  |  |  end
   |  |  end
   |  end
   end
```

Algorithm 1 illustrates the workflow that follows *amrfinder* to identify AMRs in the reads of the *epiread* file. The tool works with one chromosome at a time, so the first step consists in bringing the input reads of one chromosome to memory (Line 4). Once all the reads from the chromosome are ready, the main computation of the tool is executed, the identification phase, which produces the list of AMRs (Line 5). This process is repeated chromosome by chromosome until all the reads from the *epiread* file have been processed. After that, *amrfinder* runs several post-processing steps over the identified AMRs to ensure the accuracy of its results, merging regions close to each other and excluding intervals overlapping large subunit ribosomal RNA. Most of these post-processing steps are lightweight in terms of execution time. The only exception is the mapping of the ARMs to the reference genome (Line 8), which is expensive both in terms of CPU and I/O resources, as it requires *FASTA* files containing the reference genome (several GBs) to be read to memory so that their data can be traversed to find the right mapping of AMRs.

As previously mentioned, the main bottleneck of the tool is in the identification phase (Lines 3–5 in Algorithm 1). Algorithm 2 details how this phase works. Chromosomes are read into memory one by one (Line 2) and, for each one, all CpG site positions are traversed using a fixed-width (fixed number of CpG sites) sliding window (Line 4). Figure 2 shows the behavior of this window, which is moved

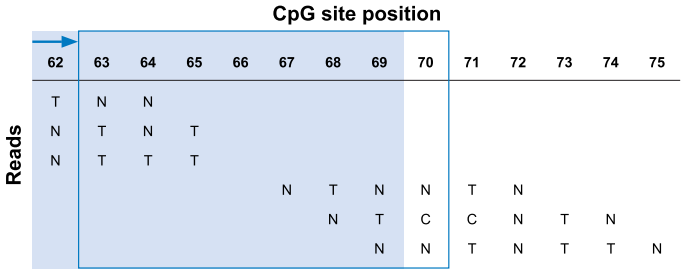


Fig. 2 Example of the sliding window behavior using a size of 8. Shaded, the window initially at position 62. Bordered, the window moving one position to analyze the next interval

one position each time. For each position, the reads bounded by the current window are selected (Line 6). If that window does not contain a minimum number of reads, those reads are discarded (Line 8). Otherwise, they are processed using the statistical models (Lines 9–11). In case ASM is ascertained, the region is added to the list of AMRs (Lines 13–14).

amrfinder provides two types of information per identified region: the bounds where it was observed and a false discovery rate provided by the probability models.

4 Implementation

PARamrfinder is a novel tool to accelerate the identification of AMRs that provides exactly the same output results as *amrfinder* (and thus its high accuracy) but at significantly lower runtime thanks to exploiting the computational capabilities of multicore clusters. The parallel tool has been implemented using MPI and OpenMP trying to achieve the lowest possible runtime on this sort of systems. In addition, *PARamrfinder* keeps the same configuration mechanism as *amrfinder* in order to simplify its adoption by those biologists who are already familiar with the original tool. More information about this configuration procedure can be found in the reference manual that is available in the public repository of *PARamrfinder*.

4.1 Sequential optimization of the statistical models

Before starting the parallel implementation, a code analysis was carried out to ensure the original tool was a right and effective baseline for the novel parallel tool. However, this analysis pointed out that the original implementation for the fitting of the statistical models was inefficient. In particular, some costly computations that were needed throughout this fitting stage (mostly the calculation of logarithms, a particularly expensive operation) were being discarded and computed back several times. To avoid that inefficient behavior, a new technique, that we have named *ComputeAndStore*, was implemented. It consists in performing all the computations the first time they are required, and storing the results in a

buffer. If the results of these computations are required again, a simple and fast access to the buffer is enough to fulfill these requests.

4.2 Load balancing issues

In addition to the inefficient fitting method, *amrfinder* comes with severe load balance issues that makes it inadequate for a naive parallel implementation and it becomes a challenge to achieve an efficient data and workload distribution. The reason is that different regions might need extremely different computation times for their analyses.

First of all, most of the regions do not have enough information to make the analysis meaningful. Thus, many of them will be discarded without even having to execute the computationally expensive models (Algorithm 2, Line 8). This divides the regions into two very distinct groups. On the one hand, the regions that will not be analyzed and which have a near-zero associated execution time and, on the other hand, the regions that will be analyzed and need a relatively high execution time.

Second, even within the regions that will be analyzed, there is also a large imbalance in the workload associated with each of them. This imbalance is due to two factors:

- The amount of information in the region. Regarding this factor, not all the regions have the same amount of reads associated with them. Also, each read does not need to contain information about every CpG site in the region. Therefore, the amount of information in the region is not a constant, but rather a variable that depends on the number of reads and the number of CpGs represented on them. There exists a direct relationship between the total number of reads corresponding to each of the CpGs associated to a region (amount of information) and the execution time of that region. In consequence, the execution time of a region can be predicted beforehand based on the amount of information it contains.
- The number of iterations it takes for the statistical models to converge. This factor also follows a direct relationship with the execution time of a region. However, it is unpredictable and remains unknown until the model fitting process is completed. Thus, it is not possible to estimate the execution time of a region based on the number of iterations since this information is not known in advance.

Moreover, it is common for a few small groups of regions to contain a huge amount of information (up to five orders of magnitude larger than the median). We have called them *elephant regions*, and their analyses represent a relevant percentage of the total runtime of the program. The management of the *elephant regions* is a key factor in the performance of the parallel implementation since they can easily generate a great unbalance in the workload that can become the bottleneck of the execution if they are not properly managed.

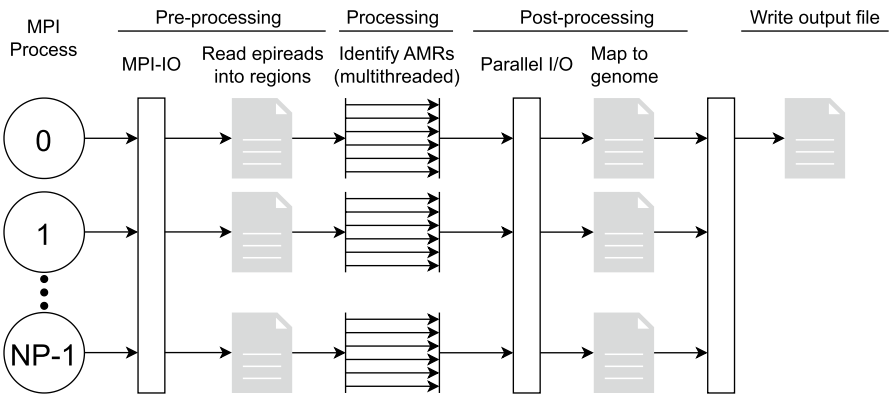


Fig. 3 Workflow of *PARamrfinder*

4.3 Parallel implementation of the identification phase

PARamrfinder includes two levels of parallelism in order to exploit the computational capabilities of current multicore clusters when accelerating the identification of AMRs. First, MPI routines are used to distribute data and workload among processes that are placed on different nodes. As seen in Sect. 3, *amrfinder* must perform the same operations on different regions. *PARamrfinder* distributes those regions among the MPI processes. Second, each MPI process spawns several OpenMP threads that run on different cores of the node. The regions assigned to each process are distributed among the OpenMP threads. As different regions can have different computational load, a dynamic scheduling policy is used to guarantee a good load balance among the threads. Figure 3 provides a graphical overview of the two-level parallel implementation applied in *PARamrfinder*. For simplicity, multithreading is only illustrated on the processing phase. However, the pre-processing and post-processing phases also use threads, as will be discussed in Sects. 4.4 and 4.5.

Even though a pure MPI program could take advantage of all the cluster hardware, this hybrid approach has several benefits:

- Improvement of the memory management, as threads belonging to the same process can access the same shared memory structures; while, MPI processes would need copies of the structures for each process, leading to memory overheads.
- Fewer synchronizations are required, as threads do not need to communicate through message passing.
- Possibility of exploiting Hyperthreading in modern CPUs, a technology that facilitates more efficient utilization of processor's resources by enabling multiple threads to run on one physical core. Two concurrent threads per CPU core are common, but some processors support up to eight concurrent threads per core.

Nevertheless, due to the issues discussed in Sect. 4.2, an efficient distribution of the workload at the process level is not trivial. The original *amrfinder* works

chromosome by chromosome, i.e., it gets all reads belonging to a chromosome, figures out the regions, analyzes them, and goes to the next chromosome. This means that only one chromosome is kept in memory at a time. Although this is the best choice for the original tool, it is not for *PARamrfinder*, as it needs to obtain information about all the regions beforehand to make a good distribution of the workload. Then, the parallel tool starts with a pre-processing step to know the total number of regions to analyze, as well as the amount of information contained in each of them. This pre-processing was designed in such a way that it does not introduce significant overhead in the execution. To accomplish this, a sort of *loop fission* is applied to the main loop of Algorithm 2 (Line 2), splitting it into two parts:

- A pre-processing loop, removing the computationally expensive instructions from the original loop. In this step the regions without enough information are filtered out. Additionally, for those regions that pass the filter, useful data such as their amount of information on their bounds is already precalculated and stored in memory.
- A computation loop, where the models are executed for the regions that contain enough information.

The pre-processing loop provides the parallel tool with the necessary information to try to get a fair workload balance. The next sections describe several balancing strategies based on this information.

4.3.1 Pure block distribution

As a first step, a naive approach was tested. Thanks to the pre-processing loop, the total number of regions to analyze is known. The tool uses this information to evenly distribute them among the MPI processes, each one receiving a contiguous block of regions to analyze. All regions, initially held in the memory of the root process, are statically distributed among processes before starting the computation loop using *MPI_Scatter*. After the computation loop, when all AMRs are correctly identified, processes synchronize and gather the results in the root process using *MPI_Gather*.

A contiguous block distribution is more appropriate than a cyclic one as contiguous regions have contiguous reads and, then, it is enough for each MPI process to store a single block of reads. This helps to improve data locality and prevents processes from having to access blocks of data that are separated from each other by an offset, which would lead to performance degradation.

Due to the reasons explained in Sect. 4.2, different regions will have huge differences in their computational cost. This generates situations where, although the number of regions is evenly distributed, the workload is not.

4.3.2 Cost-based block distribution

The previous approach could present, depending on the input dataset, severe workload imbalance at the process level, which can lead to huge performance

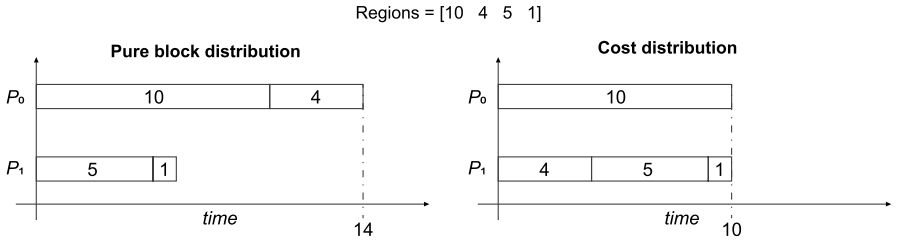


Fig. 4 Example comparing the performance of the pure block distribution and the cost-based one

degradation and inefficiencies. To improve the load balance, a static cost-based distribution was designed.

In this distribution the cost of each region is assigned according to its amount of information. That is, instead of receiving blocks with the same amount of regions, processes receive blocks of variable size but with the same total amount of information. This way, each MPI process is expected to have a similar total workload to distribute among its OpenMP threads.

For the implementation, the root process has to perform a linear iteration through the regions to figure out the boundaries of the blocks, as their sizes differ and are unknown in advance. As each process receives now a variable number of regions, they are distributed among processes using *MPI_Scatterv* and, after the computation loop, the results are gathered using *MPI_Gatherv*.

Figure 4 shows an example to compare the performance of the pure block distribution and the cost-based one, using four regions with variable execution time (represented by the vector *Regions*). The first one distributes two elements to each process, without focusing on the execution cost, which leads to *P1* remaining idle from $t = 6$ to $t = 14$. The cost-based distribution, though, takes into account this factor, giving only one element to *P0* and three to *P1*, which results in both processes ending the execution at $t = 10$.

This new approach has a great impact in reducing the imbalance due to the *elephant regions*, as it takes into account their high amount of information. However, it does not achieve fully balanced distributions for a couple of reasons. First, the cost estimation, although accurate, is not exact, so this source of imbalance is reduced, but not eliminated. Second and more important, the approach does not take into account the imbalance caused by the number of iterations needed to fit the models (see Sect. 4.2).

4.3.3 Dynamic distribution

A new dynamic distribution was designed to cope with the workload imbalance due to the fact that some regions need more iterations to converge.

The main idea behind this approach is to assign small blocks of regions on demand, minimizing synchronization among MPI processes and communication overheads. Passive RMA communications were used for this purpose. Three structures are allocated in the RMA shared memory of the root process:

- A portion of shared memory to contiguously store the blocks of regions.
- A shared array to store the results of the computations.
- A shared index initialized to zero that points to the next block to analyze.

After the initialization, all processes enter a loop where they get the current index and increment it to the next position using *MPI_Fetch_and_Opt*. If the index is within bounds, the process gets the block of regions from RMA memory using *MPI_Get*, computes its results, and stores them back at the right offset of the output RMA buffer using *MPI_Put*.

Nevertheless, a naive implementation of this algorithm does not lead to a performance improvement, as *elephant regions* still provoke workload imbalance. Firstly, the runtime of blocks containing these regions is much higher than the rest, and one single *elephant block* can fill the whole execution time of the tool in a single process. In addition, if any of these *elephant blocks* is computed at the end of the distribution, it creates a situation where only one process works; while, the rest are idle.

To deal with this problem, the dynamic distribution is performed in two steps. First, blocks containing *elephant regions* are distributed using a small block size to ensure that these regions do not overload a particular process during its whole runtime. Also, these *elephant regions* are first assigned to the processes, as they are the ones that take most time, and finding one at the end of the execution would lead to massive imbalances. Finally, the remaining regions are distributed among the processes using a larger block size.

The size of the blocks will have a great impact on the performance of the tool. Too small blocks lead to communications overheads. To the contrary, too large blocks increase the potential imbalance among them. Due to these reasons the block size is calculated at runtime to make the program flexible and able to deal with different datasets and configurations. It takes into account the amount of processes and the total cost of the dataset, guaranteeing a minimum amount of blocks per process and trying to split every dataset into roughly the same number of blocks, both to ensure that low-cost datasets are distributed with a sufficiently fine granularity, and also that the more expensive datasets do not waste time with needless communications and synchronizations that do not improve the balance. In addition, to ensure that each block takes a reasonable time to be processed, the threads-per-process ratio is also taken into account by enlarging the block size proportionally to the number of threads used for each block.

4.4 Parallel input

A benchmarking of a preliminary version of *PARamrfinder* pointed out that the input read and the pre-processing loop became a bottleneck when the identification phase was split among several processes and threads and significantly degraded the overall performance of the program. Therefore, these phases were redesigned with parallel computing in mind.

First, MPI-I/O functions were used to parallelize the input phase, allowing each process to read from a certain offset. The input format (see Fig. 1b) is very

appropriate for parallel processing in the pre-processing phase, since each process is only interested in a block of consecutive reads (rows) that must be mapped into a series of consecutive regions. Therefore, in *PARamrfinder* each process only reads a block of rows that it must map into regions to run through the pre-processing phase. However, this approach presents four main drawbacks:

- Not all rows have the same length.
- The number of rows in a file is not known in advance.
- The mapping of reads into regions is not a one-to-one, but a many-to-many mapping.
- The first and last regions in a process might be duplicated and incomplete as they might lack information from reads owned by adjacent processes.

Algorithm 3 *PARamrfinder*'s parallel input

Input: The path to the input file, *filename*, containing the input reads
Output: A list of regions to be analyzed, *regions*

```
1 file_chunk ← ReadInputFileToMemoryInParallel(filename)
2 reads ← ProcessFromRawToReads(file_chunk)
3 my_regions ← IdentifyRegionsFromReads(reads)
4 last_valid_region ← FilterLeftoverRegions (my_regions)
5 first_valid_region ← CommunicateLastValidRegion(last_valid_region)
6 RemoveLeadingRegions(my_regions, first_valid_region)
7 regions ← GatherRegionsOnRootProcess(my_regions)
```

Algorithm 3 shows the pseudocode of the parallel implementation for the input phase that solves all the pointed problems. First of all, *PARamrfinder* gets in advance the size of the input file (in bytes) and then distributes them among processes trying to generate a fair distribution of reads (Lines 1 and 2). To ensure that each process does not miss any information related to its regions, an overlapping technique is implemented: assuming p processes, process $n \in [0, p\text{-}1]$ reads its block and some extra final bytes to ensure that it is able to correctly process all the regions (Line 3), even those that may share information with process n+1. This solves the problem of incomplete regions, but not the problem of duplicate regions. To avoid this, process n removes its last regions until it finds one that may not be complete on process n+1 (Line 4). Then, process n shares this window with process n+1 (Line 5) and process n+1 removes its first regions until it finds one that is not complete on process n (Line 6). Finally, all the regions are gathered in a vector on the root process (Line 7) to be processed in the identification phase as explained in Sect. 4.3.

Figure 5 shows an example of the distribution of the reads in an *epiread* file between two processes using this algorithm. In the figure both processes P0 and P1 have assigned a block of four reads each. However, with this read distribution, none of them is able to correctly identify the selected window (the one shadowed in blue, which covers CpG positions 62–69), as both miss part of the reads associated with the window. The additional overlapping solves this issue for P0, as it now has all

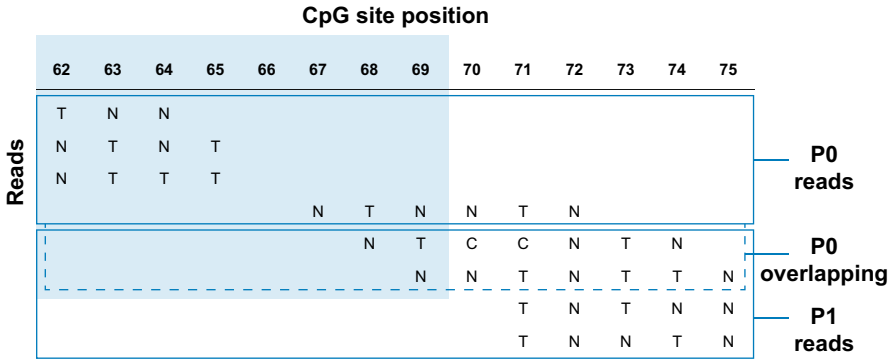


Fig. 5 Example distribution of an *epiread* file containing eight reads with parallel MPI I/O. A block with the first four reads is assigned to P0; while, another block with the four last ones is assigned to P1. Framed with a dashed line, the first two reads belonging to P1 are also assigned to P0 as overlapping between the processes. Shadowed in blue, a window covering the CpG positions 62 to 69, with whom six reads have associated CpGs

the reads it needs, but not for P1, which still misidentifies this region. P0 then must communicate to P1 that it has identified this window correctly so P1 should drop it to avoid duplicates.

4.5 Parallel post-processing

After the identification phase, the tool runs several post-processing steps over the identified AMRs (see Sect. 3). As already mentioned, the most time-consuming of these steps is the mapping of these AMRs to the reference genome. In fact, it became the main bottleneck after the parallelization of the input phase. Therefore, it was parallelized as well, as shown in Algorithm 4.

Algorithm 4 *PARamrfinder*'s parallel post-processing

Input: A list of AMRs, $amrs$, to map to the reference genome
A list of files, $fasta\_files$, containing the reference genome
Output: The same list, $amrs$, but also including their maps to the reference genome

```
1 my_chrom_mapping ← IdentifyChromsOnFastaFiles(fasta_files)
2 full_chrom_mapping ← ShareChromMapping(my_chrom_mapping)
3 my_amrs ← DistributeAMRsAmongProcesses(amrs)
4 MapAMRsToReferenceGenome(my_amrs, fasta_files, full_chrom_mapping)
5 amrs ← GatherAMRsOnRootProcess(my_amrs)
```

The objective is to distribute the AMRs among processes and threads so they can be mapped in parallel. However, the reading of the reference genome is as time-consuming as this mapping, so the input operations of the *FASTA* files also need to be performed in parallel to get rid of the bottleneck. Since each process works with a

subset of the AMRs, it only needs to read a subset of the chromosomes of the reference genome. Nevertheless, the location of each chromosome in the input file is not known in advance. Therefore, the reference genome must first be scanned to identify the location of each chromosome (Line 1). If all chromosomes are stored in a single *FASTA* file, each process will scan a block of it. If each chromosome is stored in a different *FASTA* file, these files are distributed and scanned in a round-robin fashion among the processes. After this initial scanning, processes share their information (first they share the number of chromosomes that they have identified, using *MPI_Allgather*, and then they share the actual information, using *MPI_Allgatherv*), so all of them can locate all the chromosomes (Line 2). These two steps grant processes direct access to any particular subset of chromosomes so they can read them to memory without any additional overhead. Once this objective is achieved, the AMRs are distributed among the processes (Line 3). Next, each process maps its AMRs to its subset of chromosomes using all its threads concurrently (Line 4). Finally, the AMRs are gathered back on the root process (Line 5) so that the next post-processing step can be performed.

4.6 Parallel implementation overview

After the implementation of these parallel techniques, the structure of *PARarmfinder* is significantly different than that of the original sequential tool. Figure 6 shows this new structure. *PARamrfinder* can be divided into four phases.

1. **Pre-processing** This phase is in charge of reading the input *epiread* file to memory using MPI-IO. After that, the raw text is parsed to the adequate data structure using all the processing elements (i.e., processes and threads). Finally, these reads are used to pre-process and filter the regions that will be computed in the next phases, also using all the available processing elements.
2. **AMR identification** During this phase the candidate regions are distributed among the processing elements, so that each one executes the models for each of the assigned regions to determine if there is ASM in them. When all the results are computed, the root process gathers them.
3. **Post-processing** This phase is in charge of executing several post-processing steps over the AMRs identified in the previous phase. Most of them have a low computational cost and are executed sequentially. The only exception is the mapping of the AMRs to the reference genome, which is executed in parallel. As a first step, MPI processes read the reference genome from the *FASTA* files to memory. Then, the AMRs are distributed among processes and threads and mapped to the reference genome in parallel.
4. **Output** After the post-processing, the root process sequentially writes the results to the output file.

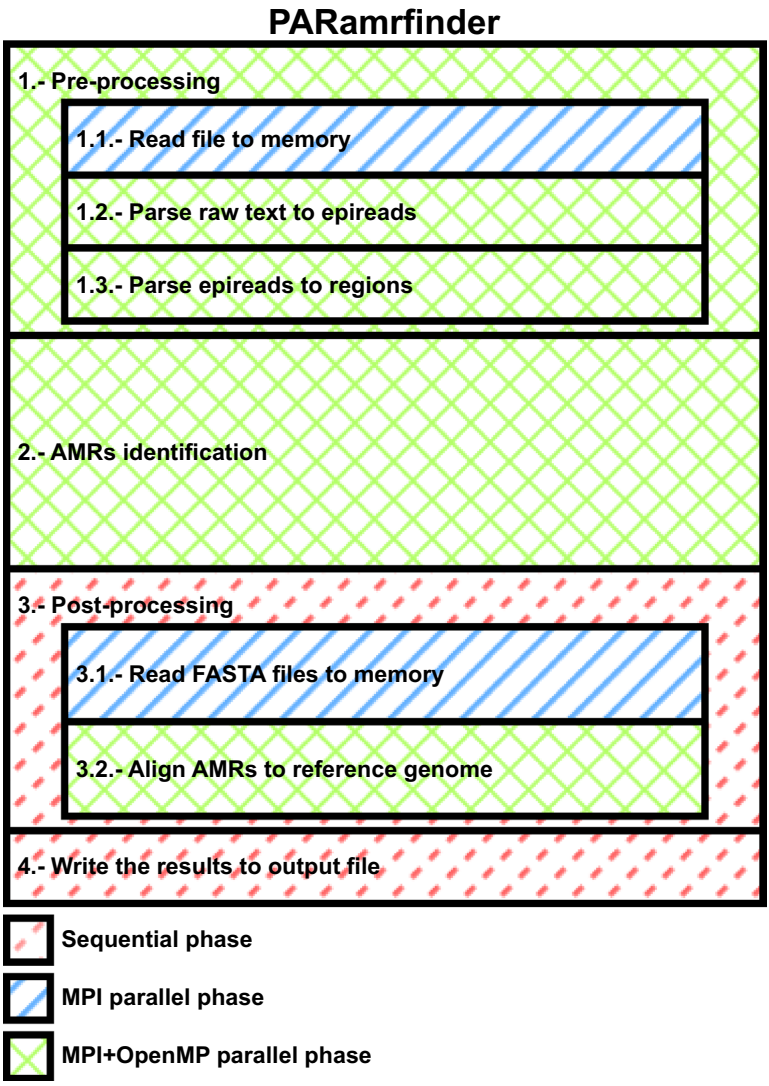


Fig. 6 Parallel structure of *PARamrfinder*

5 Experimental evaluation

The experimental evaluation of *PARamrfinder* has been performed in terms of execution time, scalability and memory consumption, as our tool provides the same AMRs as *amrfinder*, whose high accuracy has been proved in [4]. This equality has been proved by comparing the raw results for each execution of the parallel tool with a reference obtained from the execution of the original *amrfinder* using the

Table 1 Datasets specification

Dataset	Ref. genome	Size Ref. genome	Size epiread file	#Reads
ERS2586503	hg19	3.0 GB	984 MB	52,837,538
ERS4575883	mm10	2.7 GB	2.0 GB	117,430,460
ERS7819375	mm38	2.6 GB	1.6 GB	97,424,137
ERS208315	hg38	3.1 GB	9.1 GB	574,149,340

same input data and the same configuration. In addition a validation data set has been added to the tool repository[2] with the reference output, in order to facilitate the validation of the tool by the community. Consequently, this section provides a performance comparison of both tools on an 8-node cluster with a total of 256 CPU cores (32 cores per node). Each node has two sixteen-core Intel Xeon Silver 4216 Cascade Lake-SP processors with support for Hyperthreading (up to two logical threads per CPU core), and 256 GB of memory. The nodes are interconnected through a low-latency and high-bandwidth InfiniBand EDR network. Regarding software, both *amrfinder* and *PARamrfinder* were compiled with the GNU GCC compiler v.8.3.0, and the latter is linked to the OpenMPI library v.4.0.5.

Four different real biological datasets were used for this experimental evaluation, which are named according to the SRA id of their related sample. The *ERS2586503* dataset is used to investigate the epigenetic phenotype of sessile serrated adenomas/polyps [26]. The *ERS4575883* dataset provides information related to the detection of individual molecular interactions of transcription factors and nucleosomes with DNA in vivo [27]. The *ERS7819375* dataset brings treatment-resistant cells in breast cancer [28]. These three datasets have been obtained from the NCBI public repository of SRA data,[3] while the *ERS208315* dataset has been generated by the Blueprint Consortium from venous blood data[4]

The datasets are provided as raw *FASTQ* data [29]; thus, they must be converted to the *epiread* files required by *amrfinder*. The steps recommended by the *MethPipe* authors were followed. First, the *FASTQ* reads were mapped to the reference genome with the *abismal* [30] tool. After that, the *SAM* [31] file produced went through several *MethPipe*-specific steps:

1. The utility tool *format_reads* was used to adapt the format to the pipeline.
2. The external command *samtools sort* [32] was used to sort the reads by chromosome and position.
3. The *MethPipe* tool *duplicate-remover* was used to remove duplicated reads.
4. Finally, the *methstates* tool was used to convert the resulting *sam* file to the *epiread* format.

For dataset *ERS208315*, as the raw data were provided as *unaligned BAM* [31], a couple extra steps were needed. First, the *uBAM* files were converted to *FASTQ* with

[2] https://github.com/UDC-GAC/PARamrfinder

[3] https://www.ncbi.nlm.nih.gov/sra

[4] https://ega-archive.org/datasets/EGAD00001002523

Table 2 Execution times for the original and optimized version of *amrfinder* (in seconds) and speedup varying the maximum number of iterations to fit the statistical models

Dataset	Version	-i 10	-i 100	-i 1000
ERS2586503	Original	4,152	26,484	153,887
	Optimized	1,890	11,069	63,342
	Speedup	2.20	2.39	2.43
ERS4575883	Original	8,124	58,332	–
	Optimized	3,718	25,964	167,814
	Speedup	2.18	2.25	–
ERS7819375	Original	4,031	12,191	45,779
	Optimized	2,349	6,786	25,525
	Speedup	1.72	1.80	1.79
ERS208315	Original	4,152	26,484	–
	Optimized	1,890	11,069	–
	Speedup	2.20	2.39	–

the command *samtools bam2fq*. Then, the *FASTQ* file was processed as the others but adding an additional step: the *samtools merge* command had to be used before *methstates* as this dataset is composed of several runs.

Table 1 summarizes the characteristics of these datasets. It includes information about the reference genomes, the size of the derived *epiread* files and the number of reads inside these files. Note that the *epiread* files can reach up to several gigabytes and hundreds of millions of reads.

5.1 Experiments with the sequential tool

The first step of the experimental evaluation checks the impact of the sequential optimization presented in Sect. 4.1. The original *amrfinder* has been compared to an optimized sequential version, which consists of the same implementation as the original amrfinder, but with a slightly modified fitting function that uses the *ComputeAndStore* technique. The performance of the two versions has been measured using the default parameters and changing the value of the maximum number of iterations to fit the models (option *-i* in the command line). The tool has been tested with a maximum of 10 (default), 100 and 1000 iterations, to see the impact of this optimization as the execution of the statistical models gains relevance. Table 2 shows the execution time of the tool with and without the optimization. Some execution times are not shown in the table as they exceed the maximum execution time allowed in the cluster (72 h). The execution time with the optimization is always lower than the one without the optimization, achieving a reduction of around one half of the original time. Remark that these experiments were all carried out on a single core of the cluster as both implementations are sequential.

Table 3 Execution times (in seconds) for the pre-processing and post-processing phases of *PARamrfinder* (in seconds) varying the number of cores

Dataset	Phase	Baseline	32c	64c	128c	256c
ERS2586503	Pre-proc	134.69	7.65	4.93	3.18	1.81
	Post-proc	146.56	6.64	3.90	2.52	1.99
ERS4575883	Pre-proc	236.43	12.88	8.97	5.48	2.88
	Post-proc	81.44	5.33	3.45	2.16	1.65
ERS7819375	Pre-proc	202.49	12.80	8.27	5.53	3.07
	Post-proc	68.99	6.13	3.27	2.06	1.51
ERS208315	Pre-proc	3,068.30	100.53	53.37	29.1	15.20
	Post-proc	173.31	11.81	7.46	4.66	3.51

5.2 Evaluation of the parallel I/O optimizations

Table 3 shows the execution times (in seconds) obtained by *PARamrfinder* in the pre-processing and post-processing phases when using the parallel optimizations presented in Sect. 4.4 and 4.5, and compared to a sequential counterpart for a varying number of cores. During these phases, the main source of data to be processed are the *epiread* file and the *FASTA* files respectively, both depicted in Fig. 1, whose size for every dataset has been specified in Table 1. Concretely, the table shows the baseline times for both phases and the parallel times when using from one whole node (32 cores) to eight nodes (256 cores). Remark that Hyperthreading is enabled, allowing 64 threads per node (two logical threads per core). The maximum number of iterations to fit the statistical models is left as default, as it does not affect these phases. As the structure of the pre-processing phase has been modified from the original tool (see Sect. 4.4), using it as a baseline would be unfair. Instead, for this experiment the baseline is the execution time of *PARamrfinder* with one process and one thread. For the post-processing phase the baseline is the execution time of this phase on the original tool using the same configuration.

These results are satisfactory, as the bottleneck of both the pre-processing and the post-processing phases are eliminated and the execution times are reduced from hundreds to less than five seconds in almost all the cases. There is even one positive exception: the execution time of the pre-processing phase of the dataset *ERS208315*, which scales much better than the other cases. This happens because the sequential time of this stage is much higher that the rest, and, when parallelized using all the 256 cores, this phase still takes more than 15 s to execute, so the synchronizations and overheads introduced in the parallel version have its impact thinned out.

5.3 Evaluation of the load balancing algorithms

A key point in the performance of *PARamrfinder* is its ability to balance the workload. As it was explained in Sect. 4.3, three different algorithms have been implemented in the parallel tool to distribute the regions among the MPI processes.

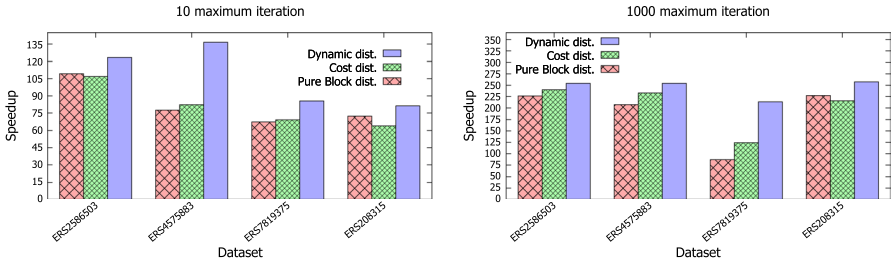


Fig. 7 Speedup of *PARamrfinder* over *amrfinder* (optimized) using three load balancing algorithms, 10 and 1000 maximum iterations, the different datasets and eight nodes of the cluster (256 cores)

1. **Pure block distribution** Contiguous blocks with the same number of regions are distributed among MPI processes.
2. **Cost-based distribution** Contiguous blocks of regions are distributed among MPI processes. The number of regions per block depends on their computational cost.
3. **Dynamic distribution** Blocks of regions are distributed among MPI processes dynamically, as they are processed, using RMA shared memory.

Figure 7 shows the speedups obtained by the three algorithms when using 256 cores, 10 and 1000 maximum iterations to fit the statistical models, and all the datasets available. All executions have taken advantage of Hyperthreading, with two logical threads per CPU (512 logical threads in total). From now on, all executions will take advantage of Hyperthreading. The baseline is the execution time of the original tool with the sequential optimization explained in Sect. 4.1. As the base execution time of the *ERS208315* dataset cannot be computed for 1000 maximum iterations due to the execution time limit of 72 h, a reference execution time has been estimated for that dataset assuming a speedup of 32x for the execution of *PARamrfinder* with a dynamic distribution using 32 cores (one whole node) and 1000 maximum iterations.

The dynamic algorithm provides the best performance in all cases, close to ideal speedups for three datasets with the maximum of 1000 iterations (x254.1 on *ERS2586503*, x254.2 on *ERS4575883*, x257.3 on *ERS208315*). It is also remarkable that it is the most consistent algorithm, as the other ones are more sensitive to the specific characteristics of the dataset. For example, the static algorithms significantly reduce their performance for dataset *ERS7819375* even with a maximum of 1000 iterations. This can be explained by a couple of factors. First, most of the regions in this dataset need a low amount of iterations to fit the statistical models, which makes the difference between these regions and the ones that require the maximum number of iterations bigger than in the other datasets. In addition, the average region in this dataset contains 2–40 times less information than the ones in other datasets; while, the *elephant regions* contain 2–5 times more information. This means that *elephant regions* gain

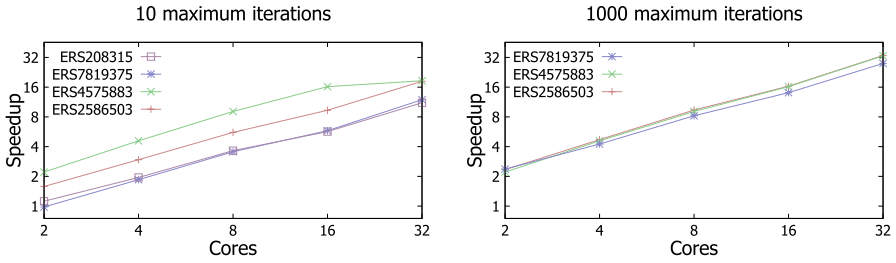


Fig. 8 Speedup of *PARamrfinder* over *amrfinder* optimized using 10 and 1000 maximum iterations for the different datasets varying the number of cores in a single node

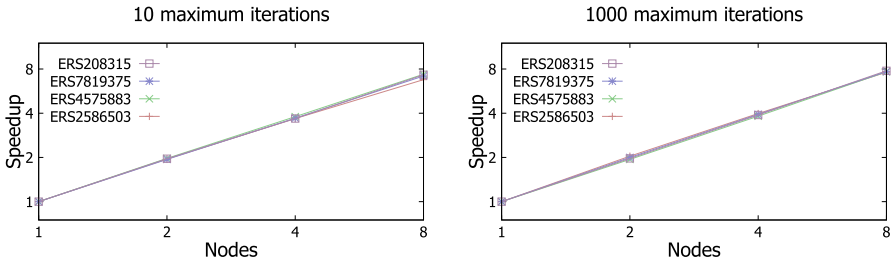


Fig. 9 Speedup of *PARamrfinder* using 10 and 1000 maximum iterations for the different datasets varying the number of nodes. The baseline is the execution time of PARamrfinder on a whole node

even more relevance and become the bottleneck of the execution if not treated carefully. However, the dynamic algorithm is able to overcome these factors and still achieves high speedup values. Therefore, this will be the algorithm included in the final version of the parallel tool, and the one used in the scalability experiments presented in the next section.

5.4 Scalability of *PARamrfinder*

The scalability test started by analyzing performance within one node, using one process per CPU and 2, 4, 8 and 16 cores per process (32 cores in total). This two-processes per node configuration was chosen because it is the one that provides the best performance in one node, as it improves memory bandwidth as well as locality, and will be maintained when increasing the number of nodes. Figure 8 shows the speedups obtained by *PARamrfinder* when using 10 and 1000 maximum iterations to fit the statistical models, all the datasets available, the dynamic distribution and a varying number of cores within one node. The baseline is again the execution time of the original tool with the sequential optimization.

It can be seen that *PARamrfinder* scales well with the number of cores, maintaining superlinear speedups when filling a node for datasets *ERS2586503*

Table 4 Execution times (in seconds) of *amrfinder* and *PARamrfinder* using different resources for 1000 maximum iterations to fit the statistical models. Every execution of *PARamrfinder* uses Hyperthreading with two logical threads per core

Dataset	amrfinder	PARamrfinder			
	1 core	1 core	32 cores (1 node)	128 cores (4 nodes)	256 cores (8 nodes)
ERS2586503	123,887	51,474	1542	404	202
ERS4575883	–	136,364	3901	980	509
ERS7819375	38,779	20,261	728	185	94
ERS208315	–	–	16,408	4218	2089

and *ERS4575883* for 1000 maximum iterations. It can also be noted that the speedups obtained by the tool are much higher when using 1000 maximum iterations than when using 10 maximum iterations, which implies that the tool performs better for heavy workloads. This is because the tool's pre-processing and post-processing phases gain relevance when the workload is smaller, and they do not scale as well as the identification phase with the dynamic distribution. In addition, the execution time of the tool for 10 maximum iterations gets reduced to less than two minutes for most of the datasets, so small overheads also gain relevance in these reduced runtimes.

Figure 9 shows the speedups obtained by *PARamrfinder* when using 10 and 1000 maximum iterations to fit the statistical models, the dynamic distribution, all the datasets available and 1, 2, 4 and 8 nodes. The baseline is the execution time of the parallel tool in one node. These results prove that *PARamrfinder* scales well with the number of nodes, and that its scalability is consistent for all the datasets.

Most of parallel bioinformatics applications only implement a multithreaded parallelization. If their users have access to a cluster, they can launch multiple jobs, one per node, and each job focused on analyzing a different dataset. This way they would have several nodes of the cluster working at the same time. The difference between this approach and a tool as *PARarmfinder* is that our tool, thanks to the MPI processes, can use all the nodes to collaborate on the analysis of the same dataset. To further justify the impact of MPI in the performance of the tool, it has been compared to the previously explained approach: an scenario with four different jobs executed simultaneously on different nodes, each one over a different dataset and exploiting the whole node thanks to the OpenMP implementation and Hyperthreading. The wall time of that experiment has then been compared against executing *PARamrfinder* using four nodes on the four datasets, one after another. The results can be extracted from fourth and fifth columns of Table 4, which shows the execution times of each dataset in both scenarios. For the OpenMP-only scenario the total execution time is the runtime of the biggest dataset (16,408 s). On the other hand, the execution time of the MPI+OpenMP version of the tool is the addition of the runtimes of each of the datasets (5,787 s). That is, the hybrid version of *PARamrfinder* is 2.84 times faster than the OpenMP-only execution in this scenario.

These experimental results prove that *PARamrfinder* can be useful for scientists in order to dramatically reduce the runtime needed to identify AMRs. Table 4 provides a summary of this runtime reduction when using 1000 maximum iterations

Table 5 Memory consumption (in GBs) for the pre-processing, processing and post-processing phases of *PARamrfinder* compared with *amrfinder*

Dataset	Phase	amrfinder	PARamrfinder
ERS2586503	Pre-proc	0.51	0.32
	Proc		0.49
	Post-proc	3.0	3.0
ERS4575883	Pre-proc	0.75	0.69
	Proc		0.49
	Post-proc	2.7	2.7
ERS7819375	Pre-proc	0.59	0.59
	Proc		1.63
	Post-proc	2.6	2.6
ERS208315	Pre-proc	3.48	3.52
	Proc		2.26
	Post-proc	3.1	3.1

Results for PARamrfinder indicate the maximum memory consumption per process for each phase

to fit the statistical models. It shows that *PARamrfinder* is faster than the original *amrfinder* even when using the same hardware (one core) due to two reasons. On the one hand, *PARamrfinder* can take advantage of Hyperthreading on that core by launching two logical threads. On the other hand, the *ComputeAndStore* technique presented in Sect. 4.1 reduces the execution time almost to half in every situation. Furthermore, the new parallel tool allows the completion of analyses that were not possible with the original tool. For instance, *amrfinder* was not able to work over the *ERS4575883* and *ERS208315* datasets in the maximum of three days allowed by the cluster. Nevertheless, *PARamrfinder* finishes these analyses in less than 9 and 45 min, respectively, using eight nodes. Finally, *PARamrfinder* is highly flexible and has been focused on maintaining high performance in every dataset, no matter how initially unbalanced it is.

5.5 *PARamrfinder* memory requirements

In addition to the performance evaluation, the memory usage of *PARamrfinder* has been analyzed, as it can be a critical factor in the execution of bioinformatics and high performance applications. The memory consumption of the tool using eight nodes and two processes per node has been measured and compared to the memory consumption of the original tool during the different stages of the execution. So the maximum memory requirements of the tool is the memory consumption of the phase with the highest memory usage (not the accumulation of all the memories, as after each phase, the main memory buffers are freed).

Table 5 shows the memory consumption of *amrfinder* and *PARamrfinder* during the different execution stages for the four datasets. Results for *PARamrfinder* indicate the maximum memory consumption per process for each phase. For the original tool, the pre-processing and processing phases are overlapped, as it pre-processes

and then processes one chromosome at a time. Because of this only one memory consumption value is shown for these two phases. During these phases *amrfinder* mainly uses memory to store the *epiread* file and the resulting AMRs for each chromosome. That is, the memory requirements of the tool are directly proportional to the number of reads for a chromosome in the *epiread* file and the number of CpG positions in that chromosome (positions the window has to scan per AMR). Something similar happens with *PARamrfinder* during the pre-processing phase, except for a pair of differences. First, a raw block of the input file is brought to memory by each process, which means a memory overhead. Second, a chromosome is not needed to be fully processed by one process, so it can be split among several processes, potentially reducing the memory requirements. Regarding the processing phase of *PARamrfinder*, the main memory consumption is only on the root process, which keeps a buffer with all the regions of all the chromosomes that have to be processed and another to store the results. However, only a small part (a few MB at most) of the *epiread* file is brought to memory by each process, so the memory overhead is reduced during this phase. Finally, note that on both tools these buffers are freed, keeping only the identified AMRs for the post-processing phase. During this phase the results have to be mapped to the reference genome, so the FASTA files have to be read. The memory consumption of *amrfinder* and each process of *PARamrfinder* is equal, as both tools bring all the FASTA files to memory at some point.

These results show that, for small and medium datasets, the memory bottleneck of *PARamrfinder* is the post-processing phase, which is the same as the original tool. However, as the size of the *epiread* file increases, the memory bottleneck of *PARamrfinder* becomes the pre-processing phase, as holding the reads in memory leads to a bigger requirement than storing the reference genome. Anyway, in all scenarios the maximum memory requirements per process of *PARamrfinder* are equal or just slightly higher than those of the original tool.

6 Conclusions

Nowadays, one interesting goal in DNA methylation studies consists in detecting AMRs under different biological conditions, which can help to understand the function of genomic imprinting. However, these analyses may take a huge time for large or even medium size datasets. In this work we have presented *PARamrfinder*, a parallel application that obtains the same biological results as the popular *amrfinder* tool, but at significantly reduced runtime thanks to exploiting the hardware of modern multicore clusters.

PARamrfinder is based on a hybrid MPI/OpenMP parallel implementation, which brings significant benefits, such as the capability to use Hyperthreading, efficient memory management and fewer required synchronizations among processing elements. The parallel tool is able to obtain great scalability thanks to its dynamic workload balance both at the process and the thread levels. In addition, the dynamic distribution at the process level has been implemented with a minimum overhead

as it uses MPI RMA operations to reduce the impact of communications and synchronizations.

The experimental evaluation was performed on a cluster with eight nodes, each one with 32 CPU cores (a total of 256 cores), using four representative datasets with real biological data and different characteristics. *PARamrfinder* is faster than *amrfinder* in all scenarios, even using the same hardware resources (one core). Its impact is more remarkable for a large number of resources, being able to reduce an execution from several days (more than 72 h) to less than nine minutes.

As future work we plan to apply similar parallel approaches to other stages of the *MethPipe* pipeline, so that the different stages could be integrated in order to exploit altogether the resources of a multicore cluster.

Acknowledgements This study makes use of data generated by the Blueprint Consortium, the consortium of researchers that was funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no 282510 BLUEPRINT.

Author Contributions Conceptualization contributed by JGD, MJM; methodology contributed by AFF; formal analysis and investigation contributed by AFF; writing—original draft preparation contributed by AFF; writing—review and editing contributed by JGD, MJM; supervision contributed by JGD, MJM; funding acquisition contributed by MJM.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work was supported by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00 and PID2022-136435NB-I00 / AEI / 10.13039/501100011033), PID2022 also funded by "ERDF A way of making Europe". It was also supported by the Ministry of Universities of Spain under grant FPU21/03408, and by Xunta de Galicia and FEDER funds (Centro de Investigación de Galicia accreditation 2019–2022 and Consolidation Program of Competitive Reference Groups, under Grants ED431G 2019/01 and ED431C 2021/30, respectively).

Availability of data and materials The datasets ERS2586503, ERS4575883 and ERS7819375 used in this study are available in the National Center for Biotechnology Information (NCBI) SRA repository (https://www.ncbi.nlm.nih.gov/sra/?term=ERS2586503+AND+Illumina+HiSeq+4000%5BTitle%5D, https://www.ncbi.nlm.nih.gov/sra/?term=ERS4575883 and https://www.ncbi.nlm.nih.gov/sra/?term=ERS7819375, respectively).), while the dataset ERS208315 has been provided by the Blueprint Consortium under request.

Declarations

Conflict of interest The authors declare that they have no competing interests.

Ethics approval Not applicable.

Consent for publication The authors have gone through the publication policies and have submitted the manuscript accordingly.

Code availability The source code of the *PARamrfinder* tool is available at https://github.com/UDC-GAC/PARamrfinder.

References

1. Nicholls RD, Knoll JH, Butler MG, Karam S, Lalande M (1989) Genetic imprinting suggested by maternal heterodisomy in non-deletion Prader–Willi syndrome. Nature 342(6247):281–285
2. Mabb AM, Judson MC, Zylka MJ, Philpot BD (2011) Angelman syndrome: insights into genomic imprinting and neurodevelopmental phenotypes. Trends Neurosci 34(6):293–303
3. Weksberg R, Smith AC, Squire J, Sadowski P (2003) Beckwith–Wiedemann syndrome demonstrates a role for epigenetic control of normal development. Human Mol Genet 12(suppl_1):61–68
4. Fang F, Hodges E, Molaro A, Dean M, Hannon GJ, Smith AD (2012) Genomic landscape of human allele-specific DNA methylation. Proc Natl Acad Sci 109(19):7332–7337. https://doi.org/10.1073/pnas.1201310109
5. Song Q, Decato B, Hong EE, Zhou M, Fang F, Qu J, Garvin T, Kessler M, Zhou J, Smith AD (2013) A reference methylome database and analysis pipeline to facilitate integrative and comparative epigenomics. PLoS ONE 8(12):81148
6. Okae H, Chiba H, Hiura H, Hamada H, Sato A, Utsunomiya T, Kikuchi H, Yoshida H, Tanaka A, Suyama M, Arima T (2014) Genome-wide analysis of DNA methylation dynamics during early human development. PLoS Genet 10(12):1–12. https://doi.org/10.1371/journal.pgen.1004868
7. Do C, Shearer A, Suzuki M, Terry MB, Gelernter J, Greally JM, Tycko B (2017) Genetic-epigenetic interactions in cis: a major focus in the post-GWAS era. Genome Biol. https://doi.org/10.1186/s13059-017-1250-y
8. Onuchic V, Lurie E, Carrero I, Pawliczek P, Patel RY, Rozowsky J, Galeev T, Huang Z, Altshuler RC, Zhang Z, Harris RA, Coarfa C, Ashmore L, Bertol JW, Fakhouri WD, Yu F, Kellis M, Gerstein M, Milosavljevic A (2018) Allele-specific epigenome maps reveal sequence-dependent stochastic switching at regulatory loci. Science 361(6409):3146. https://doi.org/10.1126/science.aar3146
9. Hu Y, Yuan S, Du X, Liu J, Zhou W, Wei F (2023) Comparative analysis reveals epigenomic evolution related to species traits and genomic imprinting in mammals. Innovation 4(3)
10. Marshall H, Jones AR, Lonsdale ZN, Mallon EB (2020) Bumblebee workers show differences in allele-specific DNA methylation and allele-specific expression. Genome Biol Evol 12(8):1471–1481
11. Benton MC, Lea RA, Macartney-Coxson D, Sutherland HG, White N, Kennedy D, Mengersen K, Haupt LM, Griffiths LR (2019) Genome-wide allele-specific methylation is enriched at gene regulatory regions in a multi-generation pedigree from the Norfolk Island isolate. Epigenetics Chromatin 12(1):1–10
12. Message Passing Interface Forum (2021) MPI: A Message-Passing Interface Standard Version 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
13. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55
14. Reyes-Ortiz JL, Oneto L, Anguita D (2015) Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. Procedia Comput Sci 53:121–130
15. Tycko B (2010) Allele-specific DNA methylation: beyond imprinting. Hum Mol Genet 19(R2):210–220. https://doi.org/10.1093/hmg/ddq376
16. Zhang Y, Rohde C, Reinhardt R, Voelcker-Rehage C, Jeltsch A (2009) Non-imprinted allele-specific DNA methylation on human autosomes. Genome Biol 10:1–11
17. Down TA, Rakyan VK, Turner DJ, Flicek P, Li H, Kulesha E, Graef S, Johnson N, Herrero J, Tomazou EM et al (2008) A Bayesian deconvolution strategy for immunoprecipitation-based DNA methylome analysis. Nat Biotechnol 26(7):779–785
18. Zhou Q, Guan P, Zhu Z, Cheng S, Zhou C, Wang H, Xu Q, Sung W-K, Li G (2021) ASMdb: a comprehensive database for allele-specific DNA methylation in diverse organisms. Nucleic Acids Res 50(D1):60–71. https://doi.org/10.1093/nar/gkab937
19. Liu Y, Siegmund KD, Laird PW, Berman BP (2012) Bis-SNP: combined DNA methylation and SNP calling for Bisulfite-seq data. Genome Biol 13(7):1–14

20. Andergassen D, Dotter CP, Kulinski TM, Guenzl PM, Bammer PC, Barlow DP, Pauler FM, Hudson QJ (2015) Allelome.PRO, a pipeline to define allele-specific genomic features from high-throughput sequencing data. Nucleic Acids Res 43(21):146. https://doi.org/10.1093/nar/gkv727
21. Guo W, Zhu P, Pellegrini M, Zhang MQ, Wang X, Ni Z (2017) CGmapTools improves the precision of heterozygous SNV calls and supports allele-specific methylation detection and visualization in bisulfite-sequencing data. Bioinformatics 34(3):381–387. https://doi.org/10.1093/bioinformatics/btx595
22. Orjuela S, Machlab D, Menigatti M, Marra G, Robinson MD (2020) DAMEfinder: a method to detect differential allele-specific methylation. Epigenet Chromatin 13(1):1–19
23. Minh BQ, Vinh LS, Von Haeseler A, Schmidt HA (2005) pIQPNNI: parallel reconstruction of large maximum likelihood phylogenies. Bioinformatics 21(19):3794–3796
24. Gonzalez-Dominguez J, Martin MJ (2017) MPIGeneNet: parallel calculation of gene co-expression networks on multicore clusters. IEEE/ACM Trans Comput Biol Bioinf 15(5):1732–1737
25. Li K-B (2003) ClustalW-MPI: ClustalW analysis using distributed and parallel computing. Bioinformatics 19(12):1585–1586
26. Parker HR, Orjuela S, Martinho Oliveira A, Cereatti F, Sauter M, Heinrich H, Tanzi G, Weber A, Komminoth P, Vavricka S et al (2018) The proto CpG island methylator phenotype of sessile serrated adenomas/polyps. Epigenetics 13(10–11):1088–1105
27. Sönmezer C, Kleinendorst R, Imanci D, Barzaghi G, Villacorta L, Schübeler D, Benes V, Molina N, Krebs AR (2021) Molecular co-occupancy identifies transcription factor binding cooperativity in vivo. Mol Cell 81(2):255–267
28. Radic Shechter K, Kafkia E, Zirngibl K, Gawrzak S, Alladin A, Machado D, Lüchtenborg C, Sévin DC, Brügger B, Patil KR et al (2021) Metabolic memory underlying minimal residual disease in breast cancer. Mol Syst Biol 17(10):10141
29. Cock PJ, Fields CJ, Goto N, Heuer ML, Rice PM (2010) The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. Nucleic Acids Res 38(6):1767–1771
30. Sena Brandine G, Smith AD (2021) Fast and memory-efficient mapping of short bisulfite sequencing reads using a two-letter alphabet. NAR Genom Bioinform 3(4):115
31. Li H, Handsaker B, Wysoker A, Fennell T, Ruan J, Homer N, Marth G, Abecasis G, Durbin R, Subgroup GPDP (2009) The sequence alignment/Map format and SAMtools. Bioinformatics 25(16):2078–2079. https://doi.org/10.1093/bioinformatics/btp352
32. Danecek P, Bonfield JK, Liddle J, Marshall J, Ohan V, Pollard MO, Whitwham A, Keane T (2021) McCarthy, S.A., Davies, R.M., Li, H.: Twelve years of SAMtools and BCFtools. GigaScience 10(2).https://doi.org/10.1093/gigascience/giab008