Applying dynamic balancing to improve the performance of MPI parallel genomics applications

Alejandro Fernández-Fraga
CITIC, Universidade da Coruña
A Coruña, Galicia, Spain
a.fernandez3@udc.es

Jorge González-Domínguez
CITIC, Universidade da Coruña
A Coruña, Galicia, Spain
jgonzalezd@udc.es

María J. Martín
CITIC, Universidade da Coruña
A Coruña, Galicia, Spain
mariam@udc.es

ABSTRACT

Genomics applications are becoming more and more important in the field of bioinformatics, as they allow researchers to extract meaningful information from the huge amount of data generated by the new sequencing technologies. The analysis of these data is a very time consuming task and, therefore, the use of High Performance Computing (HPC) and parallel processing techniques is essential. Although the structure of these applications can be easily adapted to parallel systems by distributing the data to be processed among the available processors, load imbalance is a usual cause of performance degradation. In this paper we propose a dynamic load balancing method based on MPI RMA one-sided communications to minimize the synchronization among processes and the overhead due to communications while improving the workload balance. The strategy is applied, as a case study, to *ParRADMeth*, an MPI/OpenMP parallel application for the identification of Differential Methylated Regions (DMRs). Results show that the new version of the tool outperforms the previous one in all cases, achieving high performance and scalability. For example, our approach is up to 243 times faster than the sequential version and 1.74 times faster than the previous parallel version when processing a real dataset on a cluster with 8 nodes, each one with 32 CPU cores.

CCS CONCEPTS

• **Applied computing** → **Computational genomics**; • **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Differential Methylation, MPI, OpenMP, RMA, Dynamic Load Balancing

ACM Reference Format:
Alejandro Fernández-Fraga, Jorge González-Domínguez, and María J. Martín. 2024. Applying dynamic balancing to improve the performance of MPI parallel genomics applications. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3605098.3635986

1 INTRODUCTION

In the last decade, genomics datasets growth has been impressive, doubling approximately every seven months [20]. Thanks to the advancements in sequencing technologies, whose capacities are expected to continue growing rapidly in the next years, estimates predict that this field will generate between 2 and 40 exabytes of data within the next decade [1]. Then, High Performance Computing (HPC) and parallel processing techniques become essential for researchers to apply the powerful computational and statistical methods needed to extract meaningful information out of this huge amount of data in a reasonable time.

In order to use parallel processing techniques the work must be decomposed into tasks that can be executed concurrently. When working with genomic data, these tasks usually have heterogeneous processing costs due to the variability of biological sequence sizes and the use of iterative algorithms to process the data. Therefore, dynamic task distributions are needed to avoid load balancing problems and poor performance.

MPI is the de facto standard for programming distributed-memory systems and the most widely used programming framework in the HPC community. Traditionally dynamic load balancing in MPI applications is implemented through the master-worker method, using two-sided communications for the exchange of information. However, this method also implies performance degradation, as it introduces many synchronization points, which generate an important overhead as the number of MPI processes increases, making it a non-scalable solution.

In this paper we propose a dynamic load balancing method based on the use of the MPI RMA one-sided communications included in MPI-3. It minimizes the synchronization among processes and the overhead due to communications, overcoming the limitations of the classical dynamic load balancing methods. The strategy is applied, as a case study, to *ParRADMeth* [8], a parallel genomics application for the identification of Differential Methylated Regions (DMRs).

2 RELATED WORK

MPI has been used for decades to reduce the execution time of bioinformatics applications. The proposals found in the literature can be divided into three main groups depending on how the workload distribution is performed: those that use a static distribution; those based on a hybrid scheme with MPI and multithreading (and apply a static distribution at the MPI process level and a dynamic distribution at the thread level); and those that use a dynamic distribution already at process level.

Within the first group we can find applications such as *pblatcluster* [21], a hybrid MPI/pthreads sequence alignment tool that divides the input FASTA file to be aligned into as many parts as MPI processes are available. Although it is able to significantly reduce

[1] https://www.genome.gov/about-genomics/fact-sheets/Genomic-Data-Science

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04…$15.00
https://doi.org/10.1145/3605098.3635986

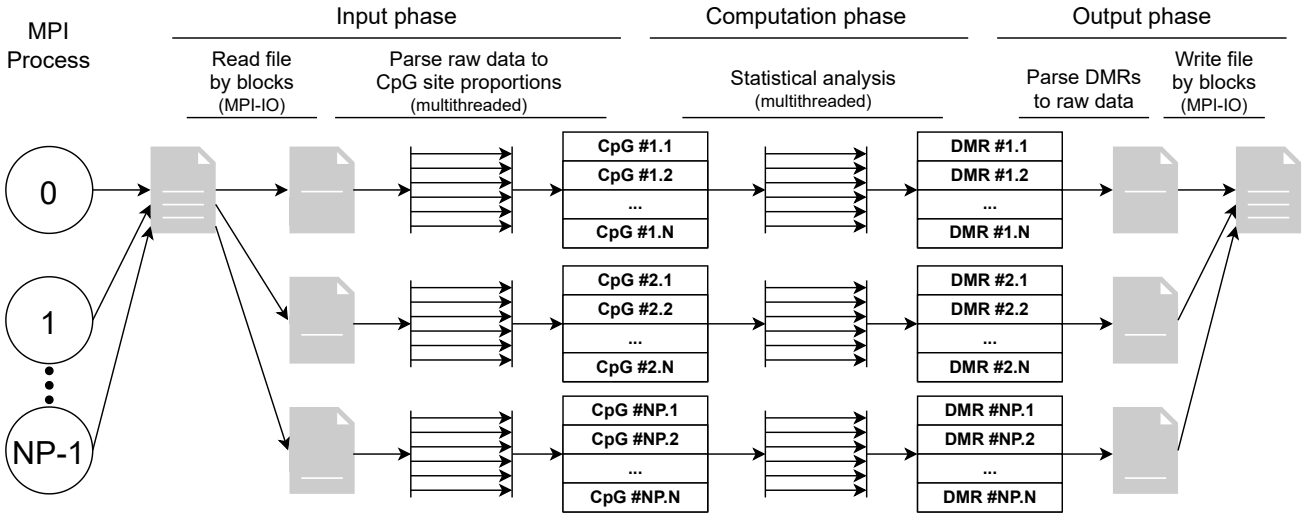

Figure 1: Workflow of *ParRADMeth*

the execution time of the sequential code, the achieved speedup is not proportional to the number of available processes due to imbalance issues related with the length of the sequences being processed. That is, the slowest process, the one with the longest sequences, determines the overall execution time.

Examples of the second group are *EPISNPmpi* [17], a hybrid MPI/OpenMP tool to accelerate epistasis detection in large-Scale GWAS studies; *MPIGeneNet* [9], which uses these two programming models for the calculation of gene co-expression networks; or *ParRADMeth* [8], a parallel tool for the analysis of differential methylation, which is the case study of this paper. Results in [17] and [8] prove that the hybrid version of the tools are more scalable than the pure MPI codes.

The applications in the third group traditionally use the master-worker execution model to dynamically distribute the tasks among the MPI processes. Some examples of this group are *DiALIGNTX* [7], which applies an iterative heuristic method to align multiple biological sequences; *parMATT* [18], which designates a master process to distribute the pairwise structural comparison between proteins among the workers; or *Autodock4.Iga.MPI* [6], which uses this approach to rapidly perform Virtual High-Throughput Docking (VHTD) of massive databases against a variety of protein targets. On this last application, the overhead due to MPI operations means a third of the total overhead of the executions.

In this paper we propose the use of the MPI RMA one-sided communications to reduce the cost associated to the implementation of a dynamic distribution at the process level. Specifically, we will make use of shared-memory window creation, as well as atomic read-modify-write operations to build a shared queue of pending tasks that will be globally accessible by all the MPI processes. The MPI RMA one-sided communications have been also successfully applied to reduce the communication overhead of applications with irregular communications patterns [2, 16].

3 BACKGROUND: *PARRADMETH*

ParRADMeth is a high performance tool for the analysis of differential methylation. It is based on the tool *RADMeth*, which is part of the *MethPipe* framework [19] and it is widely used on genomics real-world analyses [4, 15]. Both tools analyze CpG sites (fragments of DNA where a cytosine (C) is followed by a guanine (G)) in a set of samples, using a statistical method based on the Beta-Binomial distribution to determine if they present different methylating patterns.

ParRADMeth uses a hybrid parallel approach that combines MPI processes and OpenMP threads. The workflow of the tool is shown in Figure 1, and can be divided into three phases. First, during the input phase, which is explained in detail in Section 3.1, each MPI process reads a block of contiguous bytes from the input file using MPI-IO functions. The whole file is allocate to memory at the same time. Then, each process parses its block to extract the CpG sites by using all the available OpenMP threads. After that, *ParRADMeth* enters the processing phase, the most time consuming part of the tool. In this phase each MPI process fits the statistical model to each of its CpG sites, dynamically distributing them among all its OpenMP threads, to determine if they are differentially methylated. Finally, the output phase (detailed in Section 3.2) is performed, where the resulting DMRs are parsed to raw data and written to the output file in parallel.

3.1 *ParRADMeth* input phase

The input file of *ParRADMeth* follows a text-based table format, where each line represents a CpG site and each column represents a sample. Figure 2 shows an example of this format, where the first line is the header containing the name of the samples and the rest of the lines represent the CpG sites. Each CpG site is represented by its chromosome, initial and final position, as well as the coverage and methylation level of each sample.

Algorithm 1: *ParRADMeth*'s parallel input phase pseudocode

Input: A string, $path\_to\_input\_file$, containing the path to the CpG sites table
Output: A list of CpG sites, $input\_cpg\_sites$, containing the input CpG sites for each process

```
1 input_file ← MPI_File_open(path_to_input_file)
2 filesize ← MPI_File_get_size(input_file)
  /* Figure out who reads what */
3 block_size ← filesize / number_of_processses
4 start_offset ← CalculateStartOffset(filesize)
5 overlap ← CalculateMaxLineLength(input_file)
  /* Read raw data */
6 input_block ← MPI_File_read_at_all(input_file, start_offset, block_size + overlap)
  /* Parse raw data */
7 input_cpg_sites ← ParseDataToAppropiateStructure(input_block)
8 MPI_File_close(input_file)
```

●C - coverage ●M - methylation level	control_1		control_2		case_1		case_2	
	C	M	C	M	C	M	C	M
chr1:10468:+:CpG	9	7	9	6	12	10	8	7
chr1:10470:+:CpG	23	22	25	22	12	11	16	14
chr1:10483:+:CpG	27	25	22	22	30	28	24	23
chr1:10488:+:CpG	30	28	41	36	24	23	35	32
chr1:10492:+:CpG	47	45	34	29	38	36	37	33

Figure 2: Example of the input file format with five CpG sites (rows) and four individuals (pairs of columns)

Due to this text-based format, each line in the input file has a different size (different chromosome name, numbers with different amount of digits, etc.), and hence the number of lines can not be known in advance just looking at the file size. However, each line contains the same number of columns and therefore it can be assumed that two blocks of the same size will contain a similar number of lines.

Taking these particularities of the input format into account, *ParRADMeth* reads the input file using MPI-IO functions, as shown in Algorithm 1. *ParRADMeth* processes all the input file at once, that is, it reads the whole file to memory and then it parses it to extract the CpG sites, following a series of steps. First, all the processes open the file in parallel (Line 1), and get its size in bytes (Line 2). Then the processes distribute the file by blocks of the same size. To achieve this they calculate the size of the blocks (Line 3), and their start offset (Line 4). In addition, to ensure that no line is split between two blocks, each block is overlapped with the next one. This overlapping size is calculated by overestimating the maximum size of a line of the input file (Line 5). With this information, each process reads its block and the overlapping bytes (Line 6) as raw text and, then, using all the available OpenMP threads, it parses the block into a list of the appropriate CpG site data structure (Line 7). Finally, processes close the file (Line 8).

3.2 *ParRADMeth* output phase

The output file also follows a text-based format, where each input CpG site becomes a line in the output file, indicating whether it presents differential methylation or not. To obtain the same output file than the one produced by the original tool *RADMeth*, the output phase of *ParRADMeth* follows the steps shown in Algorithm 2 to write the results as if they had been processed sequentially.

First, all the processes open the output file in parallel (Line 1). Then, each process parses the list of DMRs to the text representation that will be written to the output file (Line 2). After that, processes synchronize to ensure all of them have finished the processing phase, and share the length of its text block using *MPI_Allgather* (Line 3). With this information, each process calculates the offset at which it has to start writing (Lines 4-6). Finally, processes write their block in parallel, using *MPI_File_write_at_all* (Line 7) and close the output file (Line 8).

4 LOAD BALANCING PROBLEM

The main computational bottleneck of *ParRADMeth* is the statistical analysis of the CpG sites. This phase comes with a high workload imbalance, as several factors make the processing time of each CpG site different from the others:

- Most of the CpG sites do not need to fit the statistical models, as it may be obvious that they do not represent two different methylation patterns. This splits the CpG sites into two groups: those that do not need to be tested and take almost no time to be processed, and those that need to be tested, which are a minority but require a much longer time to be processed.
- The statistical method applied to the CpG sites that need to be tested is an iterative algorithm, which means that the number of iterations needed to converge is different for each CpG site. This implies that even within the minority of CpG sites that need to be tested, there is a great disparity in the processing time. In addition, this number of iterations is not known in advance, as it is not related to any observable feature of the CpG site, as it does happen in other applications where the

Algorithm 2: *ParRADMeth*'s parallel output phase pseudocode

Input: A list of DMRs, *my_output_DMRs*, containing the output results of the process
A string, *path_to_output_file*, containing a path to the output file
Output: The output file with every output block correctly written

```
1 output_file ← MPI_File_open(path_to_output_file)
2 my_output_block ← ToString(my_output_DMRs)
  /* Gather all output blocks lengths in every process */
3 output_block_lengths ← MPI_Allgather(my_output_block.length())
4 my_write_offset ← 0
5 for i ← 0 to my_rank do
6 |   my_write_offset ← my_write_offset + output_block_lengths[i]
  end
  /* Write in parallel to the output file */
7 MPI_File_write_at_all(output_file, my_output_block, my_write_offset)
8 MPI_File_close(output_file)
```

processing time may be related to the length of the sequence, for example.

Moreover, it is common for few CpG sites to be much more expensive to process than the rest. These few CpG sites represent a small percentage of the total, but they take a high percentage of the runtime of the program. As it happens with the number of iterations, this is not related to any observable feature of the CpG site.

Due to these factors, it is not possible to statically create blocks of CpG sites with similar computational requirements at the beginning of the execution and thus all static distributions introduce workload imbalance. Although the hybrid MPI/OpenMP parallel scheme helps mitigate the problem, the use of a dynamic distribution at the process level would achieve higher performance.

5 DYNAMIC LOAD BALANCING

The new version of *ParRADMeth*, *ParRADMeth-DB*, implements a modern dynamic load balancing algorithm based on the use of MPI RMA one-sided communications.

ParRADMeth-DB follows the same three steps shown in Figure 1, with the main difference being that now the input data is not read all at once, but on demand, in small blocks.

For a certain number of blocks, two structures are allocated in the shared memory of the root process to implement this approach:

- A shared index, initialized to zero, that indicates the next block of CpG sites to be processed.
- A shared array, with a length equal to the number of blocks, that contains the size of the text-based representation of the results of each block.

After initializing these structures and opening the input and output file in parallel, all processes enter a loop that repeats until all the CpG sites have been processed.

The workflow of *ParRADMeth-DB* is shown in Algorithm 3. First, the shared memory structures are initialized in the root process (Lines 1-2). Then, each process requests its initial block index and increments it to the next position atomically (Line 3). This behaviour is implemented using *MPI_Fetch_and_op*. Afterwards, the process enters a loop where it works on the blocks until they are out of bounds (Line 4). In the first place, the process gets the block of CpG sites from the input file (Line 5). The particular details of this step are explained in Section 5.1. Then, the process fits the statistical model to each CpG site, distributing the elements among all the available threads, and based on the results determines whether there are DMRs (Lines 6-9). After that, the process writes the results of the block to the output file (Lines 10-11). To do this, it first parses its DMRs into the raw bytes that will be written to the output file (Line 10), and then it inserts the raw bytes into the output queue (Line 11). The particularities of how the output queue communicates with the shared memory structures and tries to write the results to the output file are explained in Section 5.2. Finally, the process requests the next block index (Line 12). Once the loop ends, the processes synchronize and write the blocks that still remain in each other's queue to the output file (Line 13).

A critical factor for this algorithm to be as efficient as possible is the size of the blocks. If the blocks are too big, the imbalance among them may be too high, and then it will not solve the initial problem. However, if the blocks are too small, the overhead due to communications and synchronization may be too high. Therefore, the size of the blocks must be chosen carefully. After a preliminary evaluation, we have chosen to use a fixed number of 100 blocks per process, as it has been proven to be the best option. In addition, as the imbalance has a more critical effect as computing progresses, the size of the blocks is reduced as the index advances. In particular, the first half of the blocks are of the default size, half to third quarter of the blocks are half the default size, and the last quarter of the blocks are a quarter of the default size.

In addition, to implement this new algorithm, the input and output phases of the tool have been redesigned, as each process does not know in advance which data blocks it will need to compute, and then, reading the whole file at once is not advisable. This also comes with the advantage that now the file does not need to fit in memory, allowing *ParRADMeth-DB* to process larger datasets, even in systems with limited memory.

Algorithm 3: *ParRADMeth-DB*'s pseudocode

Input: A integer, *number_of_blocks*, containing the number of blocks the input file is divided into
A file handle, *input_handle*, to the input file
A queue, *output_queue*, where the results are inserted to be written to the output file

```
   /* Initialize shared memory structures on root process */
1  global_block_index ← Init_Shared_Index()
2  global_block_sizes ← Init_Shared_Array(number_of_blocks)
   /* Get initial index and start processing */
3  current_block ← Get_Block_Index_And_Update(global_block_index)
4  while current_block ≤ last_block_index do
5  |  input_cpg_sites ← Read_Block(current_block, input_handle)
6  |  output_dmrs.Clear()
7  |  foreach cpg in input_cpg_sites do
8  |  |  result ← Fit_Model(cpg)
9  |  |  output_dmrs.Append(result)
   |  end
10 |  raw_output ← ToString(output_dmrs)
11 |  Try_To_Write_Output_Block(raw_output, current_block, output_queue, global_block_sizes)
12 |  current_block ← Get_Block_Index_And_Update(global_block_index)
   end
13 Write_Remaining_Blocks(output_queue, global_block_sizes)
```

5.1 Block input

The input phase of *ParRADMeth* has been redesigned to read the input file in small blocks. Nevertheless, few changes have been made to the original version, as the desired behaviour is very similar. Known an initial offset and a size, the MPI processes read a block of bytes from the input file. The main difference is that now this is not done once, but several times during the execution of the tool. In addition, the initial offset and size of the blocks are not fixed, but they are calculated based on the index of the block. Another challenge is to ensure that no line is split between two blocks, as this would lead to an error in the parsing of the input file. However, this can be achieved by using the overlapping technique already used in the original version (see Section 3.1).

Algorithm 4 shows the pseudocode of the new function to read a block of CpG sites (Read_Block in Algorithm 3). First, the process calculates the initial offset and size of the block (Lines 1-3). These are fixed values for each block, and they are calculated based on the index of the block. In addition, the overlapping technique is applied by increasing the size of the block by the maximum size of a line. Then, the process reads the block from the input file (Line 5), and finally parses it to the list of CpG sites using all the available threads (Line 6).

5.2 Block output

The output phase of *ParRADMeth* has also been redesigned to write the results in small blocks. This redesign is more complex than the input one, as the output phase of the original version was executed after a process synchronization, and all the processes could share all the information about the results of the blocks. This synchronization is not possible in this new version, as the processes do need to write the results of the blocks while other blocks are still being computed.

This means that two problems arise with the new version of the tool. First, for the processes to know where to write their results, they need to know the size of the previous results. However, as the processes do not synchronize after every block, they need a new mechanism to share this information with the others. Second, even if the first problem was solved and once a block is processed its size was instantly communicated to all processes, a process may remain waiting while trying to write the results of a block B_n if it does still not know the size of the previous block B_{n-1}. That is, as blocks are processed dynamically and each one has a different associated runtime, the order in which they are read is not necessarily the order in which they finish their processing. That could lead to a situation where the block B_n has been processed but it can not be written, as its start offset is not known until the results of the blocks $[B_0, B_{n-1}]$ have been computed.

With these restrictions in mind a new output algorithm has been implemented, relying on shared memory and a buffering queue to overcome two problems, respectively. Algorithm 5 shows the pseudocode of the new output phase (Try_To_Write_Output_Block function in Algorithm 3). First, the process calculates the size of the block (Line 1), and then it inserts the block into the output queue (Line 2). After that, the process updates the size of the block in the shared memory (Line 3). That is, it locks the shared memory window with *MPI_Win_lock*, writes the size of the block B_n into the n^{th} position of the array using *MPI_Put*, reads the size of the previous blocks $[B_0, B_{n-1}]$ using *MPI_Get*, and finally unlocks the shared memory window with *MPI_Win_unlock*. This way, the process ensures that the other processes know that block B_n has been computed, and also checks if the size of blocks $[B_0, B_{n-1}]$ are already known. With this information, the process tries to write

Algorithm 4: *ParRADMeth-DB*'s block input pseudocode

Input: A file handle, *input_handle*, to the input file
A integer, *block_index*, containing the index of the block to be read
Output: A list of CpG sites, *input_cpg_sites*, containing the input CpG sites contained in the current block

```
/* Figure out where to start reading and how many bytes to read */
1 block_initial_offset ← Get_Start_Offset(block_index)
2 block_size ← Get_Block_Size(block_index)
3 block_size ← block_size + MAX_LINE_LENGTH
/* Read data */
4 raw_block ← MPI_File_read_at(input_handle, block_initial_offset, block_size)
5 input_cpg_sites ← Raw_Block_To_CpG_Sites(raw_block)
```

Algorithm 5: *ParRADMeth-DB*'s block output pseudocode

Input: A file handle, *output_handle*, to the output file
A integer, *block_id*, containing the index of the block to be written
A string, *raw_block*, containing the raw bytes of the result to be written
A queue, *output_queue*, containing the output blocks to be written
A shared memory array, *shared_block_sizes*, initialized to 0s containing the sizes of the blocks to be written

```
1 current_block_size ← raw_block.Size()
2 output_queue.Push(block_id, raw_block)
/* Update the size of the current block on the shared memory */
3 Update_Block_Size(shared_block_sizes, block_id, current_block_size)
4 while Previous_Block_Sizes_Are_Known(output_queue.Front().block_id) do
5 |   block_id, raw_block ← output_queue.Pop()
6 |   block_start_offset ← Calculate_Start_Offset(shared_block_sizes, block_id)
7 |   MPI_File_Write_at(output_handle, raw_block, raw_block.Size(), block_start_offset)
end
```

the results of the blocks in its queue to the output file. To do this, for each block in the queue, it checks if the size of the previous blocks is known (Line 4) and, if it is, it pops the block out of the queue (Line 5), calculates the offset at which the block should be written by adding the previous block sizes (Line 6), and writes the block to the output file using *MPI_File_write_at* (Line 7).

This algorithm ensures that the results of the blocks are written in the correct order, and that the processes do not remain waiting for the size of the previous blocks. However, it does not ensure that the results of the blocks are written as soon as they are computed. This means that after a process exits the main loop the output queue may contain blocks that have been processed but not written to the output file. To solve this, at the end of the loop, when it is ensured that all the blocks have already been processed, processes synchronize and read the shared memory array to write the remaining blocks in the queue to the output file (Line 13 in Algorithm 3).

6 EXPERIMENTAL RESULTS

This section provides a performance comparison of *ParRADMeth* and *ParRADMeth-DB* to illustrate the benefits of the new dynamic load balancing techniques. This evaluation has been performed on an eight-node cluster with 256 CPU cores (32 cores per node). Each node has two Intel Xeon Silver 4216 Cascade Lake-SP CPUs with 16 cores each that support *Hyperthreading* (up to 64 threads per node), and 256 GB of memory. The nodes are interconnected by a Infinitiband EDR network. The programs have been compiled with the GNU GCC compiler v.8.3.0, and the parallel versions of the tool have been linked to the OpenMPI library v.4.0.5.

The precision of the results is not evaluated in this work, since both versions return exactly the same results as the original *RADMeth*, whose high precision has been demonstrated and compared in previous works [10, 13, 14].

6.1 Datasets

Five different real biological datasets were used for this experimental evaluation. Table 1 summarizes the characteristics of these datasets, which are named according to the first author of the experiment where they were published. The four datasets from the original evaluation of *ParRADMeth* were mantained [1, 3, 11, 12], while the new *Xiongfeng* dataset, which compares the methylomes of patients with type 2 diabetic with control individuals [5], was added to this evaluation. Table 1 includes two columns with the size of the input file and the sequential time required by *RADMeth* to analyze these datasets, which can be more than 8 hours.

As it will be shown in the next subsection, the *Akalin* dataset is particularly challenging for *ParRADMeth*. This is due to the fact that some of the input CpGs are much more expensive to process that the rest. While most CpG sites take less than 0.1 seconds to process,

Table 1: Datasets specification

Dataset	#CpG sites	#Samples	Size input file (GB)	Seq. time (s)
Akalin	28,670,426	2	0.82	6,894
Heyn	28,299,639	2	0.87	12,147
Berman	28,149,963	2	0.88	26,984
Hansen	28,217,449	6	1.4	30,027
Xiongfeng	28,532,123	18	3.3	29,744

a few of them need more than 20 seconds. This, joined to the fact that these CpGs sites appear at the end of the file, leads to a very high workload imbalance at the end of the processing loop, with a very low margin to balance it. This is a very challenging scenario both for the original and the dynamic load balancing algorithm, and the reason why the *Akalin* dataset obtains the worst scalability.

6.2 Performance Evaluation

The threads-to-process ratio has a huge impact on the performance of the tool. For *ParRADMeth* the higher this ratio the better, as it only applies a dynamic distribution that alleviates the workload imbalance at thread level. Therefore, the best configuration for *ParRADMeth* is to use one process per node, and the maximum number of threads per process. However, this is not true for *ParRADMeth-DB*, as it relies on both processes and threads to balance the workload. After a preliminar experimental evaluation, it was determined that the best configuration is two processes per node, as this is the number of sockets per node.

Figure 3 shows the speedups over the sequential tool *RADMeth* of the new and the original version of *ParRADMeth*, for a varying number of nodes (32 cores per node), using *Hyperthreading* and the best configuration of processes and threads for both tools. *ParRADMeth-DB* improves the scalability of the original version in all the datasets. However, the impact is not the same for all datasets.

First, for the *Berman*, *Hansen* and *Xiongfeng* datasets, where *ParRADMeth* already obtained almost ideal speedups, *ParRADMeth-DB* still improves the performance, showing that, even in these cases where the workload is fairly balanced, the dynamic load balancing brings benefits as it reduces the overheads and synchronizations.

Second, for the *Akalin* dataset, where the workload imbalance is extreme, *ParRADMeth-DB* improves the performance of the original version, even though the speedups are still not as high as in the other datasets, due to the reasons already discussed at the beginning of this section.

Finally, for the *Heyn* dataset, where the original version did not obtain as high values, the new version of the tool obtains the best results, showing that the dynamic load balancing is able to overcome the workload imbalance and obtain almost ideal speedups. That is, for this dataset the new version of *ParRADMeth* is able to obtain as high performance as it does on the fairly balanced datasets.

This proves that the dynamic load balancing at process level alleviates the workload imbalance and does not introduce significant overheads but instead brings benefits to the tool that contributes to achieving high performance. Table 2 summarizes the impact of these benefits in terms of execution time. This translates into a reduction of the execution time of the *Heyn* dataset from 3 hours and 22 minutes, to just 50 seconds. In addition, when compared to *ParRADMeth* for this dataset on eight nodes, the execution time is reduced by 37 seconds, which represents a 42% of its runtime.

Table 2: Execution times for different versions of the tool (in seconds)

Dataset	RADMeth	ParRADMeth		ParRADMeth-DB	
	1 core	1 node	8 nodes	1 node	8 nodes
Akalin	6,894	249	77	271	64
Heyn	12,147	415	88	392	51
Berman	26,984	846	138	840	111
Hansen	30,027	874	159	938	131
Xiongfeng	29,744	888	155	950	125

7 CONCLUSIONS

In this paper we have presented a low overhead dynamic load balancing algorithm from which several genomics applications can benefit. This approach has been implemented in a new version of the tool *ParRADMeth*, which is a parallel tool that detects DMRs in methylomes. The new version of the tool has been evaluated in terms of performance and scalability on an eight-node cluster with 32 CPU cores per node (256 CPU cores in total), using five different real biological datasets with different characteristics.

The results show that, while the original *ParRADMeth* obtains a better performance in one node for some datasets, the new version of the tool obtains a significant improvement in all the datasets when scaling to multiple nodes. For example, for the *Heyn* dataset *ParallelTool-DB* achieves an almost ideal performance, while the original version did not obtain a good scalability. In addition, the approach has also proven to not introduce significant overheads in the scenarios where the original tool was already obtaining high performance. For example, the *Berman* dataset which is fairly balanced and already had a great scalability in the original tool (210 Speedup), is even outperformed by the newer version (243 Speedup), reducing the execution time to less than two minutes.

These experimental results prove that this dynamic approach is a good candidate to be used in other bioinformatics applications, as it is able to achieve fair distributions in the imbalanced datasets, while still matching the performance of the static version in the balanced ones. As future work, we plan to apply this approach to other bioinformatics applications, focusing on other stages of the *MethPipe* pipeline, and other genomics tools for the analysis of DNA methylation.

ACKNOWLEDGMENTS

This work has been supported by grants PID2019-104184RB-I00 and PID2022-136435NB-I00, both grants funded by MCIN/AEI/10.13039/501100011033, PID2022 also funded by "ERDF A way of making Europe", EU; 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).

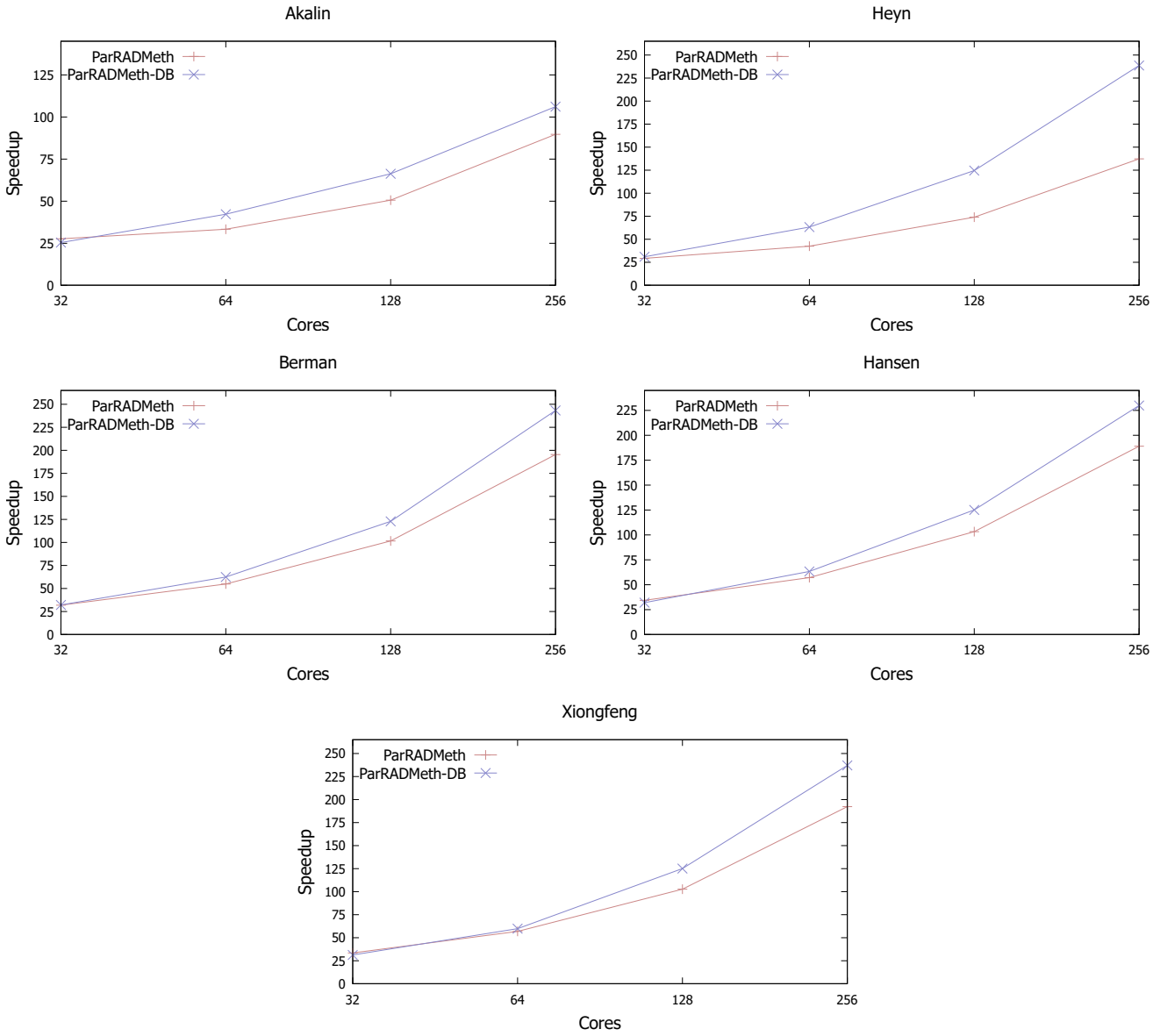

Figure 3: Speedup of *ParRADMeth-DB* and *ParRADMeth* over *RADMeth* for a varying number of nodes (32 cores per node)

REFERENCES

[1] Altuna Akalin, Francine E Garrett-Bakelman, Matthias Kormaksson, Jennifer Busuttil, Lu Zhang, Irina Khrebtukova, Thomas A Milne, Yongsheng Huang, Debabrata Biswas, Jay L Hess, et al. 2012. Base-pair resolution DNA methylation sequencing reveals profoundly divergent epigenetic landscapes in acute myeloid leukemia. *PLoS genetics* 8, 6 (2012), e1002781.

[2] Matthew Baker, Aaron Welch, and Manjunath Gorentla Venkata. 2015. Parallelizing the Smith-Waterman algorithm using OpenSHMEM and MPI-3 one-sided interfaces. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies: Second Workshop, OpenSHMEM 2015, Annapolis, MD, USA, August 4-6, 2015. Revised Selected Papers 1*. Springer, 178–191.

[3] Benjamin P Berman, Daniel J Weisenberger, Joseph F Aman, Toshinori Hinoue, Zachary Ramjan, Yaping Liu, Houtan Noushmehr, Christopher PE Lange, Cornelis M van Dijk, Rob AEM Tollenaar, et al. 2012. Regions of focal DNA hypermethylation and long-range hypomethylation in colorectal cancer coincide with nuclear lamina–associated domains. *Nature genetics* 44, 1 (2012), 40–46.

[4] Yangyang Cao, Haoqing Yang, Luyuan Jin, Juan Du, and Zhipeng Fan. 2018. Genome-wide DNA methylation analysis during osteogenic differentiation of human bone marrow mesenchymal stem cells. *Stem cells international* 2018 (2018).

[5] Xiongfeng Chen, Qinghua Lin, Junping Wen, Wei Lin, Jixing Liang, Huibin Huang, Liantao Li, Jianxin Huang, Falin Chen, Deli Liu, et al. 2020. Whole genome bisulfite sequencing of human spermatozoa reveals differentially methylated patterns from type 2 diabetic patients. *Journal of diabetes investigation* 11, 4 (2020), 856–864.

[6] Barbara Collignon, Roland Schulz, Jeremy C Smith, and Jerome Baudry. 2011. Task-parallel message passing interface implementation of Autodock4 for docking of very large databases of compounds using high-performance super-computers. *Journal of computational Chemistry* 32, 6 (2011), 1202–1209.

[7] Emerson de Araujo Macedo, Alba Cristina Magalhaes Alves de Melo, Gerson Henrique Pfitscher, and Azzedine Boukerche. 2011. Hybrid MPI/OpenMP strategy for biological multiple sequence alignment with DIALIGN-TX in heterogeneous multicore clusters. In *2011 IEEE International Symposium on Parallel and Distributed*

Processing Workshops and Phd Forum. IEEE, 418–425.

[8] Alejandro Fernández-Fraga, Jorge González-Domínguez, and Juan Tourino. 2022. ParRADMeth: Identification of Differentially Methylated Regions on Multicore Clusters. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (2022).

[9] Jorge Gonzalez-Dominguez and Maria J Martin. 2017. MPIGeneNet: parallel calculation of gene co-expression networks on multicore clusters. *IEEE/ACM transactions on computational biology and bioinformatics* 15, 5 (2017), 1732–1737.

[10] Chenggong Han, Hancong Tang, Shuyuan Lou, Yuan Gao, Min Ho Cho, and Shili Lin. 2018. Evaluation of recent statistical methods for detecting differential methylation using BS-seq data. *OBM Genetics* 2, 4 (2018), 1–36.

[11] Kasper D Hansen, Sarven Sabunciyan, Ben Langmead, Noemi Nagy, Rebecca Curley, Georg Klein, Eva Klein, Daniel Salamon, and Andrew P Feinberg. 2014. Large-scale hypomethylated blocks associated with Epstein-Barr virus–induced B-cell immortalization. *Genome research* 24, 2 (2014), 177–184.

[12] Holger Heyn, Ning Li, Humberto J Ferreira, Sebastian Moran, David G Pisano, Antonio Gomez, Javier Diez, Jose V Sanchez-Mut, Fernando Setien, F Javier Carmona, et al. 2012. Distinct DNA methylomes of newborns and centenarians. *Proceedings of the National Academy of Sciences* 109, 26 (2012), 10522–10527.

[13] Iksoo Huh, Xin Wu, Taesung Park, and Soojin V Yi. 2019. Detecting differential DNA methylation from sequencing of bisulfite converted DNA of diverse species. *Briefings in bioinformatics* 20, 1 (2019), 33–46.

[14] Hans-Ulrich Klein and Katja Hebestreit. 2016. An evaluation of methods to test predefined genomic regions for differential methylation in bisulfite sequencing data. *Briefings in bioinformatics* 17, 5 (2016), 796–807.

[15] Mikko Konki, Maia Malonzo, Ida K Karlsson, Noora Lindgren, Bishwa Ghimire, Johannes Smolander, Noora M Scheinin, Miina Ollikainen, Asta Laiho, Laura L Elo, et al. 2019. Peripheral blood DNA methylation differences in twin pairs discordant for Alzheimer's disease. *Clinical epigenetics* 11 (2019), 1–12.

[16] Mingzhe Li, Xiaoyi Lu, Khaled Hamidouche, Jie Zhang, and Dhabaleswar K Panda. 2016. Mizan-rma: Accelerating mizan graph processing framework with mpi rma. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 42–51.

[17] Glenn R Luecke, Nathan T Weeks, Brandon M Groth, Marina Kraeva, Li Ma, Luke M Kramer, James E Koltes, and James M Reecy. 2015. Fast epistasis detection in large-scale GWAS for Intel Xeon Phi clusters. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 3. IEEE, 228–235.

[18] Maksim V Shegay, Dmitry A Suplatov, Nina N Popova, Vytas K Švedas, and Vladimir V Voevodin. 2019. parMATT: parallel multiple alignment of protein 3D-structures with translations and twists for distributed-memory systems. *Bioinformatics* 35, 21 (2019), 4456–4458.

[19] Qiang Song, Benjamin Decato, Elizabeth E Hong, Meng Zhou, Fang Fang, Jianghan Qu, Tyler Garvin, Michael Kessler, Jun Zhou, and Andrew D Smith. 2013. A reference methylome database and analysis pipeline to facilitate integrative and comparative epigenomics. *PloS one* 8, 12 (2013), e81148.

[20] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. 2015. Big data: astronomical or genomical? *PLoS biology* 13, 7 (2015), e1002195.

[21] Meng Wang and Lei Kong. 2019. pblat: a multithread blat algorithm speeding up aligning sequences to genomes. *BMC bioinformatics* 20, 1 (2019), 1–4.