Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DE COMPUTADORES

pRIblast: a high efficient, parallel application for RNA-RNA interaction prediction

Estudante: Iñaki Amatria Barral
Dirección: Jorge González Domínguez
Juan Touriño Domínguez

A Coruña, setembro de 2021.

"What if changing the world was just about being here, by showing up no matter how many times we get told we don't belong, by staying true even when we're shamed into being false, by believing in ourselves even when we're told we're too different? And if we all held on to that, if we refuse to budge and fall in line, if we stood our ground for long enough, just maybe...
The world can't help but change around us."

Elliot Alderson (Mr. Robot)

Acknowledgements

Mamá, Papá, Aritz, Anita and Xabier. Abuela, Abuelo, Amatxi and Aitatxi. Belén, Deyse, Nerea, Edu and Phil. Ester, Santi, Fran, Anita, Miriam, Yago, Juancho, Violeta, Aitana, Jorge, Félix and Celia. Juan and Félix. Dani, Jaime, John, Cristina, Rubén and Manu. Desirée, Santi, Emma and Juan. Nacho, Paula and Miguel. Iván, Jorge, Alonso, Javi, Xabier and Pedro. Óscar, Javier, Cristina, Fran, Emilio, Jorge and Juan.

Abstract

For a long time, it was a common and well-established belief that RNA's only role was to intermediate between DNA and protein. However, during the last three decades, this long-held belief has been completely shattered. With the development of next generation sequencing technologies, it has been found out that most RNA in the human genome does not translate into protein. This is the so called long noncoding RNA (lncRNA), whose discovery has drastically changed the way biologists approach genetics. Furthermore, studies show that, besides playing important roles in many biological processes, the dysfunction of many lncRNA sequences are associated with serious diseases, such as cancer or diabetes.

Consequently, noncoding RNA biology is a hot research topic, and biologists are constantly trying to come up with new strategies to elucidate lncRNA functions, some of which include computational prediction of interacting RNA and lncRNA pairs (lncRNA works by being assembled with other proteins or RNA). For this very purpose, many application-specific bioinformatics tools have been developed. For instance: RIsearch2, ASSA and RIblast, which is one of the fastest, yet accurate, tools in the market right now. However, even though it is up to 64 times faster than other predictors, RIblast still falls very short when it is supplied with huge and significant lncRNA datasets, and, therefore, further progress in the field is still very limited.

To address this particular problem, this thesis presents pRIblast: a high efficient, parallel application for extensive and comprehensive RNA-RNA interaction analysis. Programmed with industry standard parallel technologies (MPI and OpenMP), pRIblast introduces the RIblast algorithm into high performance computing facilities (i.e. clusters of multicore systems joined together by an interconnection network). Moreover, pRIblast has been optimized to reduce memory usage and input and output latencies to the bare minimum and ,therefore, the novel application is ready to take on new challenges that could never have been faced with the former RIblast tool (i.e. the human genome).

To ensure pRIblast fulfills all quality criteria to be considered production ready, this thesis presents comprehensive benchmarking done on a 16-node computer cluster too (64GiB of main memory and 16 CPU cores per node, which amount for a total of 256 CPU cores). The results are outstanding. They not only point out that the parallelization of RIblast is successful (101 days worth of work were reduced to just 21 hours), but they also assert the importance of the optimizations applied to the tool (it was possible to analyze two datasets which exceed

RIblast memory requirements, and I/O times were reduced from 4000 to just 90 seconds with a dataset that produced 407GiB of output data).

Resumo

Durante moitos anos pensouse que o ARN era un simple intermediario entre o ADN e as proteinas, mais, porén, a aparición de tecnoloxías de secuenciación de nova xeración permitiu descubrir que a maior parte do xenoma humano está formado por cadeas longas de ARN non codificante (lncRNA, polas súas siglas en inglés). É dicir, un tipo de ARN que non sintetiza proteínas. Ademais, estudos recentes demostraron que a disfunción dunha gran parte destas cadeas de ARN están relacionadas con enfermidades tan graves coma o cancro ou a diabetes.

Para dilucidar a función das lncARNs, xurdiron numerosas ferramentas informáticas que tratan de predicir interacións ARN e lncRNA, xa que, as últimas, funcionan ensamblándose xunto a outras proteínas ou cadeas de ARN. Algunhas destas ferramentas son: RIsearch2, ASSA e, máis notablemente, RIblast, que obtén resultados até 64 veces máis rápido que outras aplicacións dispoñibles no mercado sen comprometer a calidade das predicións. Malia isto, RIblast aínda é demasiado lenta e non pode traballar con conxuntos de lncRNAs moi grandes sen que os tempos de predición medren exponencialmente.

Neste Traballo Fin de Grao desenvolveuse pRIblast, que é unha mellora sobre o algoritmo RIblast que permite executalo en contornas de computación de altas prestacións. Para isto, utilizáronse tecnoloxías de programación paralela estándar (MPI e OpenMP) que fan que pRIblast poida explotar, eficientemente, calquera sistema de computación multinó con nós multinúcleo. A nova ferramenta tamén se optimizou para minimizar a latencia das operacións de entrada e saída e o uso de memoria. Así pois conseguiuse tanto reducir o tempo de cómputo do algoritmo RIblast en varias ordes de magnitude como posibilitar a execución de conxuntos de datos de gran tamaño que a ferramenta orixinal endexamais podería analizar (i.e. o xenoma humano).

Para asegurar que a paralelización da ferramenta foi efectiva, fixéronse longas e extensivas probas de rendemento nun clúster con 16 nós de cómputo, con 64GiB de memoria e 16 núcleos por nó (256 núcleos en total). Os resultados obtidos foron moi satisfactorios, xa que se acadaron grandes aceleracións que permitiron executar un gran xenoma, que tardaría 101 días en procesar, en tan só 21 horas. A maiores, demostrouse que as optimizacións desenvolvidas sobre o algoritmo paralelo son moi efectivas. Por exemplo, reducíronse os tempos de escritura dende 4000 a 90 segundos nun conxunto de datos que produce 407GiB de resultados, e se puideron analizar dous *datasets* que non poderían ser procesados polo algoritmo orixinal

debido ao seu uso intensivo de memoria.

Keywords:

- lncRNAs
- Bioinformatics
- MPI
- OpenMP
- High Performance Computing
- Parallel computing
- Big data

Palabras chave:

- lncRNAs
- Bioinformática
- MPI
- OpenMP
- Comput. de altas prestaciones
- Computación paralela
- Big data

Contents

List of Figures

List of Tables

Chapter 1
Introduction

This first chapter briefly presents the background research that has motivated the development of the novel pRIblast application. Also, it sets the goals pursued by this thesis and outlines the structure of the paper.

1.1 Motivation

DNA is a molecule which contains genetic code, the blueprint of life. This essential element is a long, double-stranded molecule made up of bases, and the order of these bases determines the genetic blueprint. It is similar to the way the letters in the alphabet are used to form words and communicate. DNA's "words" are three letters long, and they code for genes, which are the blueprints for proteins to be manufactured. To "read" these blueprints, the double-helical DNA is unzipped to expose the individual strands as a mobile, intermediate message called RNA.

So, essentially, RNA's function is to synthetize proteins. And for a long time, it was a common and well-established belief: RNA's only role is to intermediate between DNA and protein. However, during the last three decades, this long-held belief has been shattered. With the development of next generation sequencing technologies, scientists have witnessed amazing discoveries with regards to RNA biology. For instance, there exists a special type of double-stranded RNA which can turn off specific genes based on certain sequences (RNAi). Furthermore, it has been found out that most RNA in the human genome does not translate into protein. This is the so called long noncoding RNA (lncRNA), whose discovery has drastically changed the way biologists approach genetics: how do we understand noncoding RNA if it does not synthetize proteins? Moreover, studies show that it plays important roles in several mission-critical biological processes, and its dysfunction is associated with serious diseases, such as childhood developmental disorders [1] or the SARS-CoV-2 [2].

As a consequence, noncoding RNA biology is a hot research topic, and biologists are, as

of today, still trying to come up with new strategies to elucidate noncoding RNA functions, some of which include computational prediction of interaction of lncRNAs. Yet, because these algorithms have very demanding memory requirements and computation times that exceed what is feasible, further progress in the field is still very limited.

1.2 Objective

The main objective of this thesis is to develop pRIblast: a high efficient, parallel algorithm for RNA-RNA interaction prediction. Indeed, this novel tool will allow to predict RNA-RNA interactions using all hardware available in state-of-the-art supercomputing facilities and multinode computing clusters. As a consequence, the execution times of such experiments will be reduced by orders of magnitude. Furthermore, it will even allow to run the algorithm over huge datasets that would have never been possible to process before. For this purpose, the former RIblast algorithm will be progressively refined to add both explicit and implicit parallelism support, using MPI functions and OpenMP directives respectively.

Moreover, comprehensive benchmarking and testing will be conducted to draw extensive conclusion. Once the pRIblast algorithm has been developed and tested to be production ready, it will be evaluated against a set of resource-intensive, real datasets. This will allow to understand the accelerations achieved by the new application and assess if any other improvements could be developed to further enhance pRIblast's performance.

As a side objective, this project also aims to introduce the student to the development processes and workflows followed in research today. Finally, all the code developed within this thesis will be uploaded to GitHub (under the MIT license) so that it can be downloaded, compiled and run by scientists all around the globe.

1.3 Outline

Chapter 2 begins with a brief description of what lncRNAs are and why their function estimation is an important research topic. Subsequently, the FASTA file format, which is used to represent this type of sequences, and the RIblast algorithm are presented. Finally, the computer architecture targeted by pRIblast is described, alongside with two parallel programming technologies (MPI and OpenMP) used to develop the novel algorithm.

Chapter 3 presents the implementation of the high efficient, parallel algorithm pRIblast. For that purpose, it describes the three distinct workload distribution procedures developed within the tool. Moreover, it introduces the two optimizations over the novel application that allow it to process large-scale datasets.

Moving on, Chapter 4 analyzes and discusses pRIblast's benchmarking results against

five real and representative lncRNA datasets. Also, it describes the computing cluster where tests were conducted and the methodology followed to assess the capabilities of the new bioinformatics tool. Lastly, early conclusions are presented.

Chapter 5 provides information regarding the development methodology, planning and management of the work conducted in this thesis; and Chapter 6 draws both final conclusions and future lines of work.

At last, Appendix A provides the reader with a user guide, which may help compiling, executing and configuring pRIblast.

Chapter 2

Background

Throughout this chapter, a basic understanding of what lncRNAs are, how they are represented in a computer and why their function estimation is a hot research topic will be settled. Also, a state-of-the-art RNA-RNA interaction prediction system (namely RIblast) and the main obstacle these programs face when characterizing lncRNAs will be presented. Finally, the computer architecture targeted by the new high performant pRIblast algorithm will be described, alongside with two parallel programming technologies (MPI and OpenMP) that were used to develop the novel application.

2.1 lncRNAs

lncRNAs are a type of RNA defined as being transcripts with lengths exceeding 200 nucleotides that are not translated into protein. More than 58000 lncRNA genes are encoded by the human genome, but most of them are still poorly characterized. Moreover, lncRNAs play integral roles in various biological processes, and the dysfunctions of many lncRNAs are associated with severe diseases, such as diabetes and various types of cancer [3]. Consequently, elucidating lncRNA purposes is an important research topic.

As lncRNAs work by being assembled with other proteins or RNAs, the identification of interaction partners for each lncRNA is a powerful approach for inferring their function. However, *in vivo* experimental detections of comprehensive lncRNA-RNA interactions are difficult. Hence, fast and reliable computational prediction of interacting lncRNA-RNA pairs is an indispensable technique to further progress in lncRNA function estimation.

Several sequence-based technologies have been developed for the experimental discovery of RNA-RNA interactions. ASSA [4] and RIsearch2 [5] are examples of some of the more advanced and up-to-date RNA-RNA predictors (and paper [6] benchmarks and reviews many more). Nevertheless, as prediction models become more sophisticated (and thus results more accurate), the computational cost associated to RNA-RNA interaction prediction grows expo-

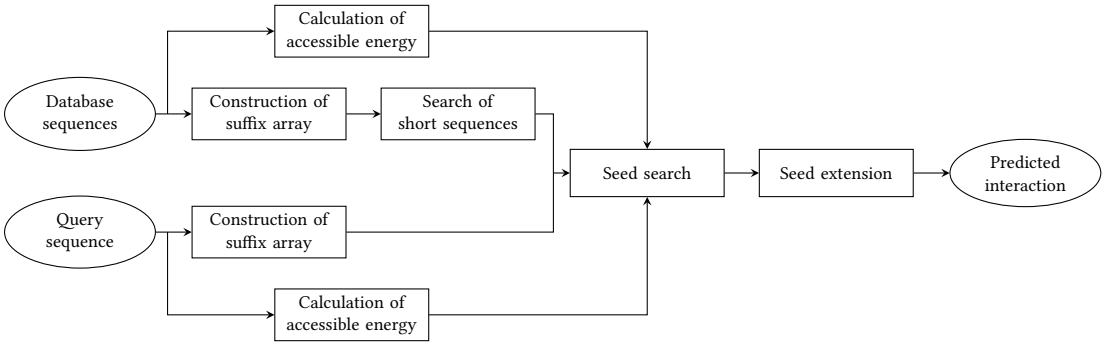

Figure 2.1: Overview of the RIblast algorithm

nentially and prevents their application to large-scale datasets (i.e. the human genome).

2.2 RIblast

RIblast [7], developed by T. Fukunaga and M. Hamada at Waseda University (Tokyo), is an RNA-RNA interaction prediction algorithm for comprehensive lncRNA interaction analysis. The key idea of the algorithm is the utilization of a seed-and-extension heuristic, which is widely adopted in sequence homology search tools. Furthermore, it also uses suffix arrays[1], Single Instruction Multiple Data (SIMD) directives and an approximate RNA energy model to predict lncRNA interactions at unrivalled speeds.

As shown in Figure 2.1, RIblast implements two major steps: database construction and RNA interaction search. In this sense, different batches of input queries can be run against the same target RNA dataset (database) with no need to construct it again. Therefore, the focus of this thesis is the parallelization of the second step, which not only is the most often executed, but the most computationally expensive too.

In the database construction step, RIblast first calculates approximate accessible energies of each sequence in the target RNA dataset. Accessible energies are later used in the RNA interaction search step to find promising seed regions. Second, target RNA segments are reversed and concatenated with delimiter symbols to build a suffix array. Third, search results of short strings are exhaustively pre-calculated in order to speed up the RNA interaction search. And finally, the approximate energies, concatenated sequences, suffix arrays and search results of short strings are stored in a database (i.e. a set of text and binary files).

In the RNA interaction search step, RIblast first calculates approximate accessible energies and constructs a suffix array for every query sequence. Second, RIblast finds seed regions with

[1] A suffix array is a space-efficient text-indexing data structure that comprises a table of the starting indexes of all suffixes of a string in alphabetical order. It can be constructed in linear time relative to the string length.

```
1  procedure RNAInteractionSearch()
2    db := loadDB()
3    seqs := loadSeqs()
4    for seq in seqs do
5      suff := constructSuffixArray(seq)
6      acc := calculateAccessibility(seq)
7      seeds := seedSearch(suff, acc, db)
8      hits := extendSeeds(seeds, suff, acc, db)
9      saveHits(hits)
10   end for
11 end procedure
```

Figure 2.2: The RNA interaction search step expressed in pseudocode

a particularly low interaction energy based on the suffix arrays and the accessibility energies of the query and the target RNAs. Third, RIblast extends interactions from seed regions and, finally, interactions that fully overlap are removed and results are stored in an output text file.

Figure 2.2 shows a more detailed overview of the interaction prediction algorithm. Finding regions with a particulary low interaction energy (Line 7) is the most computationally demanding step in lncRNA-RNA interaction prediction (computation time is quadratic with the number of sequences when all-to-all interaction prediction is conducted). That is why state-of-the-art RNA-RNA interaction prediction programs base its functioning on heuristics, such as the seed-and-extension approach. It is feasible to explore the solution space based on heuristic knowledge (i.e. using seeds) instead of exhaustively calculating all the possible options in a brute-force fashion.

Without a doubt, RIblast has proved to be an important step forward towards comprehensive lncRNA function estimation. Indeed, it has been key to gain new insight into understanding genetic interaction in schizophrenia [8] and to identify potential targets for treatment and prevention of bleaching in coral [9]. Moreover, benchmarking [10] confirms the authors' claims regarding the speed of the algorithm, and it positions the application within the top three best RNA-RNA interaction prediction tools (yet it also asserts that there is still room for improvement in prediction accuracy). However, RIblast is still extremely slow when it is used against large-scale datasets. As a consequence, T. Fukunaga and M. Hamada proposed the use of parallelization techniques as a next step in order to successfully speed up the calculation of interacting lncRNA-RNA pairs.

2.2.1 Input file format: the FASTA format

RIblast input files are codified using the FASTA file format. FASTA is a text-based format for representing nucleotide or protein sequences based on single-letter codes. The format also

```
>ENSLCOT00000000003.1 ncrna
GATGCTGCGGCGGGTTCTGGGGGGTCTCA
GGGTGTTTTTTGGGGTCTCAGGGTGGATT
>ENSLCOT00000000010.1 ncrna
GAGTATGTCGAGGGGCTAAAGGTGGATGG
GGTCACAGGTTGTGGGTGGGGGTAATTTG
```

Figure 2.3: Multiple sequence FASTA file with two reads

allows for sequence names and comments to precede the sequences. It originated from the FASTA software package [11], but has now become a near universal standard in the field of bioinformatics.

The first line in a FASTA file starts either with a ">" (greater-than) symbol or a, now deprecated, ";" (semicolon), which was taken as a comment. The line beginning with a ">" gives a name and/or a unique identifier for a sequence. However, it may also contain additional information, such as a unique library accession number. Following the initial line is the actual sequence itself in standard one-letter character string. Any other character is ignored (spaces, tabulators, line breaks, etc.). Among others, valid characters are: A, C, G, T, U and - (gap of indeterminate length).

Once the first sequence is defined, many more may come afterwards using the format described above. First, the sequence is identified using the character ">" and then described using the set of standard one-letter characters. In this sense, Figure 2.3 shows an example of a multiple sequence FASTA file with two RNA segments.

2.2.2 RNA interaction search output file format

RIblast outputs detected interactions using a non-standard file format. It represents interactions in plain text lines with five values each. The values that characterize a certain interaction are: an interaction identifier, a query sequence name, a target sequence name, the released interaction energy and the interacted regions (i.e. [region in query]:[region in target]).

The first three lines in an output file are reserved by the application to record execution data. For instance, user-defined RNA energy model parameters are saved into the file, and so are the names of the input FASTA file and target database used to execute the interaction search step.

After these lines are the interactions. Each interaction is stored in a new line, and it is defined by the parameters enumerated above, which are conveniently separated by commas. Figure 2.4 shows an example of a RIblast output file with two predicted interactions.

RIblast ris result
input:test_input.fa,database:test_db,MaximalSpan:100,...
Id,Query name,Target name,Interaction energy,Base pair
0,qrna,target_rna1,-10.218,(4-21:30-13)
1,qrna,target_rna2,-9.105,(72-83:185-170)

Figure 2.4: Example of a RIblast output file

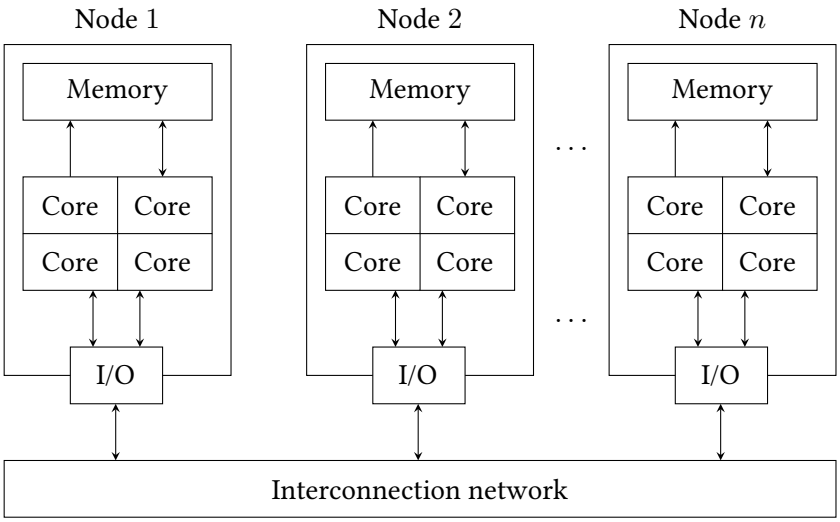

Figure 2.5: Cluster architecture targeted by the novel pRIblast algorithm

2.3 Target computer architecture

The target parallel architectures for this work are distributed-memory clusters that consist of several nodes (each of them with several CPU cores and memory modules) joined together by an interconnection network (see Figure 2.5). As a rather bold comparison, think of two or three desktops, each one with its very own memory, processor and Input and Output (I/O) interface, directly connected to a single switch. In this context, the Single Program Multiple Data (SPMD) programming paradigm is used to distribute workload among the collection of nodes, and therefore accelerate the computations of a given application. The maximum acceleration achieved by the cluster will then depend on how many nodes exist in the configuration, how many cores each node has and the specs of the hardware, such as the interconnection network and the RAM access interface.

Deepening into the SPMD programming model, and having identified the computer architecture behind distributed-memory clusters, it is now easy to undercover how to effectively exploit these sets of processing resources. First of all, as each node is an independent computer, workload can be distributed among them using mailboxes (i.e. sending messages over the interconnection network). That is, the programmer must define how data is divided into

smaller pieces, and sent over the network, to split the computing stress among the nodes of the cluster. This method of parallelization is the so called explicit parallelism. It is the programmer who is in charge of deciding which node stores a given piece of information, and how and when it is transferred over the network.

Moreover, it is possible to dive even deeper into the architecture and take advantage of yet another level of parallelism: the implicit parallelism. In this case, as every node has several CPU cores which share a common address space, threads may be used to further divide the workload. Consequently, the computing stress is split again among all the cores of every node in the cluster. This approach is known as implicit because no message passing is required. However, locking mechanisms are sometimes needed to avoid race conditions (i.e. when a software program depends on the timing of one or more processes/threads to function correctly) when data is read and written from main memory.

Finally, nodes with more sophisticated and specialized components, such as General Purpose Graphical Processing Units (GPGPUs), are, every day, more common in high performance data centers. Indeed, this type of resources may come in handy at the time of exploiting massively parallel, unconditional computations. However, GPGPUs are out of the scope of this work, and, thus, have not been used to speed up the RIblast algorithm. The main focus in this thesis has been the correct assignment of workload to nodes, using both implicit and explicit parallelism, to better utilize all the CPU cores available in supercomputing facilities.

2.4 MPI

Message Passing Interface (MPI) [12] is a high performance, scalable and portable communication protocol for programming parallel computers. Indeed, because hardware vendors provide a high level of support, MPI has become a *de facto* standard for communication among processes that model a parallel program running on a distributed-memory system (i.e. the explicit parallelism use case).

The MPI interface is meant to provide essential virtual topology, synchronization and communication functionality among a set of processes in a language-independent manner. More specifically, MPI functions include: point-to-point send/receive operations, collective functions involving a group of processes, organizing processes in Cartesian and graph-like logical topologies, one-sided communications and many more.

As of today, most high performance computer vendors offer their own MPI implementation. Nevertheless, open source and free alternatives, such as OpenMPI[2] and MPICH[3], mainly developed and maintained by academic institutions, are the most widely used implementations.

[2] https://www.open-mpi.org
[3] https://www.mpich.org

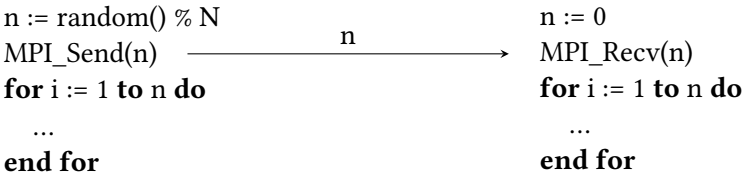

Figure 2.6: MPI_Send and MPI_Recv functions used together to coordinate a parallel computation

Considering the pRIblast algorithm, these are the MPI functions used within the development of the novel tool:

- **MPI_Init**, **MPI_Finalize** and **MPI_Abort**. The first two are used, respectively, to initialize and destroy the data structures used to communicate processes. MPI_Init must be called before any other MPI operation is invoked, and MPI_Finalize to free up resources as soon as the parallel section of code is finished. Finally, MPI_Abort allows for a coordinated shutdown of processes when an error event occurs.

- **MPI_Comm_rank** and **MPI_Comm_size**. These functions allow to obtain the numeric identifier of a running process and the number of processes working together in a parallel computation respectively. These two values are then used to specify the data and workload assigned to each process. For instance, they can be used to calculate offsets within an array to distribute it with other MPI operations.

- **MPI_Send** and **MPI_Recv**. These two operations are the building blocks of more sophisticated MPI operations that transfer data between a set of processes. Indeed, they are used together and block the execution of code until there exists a process sending information and another one receiving it. Figure 2.6 shows how these functions may be used side by side to coordinate a parallel computation.

- **MPI_Bcast**. It sends a message from a process to all the other $(n-1)$ processes in an MPI group. It is a collective operation, and, therefore, it must be invoked by every process involved in the parallel computation. Naively, one may think that this operation is implemented using a loop where the sending process calls $n-1$ times the *MPI_Send* function and the receiving processes the *MPI_Recv* operation. However, to better optimize the resource usage and runtime of the broadcast operation, those processes that have already received the message are used to send it again to other processes that have not yet received it. In this sense, a tree-like structure of send and receive operations is created, which distributes the message across all the processes in $O(\log n)$ time (see Figure 2.7).

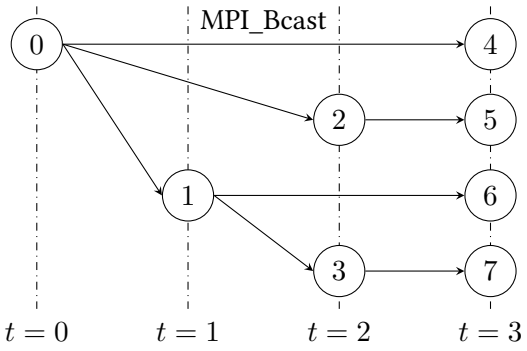

Figure 2.7: MPI_Bcast implementation to reduce the communication time from $O(n)$ down to $O(\log n)$

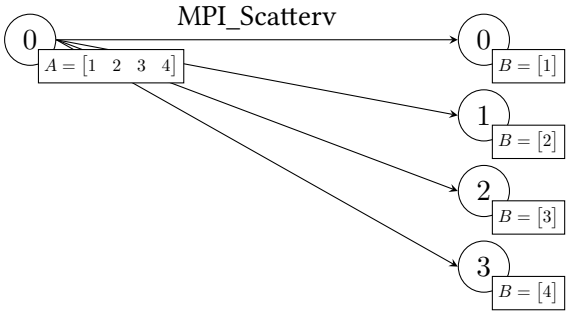

Figure 2.8: Example to illustrate how the MPI_Scatterv collective operation uniformly dispatches an array of integers among four MPI processes

- **MPI_Scatterv**. This function dispatches data from one process across all the other processes running the MPI section of code. The number of elements sent to each process does not need be to be the same for all of them. Similarly, it is possible to compute a displacement index for each process to have fine-grained control of the dispatching operation. Just like *MPI_Bcast*, *MPI_Scatterv* is also a collective operation. Figure 2.8 shows an example scenario where *MPI_Scatterv* has been used to evenly distribute an array of integers among four processes.

- **MPI_Fetch_and_op**. It is a one-sided type of communication, and it was included in the MPI specification as of version 2.0. *MPI_Fetch_and_op* allows one process to retrieve and modify another process' memory atomically, with no need for synchronization. It relies on Remote Direct Memory Access (RDMA) protocols to function. Two of the available operations are: *MPI_REPLACE*, which retrieves the piece of data stored in memory before substituting it for another value; and *MPI_SUM*, which reads the last value in the buffer before adding it a value passed in to the function as an argument.

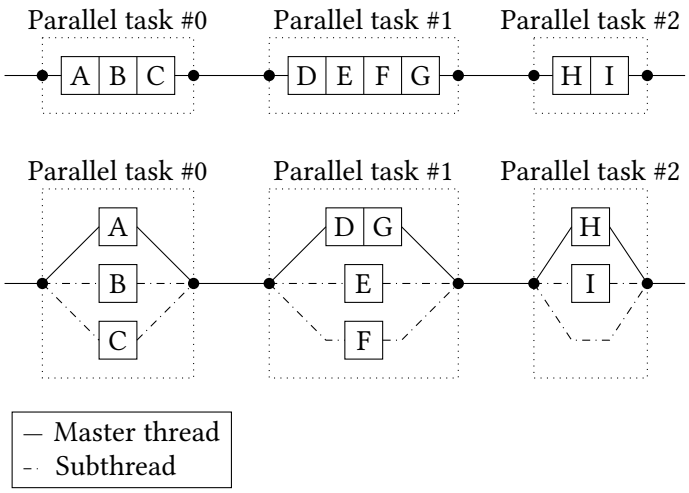

Figure 2.9: An illustration of the fork-join paradigm. Sequential execution is displayed on the top while its equivalent fork-join execution is on the bottom

2.5 OpenMP

MPI's future appears solid: the message passing paradigm remains the dominant model used in high performance computing today. However, computer architectures are constantly evolving. With greater internal concurrency (multicore), better concurrency control (thread affinity) and more levels of memory hierarchy, multithreading standards have been increasingly important in order to exploit fine-grained parallelism in shared-memory systems (i.e. implicit parallelism).

Open Multi-Processing (OpenMP) [13] is an application programming interface that supports multi-platform shared-memory multithreading programming. That is, OpenMP consists of a set of compiler directives, library routines and environment variables that influence runtime behaviour of shared-memory systems programs.

OpenMP implements the fork-join model (see Figure 2.9), a method of parallelization whereby a main thread forks a specified number of subthreads and the system divides a task among them. The threads run concurrently and, finally, join back into the main thread to resume the normal execution of the program. The section of code that is meant to run in parallel is marked accordingly with a compiler directive that will create the threads before the section is executed (see Figure 2.10). By default, each thread executes the parallel section of code independently, but work-sharing constructs can be used to divide a given loop among the threads (Line 2).

In addition, it is possible to determine how loop iterations are assigned to threads using scheduling clauses. The three main types of scheduling are:

```
1  procedure parallelForLoop(n: integer, a: *integer, b: *integer)
2    #pragma omp parallel for
3    for i := 2 to n do
4      b[i] := a[i - 1] + a[i]
5    end for
6  end procedure
```

Figure 2.10: An embarrassingly parallel procedure accelerated with an OpenMP annotation

- **Static**. This clause divides the loop iterations into fixed size chunks and distributes them to the threads in circular order. It is the default scheduler used by loop constructs and distributes the iterations at the beginning of the execution, without modifying this distribution all along the loop. This is appropriate when all iterations have the same computational cost. By default, the chunk size is equal to the number of iterations divided by the number of threads.

- **Dynamic**. Similarly to the previous one, OpenMP also divides the iterations into fixed size chunks (1 by default). However, instead of assigning these chunks to the threads in advance, each thread only executes one chunk of iterations at a time. Then, it requests another chunk until there are no more chunks available. There is no particular order in which the chunks are distributed to the threads. This type of scheduling is appropriate when loop iterations require different computational cost.

- **Guided**. The guided scheduling type is similar to the dynamic one. Again, OpenMP divides the iterations into chunks, each thread only executes one chunk of iterations, and, then, it requests for another chunk until there are no more chunks available. The difference with the dynamic scheduling type is in the size of the chunks, which is proportional to the number of unassigned iterations divided by the number of threads. In this sense, the size of the chunks decreases exponentially over time. This scheduling policy is appropriate when the iterations are not balanced towards the end of the computation.

Figure 2.11 shows the difference in execution time for the three scheduling schemes after executing a carefully crafted for loop construct. Both the static and the guided schedulings achieve the same execution time. However, in practice, the guided policy would be slower, because T_1 has to ask three times for iterations to process. In contrast, the static scheme distributes the iterations before executing the loop, and therefore no thread needs to synchronize in order to process more iterations.

Using a portable, scalable and yet simple interface, OpenMP allows programmers to easily develop parallel applications for platforms ranging from the standard desktop computer

```
a := {3, 3, 2, 2, 1, 1}
#pragma omp parallel for schedule(type)
for i := 1 to 6 do
   sleep(a[i])
end for
```

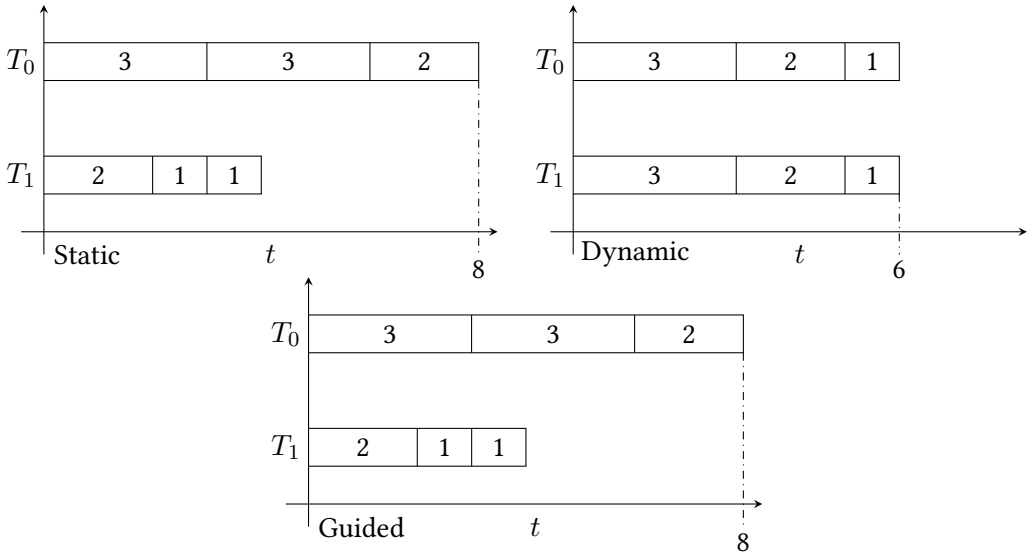

Figure 2.11: Difference in execution time for the static, dynamic and guided OpenMP scheduling schemes for an example with variable workload per loop iteration

to the supercomputer. Nevertheless, it is only valid for shared-memory systems, which are quite limited in terms of number of cores. Therefore, hybrid approaches, where OpenMP is used for intra-node parallelism (within a node) while MPI is used for inter-node parallelism (among nodes), are the most suitable paradigm in order to effectively exploit the capabilities of state-of-the-art supercomputing facilities (that is, a combination of both explicit and implicit parallelism).

Furthermore, hybrid approaches based on industry standard parallel programming technologies allow easy and manageable exploitation of the full set of underlying resources operating systems and modern processors may offer. For instance, the hyperthreading technology, which increases the number of independent instructions issued to the processor by up to 30%. One physical core appears as two to the operating system, allowing concurrent scheduling of two processes per core. In addition, two or more processes may be able to use the same resources (i.e. the I/O subsystem) without stalling the processor pipeline.

Chapter 3

Parallel implementation

Having characterized the RIblast algorithm, this chapter presents the high performant pRIblast application developed in this thesis. The novel tool takes advantage of the standard parallel programming technologies previously described (namely MPI and OpenMP) to efficiently exploit computer architectures similar to that of Figure 2.5. As its former predecessor, pRIblast is a command-line interface utility[1], and it is available for the scientific community to download at GitHub[2] under the MIT license.

The pRIblast interaction search step can be expressed in a high level pseudocode as in Figure 2.2. A quick analysis of the algorithm shows the inherent data parallelism the tool exploits. Indeed, the input batch of query sequences (Line 3) can be distributed among a set of working processes and loop iterations (Lines 4-10) executed concurrently.

However, there existed three main pitfalls in order to successfully parallelize the tool:

1. The correct assignment of workload to processes, which can seriously compromise the runtime of the application because sequences produce different amounts of seeds (more seeds mean longer computing time).

2. The memory usage, because the former algorithm was not designed to be run against large lncRNA datasets.

3. The I/O mechanism, which had to be modified in order to support fast reading and writing of files shared by both threads and processes over a network-based file system.

In this sense, the rest of this chapter focuses on presenting three different workload balancing schemes developed within pRIblast, a database paging mechanism that overcame the excessive memory usage of the RIblast algorithm and a suitable I/O procedure that limited the stress put in interconnection networks as datasets became larger.

[1] See the pRIblast user guide, Appendix A, for further information regarding the execution parameters and installation steps.

[2] https://github.com/amatria/pRIblast

3.1 Workload distribution

As it was discussed earlier, one of the main problems behind the development of pRIblast was the correct distribution of sequences between processes. This is important because, if it is not handled with special care, it may completely ruin the benefits of the parallelization. For instance, think of two web servers behind a load balancer. How would the performance degrade if one of those received the 80% of the HTTP requests and the other one the remaining 20%?

Even though it may not be obvious by examining Figure 2.2, the workload is, in fact, not evenly distributed among the sequences of the input sets. That is, sequences may produce different amounts of seeds, and thus successive steps to predict interactions may be computationally more expensive for those lncRNA transcripts considered more promising by the RIblast algorithm.

In order to overcome this extremely delicate situation, three different approaches to assign sequences to processing units were developed within the pRIblast algorithm. These are: a pure block scattering procedure, an area sum distribution and a dynamic decomposition scheme.

3.1.1 Pure block scattering

As a first step to parallelize the tool, a pure block data decomposition strategy was followed. That is, MPI functions were used to divide the input batch of sequences among a set of processes in chunks of size

$$S = \left\lceil \frac{\#\text{seqs}}{\#\text{procs}} \right\rceil,$$

and then each one spawned several OpenMP threads (one per physical core of the target architecture, as seen in Section 2.3) to compute its assigned sequences in parallel. Moreover, to minimize the impact of the variable workload, a dynamic scheduling policy (see Section 2.5) was applied in this second level of parallelism.

Because communicating sequences among processes is an expensive operation (remind that lncRNA transcripts are, indeed, long), interprocess synchronization was avoided in order to divide the input set of RNA segments. Hence, taking advantage of the lack of complexity of this decomposition scheme, the MPI ranks were used to calculate an offset within the input file and read the S sequences associated to each process. The offsets are computed as follows

$$O = S \cdot \#\text{rank}.$$

In those scenarios where RNA segments produce a similar amount of seeds (i.e. a similar amount of workload), the speedups achieved by this sequence scattering policy are outstanding. Nevertheless, experimental evaluation proved that, as lncRNA datasets become larger

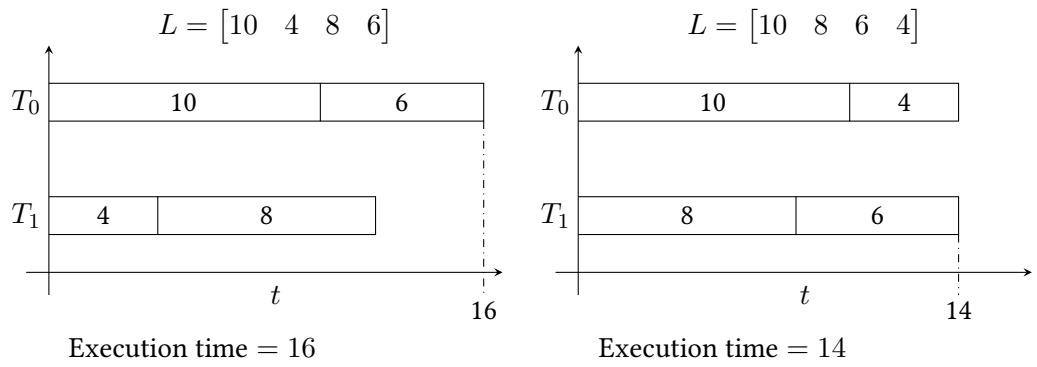

Figure 3.1: Example to prove that the sorting heuristic (right) produces optimal results if the workload is a direct function of the length of a sequence and a dynamic scheduling policy is applied

and sequences more diverse, the difference between the amount of seeds that each block produces rapidly compromises the effectiveness of this, rather naive, approach. Put differently, large-scale datasets contain sequences that differ a lot from one another. Therefore, it is reasonable to assume that the workload will not be equally distributed among the logical blocks of lncRNAs of the datasets. As a consequence, the tool will not scale properly, because it will end up in a situation similar to that of the 80-20 described before.

Did it mean that this decomposition scheme was all but obsolete? Not at all. A simple, yet powerful, heuristic was developed in order to optimize the per block execution time. This heuristic states as follows: sort the block of assigned sequences in descending order with respect to their lengths. Assuming the workload is a direct function of the length of a sequence, processing lncRNA transcripts in descending order according to their size leads to optimal execution time and resource usage in multithreading environments (but only when a dynamic thread scheduling policy is applied). Because mathematically proving this last statement is not within the scope of this thesis, a proof by example will be shown here.

Suppose that there exists a process with two threads, T_0 and T_1, that receives a block of sequences where the length of the sequences in the block is represented by the vector

$$L = \begin{bmatrix} 10 & 4 & 8 & 6 \end{bmatrix}.$$

Taking advantage of the assumption that the workload is a direct function of the sequences' lengths, execution time can be estimated as the number of characters of an RNA segment. Therefore, using vector L, it is possible to simulate the processing of the block, with and without making use of the sorting heuristic (as illustrated in Figure 3.1). After examining the example, the reader can check that the statement presented above holds, and the sorting approach produces optimal results indeed.

```
1  procedure blockRNAInteractionSearch()
2      db := loadDB()
3      seqs := blockDecomposition(rank, procs)
4      sort(seqs, reverse=True)
5      #pragma omp parallel for schedule(dynamic)
6      for seq in seqs do
7          suff := constructSuffixArray(seq)
8          acc := calculateAccessibility(seq)
9          seeds := seedSearch(suff, acc, db)
10         hits := extendSeeds(seeds, suff, acc, db)
11         saveLocalHits(hits)
12     end for
13     mergeHits()
14 end procedure
```

Figure 3.2: The RNA interaction search step accelerated by means of the block decomposition strategy

Note that, although in theory this heuristic works really well, the number of seeds a sequence produces does not only depend on its length. It is true that more characters may increase the chances of finding a seed, but there could be the case where a long sequence produces a small number of seeds. What is worse, a rather small sequence could end up producing a large amount of seeds. Thus, the sorting heuristic would underestimate the cost of computing that RNA segment (what will no longer lead to an optimal execution time). However, as those are very pathological cases and sequences tend to produce a number of seeds proportional to their lengths, the sorting heuristic was good enough to reduce the per block execution time.

To sum up, Figure 3.2 shows the complete pseudocode of the pure block decomposition strategy. First, every process loads into memory the target database (Line 2). Secondly, processes read from the input file their corresponding S sequences (Line 3). And lastly, the blocks of RNA segments are sorted in descending order with respect to their lengths (Line 4), and loop iterations run in parallel until there are no more lncRNAs to process (Lines 5-12). Moreover, it is important to point out that the final output file must be recomposed (Line 13) now that the input set of sequences has been divided among "#procs" different processes. The mechanism developed to gather these results will be explained in Subsection 3.2.2. For now, suppose that the final output file is constructed sequentially once all processes have exited the interaction search loop.

3.1.2 Area sum distribution

Early testing proved that the pure block decomposition was a rather simple, yet effective, data decomposition scheme. However, it soon fell short in scalability. Execution time optimization could only be made at per block level by means of the sorting heuristic, which strictly depends on the availability of threads. Therefore, a much more powerful scattering algorithm was needed in order to successfully speed up the prediction of RNA interacting segments.

Analyzing the pure block approach, it was easy to uncover the main problem behind it: it is so dead simple that the decomposition ends up being tightly coupled to the order in which sequences are defined in the input files. For instance, suppose again there exists a vector

$$L = \begin{bmatrix} 10 & 6 & 8 & 4 \end{bmatrix}$$

that represents the length of the sequences in a given input batch. With two processes available, and taking into account the assumption that the workload is a direct function of the sequences' lengths, the reader can quickly deduce that the best possible execution time would be, again, of 14 units of time. Indeed, the first process should receive sequences one and four and the second one sequences two and three. However, following the pure block decomposition policy, the first process would have the first and the second RNA segments assigned, while the second one would have the third and the fourth. In that case, the execution time would increase by two units of time and end up being 16.

Assuming again that the workload is proportional to the lengths of the sequences (i.e. the probability of finding seeds increases with the RNA segments' lengths), a new heuristic could be developed to optimally decompose the input batch of queries. Hence, not only optimizing the per block execution time, but the overall runtime too. This second heuristic can be summarized as trying to give to each process a number of lncRNA transcripts so that their lengths sum up to

$$S = \left\lfloor \frac{\sum_{i=1}^{N} \text{length(seqs[i])}}{\text{\#procs}} \right\rfloor .$$

In other words, this approach tries to provide a similar amount of characters to each process, even though it can mean that they receive a different amount of sequences.

Working on this idea, a more sophisticated data decomposition function was developed as shown in Figure 3.3. The new algorithm is an $O(np)$ procedure, where n is the number of sequences and p the number of processes, and it distributes sequences in a greedy-like manner (i.e. the algorithm makes locally optimal decisions). In this sense, for each process (Line 7), it goes through the list of available lncRNAs and tries to assign it as many sequences as possible without exceeding the value S defined earlier (Lines 11-19). It is also important to point out Line 6: the list of sequences is sorted in descending order according to their size before being

```
1  function areaSumDecomposition(rank: integer, procs: integer) : **char
2    decomp := emptyList()
3    if rank == 0 then
4      seqs := loadSeqs()
5      s := countChars(seqs) / procs
6      sort(seqs, reverse=True)
7      for i := 1 to procs do
8        j := 1
9        acu := 0
10       localBatch := emptyList()
11       while acu < s and j <= length(seqs) do
12         seq := seqs[j]
13         if acu + length(seq) <= s then
14           append(localBatch, seq)
15           remove(seqs, j)
16           acu := acu + length(seq)
17         end if
18         j := j + 1
19       end while
20       append(decomp, localBatch)
21     end for
22   end if
23   MPI_Scatterv(decomp)
24   return decomp
25 end function
```

Figure 3.3: Area sum data decomposition pseudocode

assigned to the processes. This is done to evenly distribute the big, medium and small RNA segments among all processes. Finally, following the MPI communication paradigm, the root process synchronizes with the rest of the workers in the MPI group and sends them their corresponding lncRNAs (Line 23) so that they can start the interaction prediction loop.

Note that the figure is just a high level description of the actual procedure. Indeed, the real implementation does not scatter sequences per se (Line 23). Recall that it is expensive to communicate long streams of characters over an interconnection network. In contrast, the actual function distributes arrays of indexes over the network, and each process reads its assigned sequences directly from the input file.

Looking back on the example presented earlier in the section, Figure 3.4 shows the difference in execution time between the pure block decomposition approach and the area sum distribution. The figure helps to better understand how the latter scattering policy works, and what exactly the S parameter represents here, which is no more than a "bucket size". In this way, the area sum distribution carefully fills up the buckets so that all of them have a

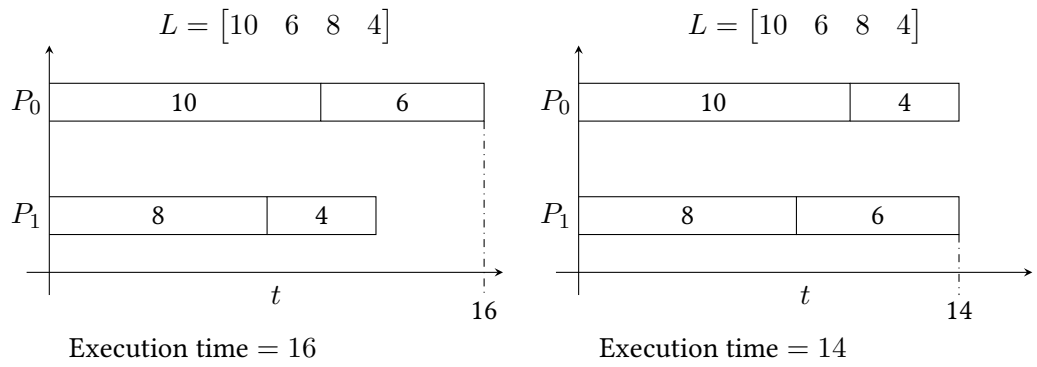

Figure 3.4: Comparison between the pure block decomposition scheme (left) and the area sum approach (right)

similar amount of workload. On the other hand, the block strategy does not evenly distribute characters among the processes at all, and it is for this very reason that it performs worse when supplied with diverse input files.

To conclude, note again that this decomposition is based on the assumption that the workload is directly proportional to the size of the sequences. Therefore, it is completely feasible to over or underestimate the cost of computing a sequence at the time of assigning it to a process (i.e. bucket). To minimize the impact on execution time of this scenario, the sorting heuristic and OpenMP's dynamic scheduling policy were used alongside with the area sum distribution. As a consequence, the final pseudocode of this interaction prediction procedure looks just like Figure 3.2. But this time, the *blockDecomposition* call (Line 3) is substituted by the *areaSumDecomposition* function studied in Figure 3.3.

3.1.3 Dynamic decomposition

The area sum approach proved to be a massive step forward towards successfully speeding up the pRIblast's interaction search procedure. Weighting sequences according to their size made it possible to better balance the workload among processing units, leading to significant boosts in performance. Still, there existed a case in which the area sum distribution fell as short as the naive block scattering scheme. What if there was a process that received a big amount of sequences considered unpromising and it turned out they produced the most workload?

As it was previously discussed, all the reasoning behind the heuristic knowledge used within the area sum strategy is based on the assumption that the workload is a direct function of the sequences' lengths. But other factors, such as the energy released by the interaction of two RNA segments, play a more important role at the time of finding seeds (and so considering a sequence more promising). Therefore, even though remote, the scenario presented above is completely feasible. And in this case, the workload would be so unbalanced that the running

time of the pRIblast algorithm would end up being close to its sequential counterpart.

In order to solve this problem, it was possible to follow two different approaches:

1. To use a more informed heuristic, which implies moving away from using the sequences' lengths as an indicator of workload.

2. To further minimize the time penalty associated with under or overestimating the cost of computing a sequence (recall that, at this point, the sorting heuristic does it at per process level).

Ideally, the first approach would have been the "goto". With a more informed heuristic, which would introduce domain-specific knowledge to better rank lncRNA transcripts, an algorithm based on the use of a heap could have been developed in order to optimally distribute the workload among a set of processing units. However, in order to develop a better heuristic, it was a must to have expert knowledge in the field of bioinformatics. Hence, it was the second technique the one followed in the development of this third decomposition scheme.

The idea behind minimizing time penalties is pretty simple: do not distribute sequences. Instead, use MPI one-sided communications to pull sequences from a shared pool of lncRNAs and collaboratively solve the problem. As a consequence, a situation as the one described at the beginning of the section would never become true. No process would ever have the responsibility of computing a concrete sequence.

Indeed, this last procedure increases the amount of communications required to run the algorithm. Whereas before there was only one point of synchronization (i.e. decomposing the sequences), now MPI-level mutual exclusion must be ensured in order to effectively make processes pull from a shared pool of RNA segments. Yet, providing that high performance interconnection networks, such as InfiniBand, are used for communication among processes, this overhead can be considered non-significant.

Figure 3.5 shows the pseudocode of the dynamic pRIblast algorithm. The first most noticeable difference is that data is no longer scattered among processes (Line 3). Instead, all processes load into memory the input set of sequences to avoid unnecessary and recurring access to disk. The price to be paid here was, clearly, memory space. But it is generally a lower price than reading over and over again the input file, seeking each time deeper and deeper to find the following sequence to process. Secondly, the pool of shared RNA segments is implemented using the *MPI_Fetch_and_op* operation (Line 7) that was presented in Section 2.4. That is, processes atomically pull and increment a shared index to obtain a reference to the next sequence that has not already been processed. And lastly, now the sorting heuristic (Line 4) is applied globally among all the processes working together to solve the problem. All threads run in parallel and ask the higher level in the hierarchy (MPI) for sequences to process (thus no concrete OpenMP scheduling is needed). However, this does not imply that threads

```
1  procedure dynamicRNAInteractionSearch()
2    db := loadDB()
3    seqs := loadSeqs()
4    sort(seqs, reverse=True)
5    #pragma omp parallel
6    while true do
7      idx := MPI_Fetch_and_op(&one, MPI_SUM)
8      if idx > length(seqs) then
9        break
10     end if
11     seq := seqs[idx]
12     suff := constructSuffixArray(seq)
13     acc := calculateAccessibility(seq)
14     seeds := seedSearch(suff, acc, db)
15     hits := extendSeeds(seeds, suff, acc, db)
16     saveLocalHits(hits)
17   end while
18   mergeHits()
19 end procedure
```

Figure 3.5: The RNA interaction search loop sped up using MPI one-sided communications

do not synchronize. Indeed, in order to call the *MPI_Fetch_and_op* function, shared-memory mutual exclusion is required.

Figure 3.6 better illustrates in a block diagram the dynamic procedure described above. Again, every process calls the *loadSeqs* procedure independently. It is the *MPI_Fetch_and_op* function that is responsible for assigning sequences to the processes. However, it does not show how every process spawns OpenMP threads to run in parallel the interaction search loop, but it is easy to imagine several arrows running the per process loop independently and joining back together before the *mergeHits* procedure is invoked.

3.2 Optimizations

With three distinct and robust data decomposition schemes, some preliminary tests were run to draw comprehensive conclusions about the assumptions made throughout the development of the different workload balancing strategies. Nevertheless, as the RIblast interaction prediction algorithm does not manage memory efficiently, some of the more resource-intensive experiments (and therefore more significant) abruptly crashed. This indicated that a memory-efficient scheme had to be developed in order to be able to process large-scale datasets. For instance, the human genome, which, ultimately, is one of the main goals behind computational prediction of lncRNA-RNA interacting pairs.

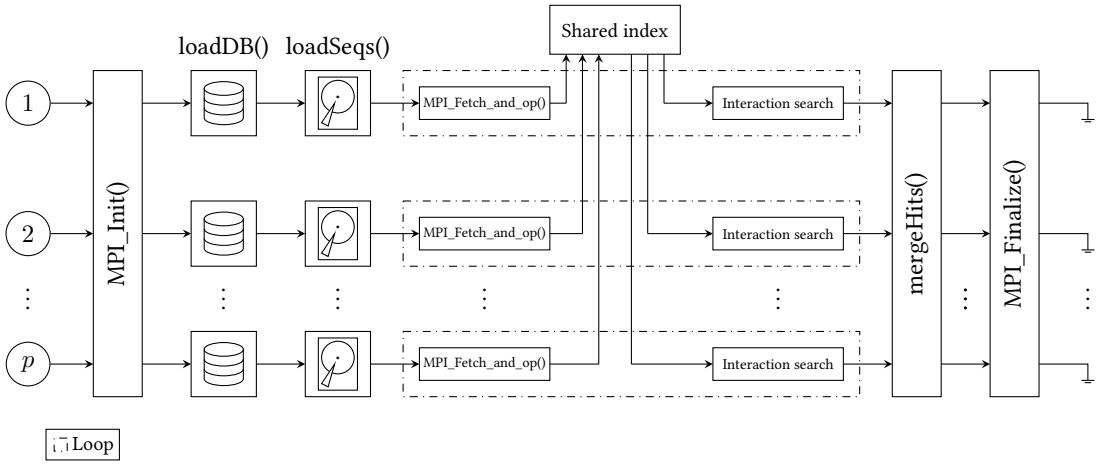

Figure 3.6: Overview of the pRIblast algorithm using the dynamic decomposition scheme

Moreover, after executing the parallel tool with increasingly larger input files, it became clear that a more sophisticated I/O procedure had to be developed in order to recompose the final output file. With large input datasets the number of predictions grew so fast that the time spent recording results became comparable to that of searching interactions, seriously compromising the speedup of the new parallel algorithm.

This section, therefore, presents two optimizations, namely a database paging mechanism and a parallel-aware I/O procedure, developed within the pRIblast application to be able to process large input datasets efficiently and effectively.

3.2.1 Database paging

Examining the simplified model of the pRIblast algorithm (see Figure 2.2), it can be quickly deduced why the former RIblast tool runs out of memory. Results are only written to the output file after all target RNA segments (i.e. the sequences in the database) have been compared against a query sequence. Thus, if a target database is big enough, or a query lncRNA produces a large amount of prediction candidates, the memory may be filled up before results are finally saved into disk.

Solving memory issues is usually tricky. It is a very delicate situation from which it is hard to recover the normal flow of execution of a program. pRIblast was no exception. And in addition, because it is a bioinformatics tool, it is hard to introduce changes in the algorithm without compromising the correctness of the results. Therefore, the straightforward solution to this problem was to divide the target database into smaller chunks (i.e. pages), which, because of their reduced size, would not overflow the available memory space. In other words, as results can only be written after a sequence has been compared against a target database,

```
1  procedure pagedRNAInteractionSearch()
2    dbs := loadDBChunks()
3    seqs := loadSeqs()
4    for seq in seqs do
5      suff := constructSuffixArray(seq)
6      acc := calculateAccessibility(seq)
7      for db in dbs do
8        seeds := seedSearch(suff, acc, db)
9        hits := extendSeeds(seeds, suff, acc, db)
10       saveHits(hits)
11     end for
12   end for
13 end procedure
```

Figure 3.7: The database paging mechanism expressed in pseudocode

now predictions will be made against subsets of such database. In this way, after a given sequence has been compared against all the subsets of the former target dataset, the output file will hold the same results as if the sequence had been compared against the entire database in one single round.

Even though the idea of dividing the target database in chunks was easy to conceptualize, problems arose once it was time to translate it into code. Because of the nature of the data structures stored in the database, it was not possible to divide it without no previous preprocessing. So, the database construction step was slightly modified to be able to read chunks of a fixed size N.

Figure 3.7 shows how the database paging mechanism changed the pRIblast interaction search step. Indeed, the database is now read in chunks (Line 2), and each query sequence is processed against the small and disjoint subsets individually (Lines 7-11). As a consequence, large input datasets (i.e. the human genome) are an easy go for the pRIblast algorithm. In addition, a very careful design process was followed so that databases created without the tweaked version of the construction step are completely compatible with the new procedure. Hence, end users can still use their target databases with the new parallel tool and only reconstruct them when memory outage issues force them to do so.

3.2.2 Parallel-aware I/O

Early testing showed that the time spent writing results to disk was non-significant when working with small-scale datasets. However, comprehensive testing proved that, as output files started to grow to the order of giga and terabytes, disk operations became comparable to those of the prediction loop. As a consequence, the maximum possible speedup achievable by the tool began to be increasingly limited. In a similar fashion, for a given dataset, the

I/O time became progressively important when the number of processing units dramatically reduced the time spent in the interaction prediction loop. In this case, the computing time became proportional to that of the disk operations, which, yet again, seriously compromised the efficiency of the algorithm. As a result, the last step to optimize the pRIblast algorithm in order to be able to process large lncRNA datasets was to develop a more suitable I/O procedure, decoupling the scalability of the tool to that of the disk operations.

Throughout this chapter, it has been assumed that the final results were only written to disk after all processes had finished their interaction search loop. That is, processes synchronize at the end of the search step, and, finally, copy their results into a single output file one by one. And indeed, it was the approach followed at the early steps of the development of pRIblast. However, it did not only break with parallelism (i.e. it forced processes to write their results sequentially), but it also produced increasingly slower writing times.

As an example, suppose there are two processes available, the area sum approach is used to decompose the lncRNA segments between them and vectors

$$L = \begin{bmatrix} 16 & 6 & 7 \end{bmatrix}$$

and

$$W = \begin{bmatrix} 4 & 1 & 2 \end{bmatrix}$$

represent the length of the sequences in a given input set and the time units it takes to write their predictions to disk respectively. Assuming that the probability of finding seeds is proportional to the length of a sequence, results can be presented as in Figure 3.8, which proves that this naive approach is not appropriate to write results at the time of processing large-scale datasets. It can be clearly seen that process P_1 could have written its results before P_0 did so. However, as no proper I/O procedure is applied and processes write their results to the output file in strict order after the interaction search loop has finished, P_1 is forced to wait seven units of (valuable) time before finally saving its results. Consequently, even though the prediction step was sped up successfully, the acceleration achieved by the tool was far from being optimal due to those three units of time that could have been run before P_0 had finished the prediction loop.

The takeaway here is that processes should copy their results into the main output file as soon as they have finished the interaction search loop. Put differently, as processes may receive a slightly different amount of workload (recall that the decomposition is based on an assumption), merging operations can be cleverly hidden in the pRIblast parallel execution pipeline while other processes finish their interaction prediction step. Nevertheless, mutual exclusion must exist to prevent data corruption if two or more processes try to copy their results at the same time.

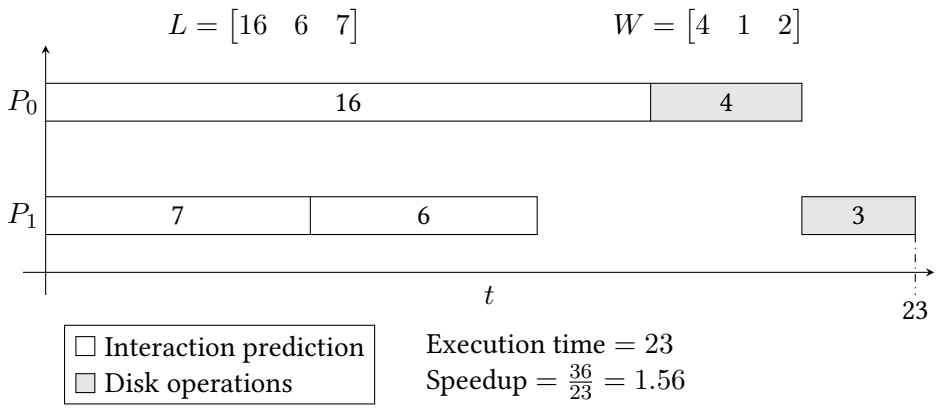

Figure 3.8: Sequential recomposition of the output file

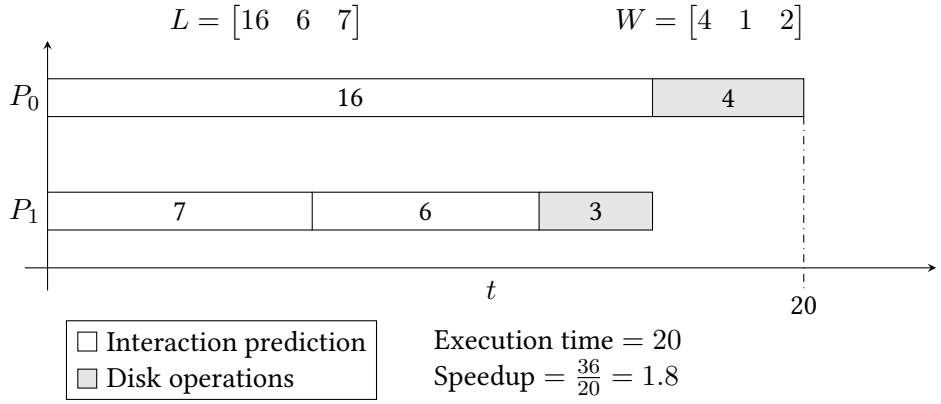

Figure 3.9: Parallel-aware recomposition of the output file

Working on this last idea, MPI one-sided communications were used to create a process-level lock and let processes merge their results individually into the final output file as soon as they finished the interaction search loop. In this way, under the same assumptions as those of the previous example, results can be drawn as in Figure 3.9, showing that the new parallel-aware procedure achieves both the best possible prediction and overall speedup.

Nonetheless, if processes receive an even amount of workload, this last technique quickly degenerates into a sequential writing of results to disk. In that case, a better I/O scheme should be designed (for instance, not merging results at all and writing the outputs into distinct prediction files). However, as such scenario is very unlikely (an optimal decomposition does not imply equal execution time), this approach is good enough for the time being. In fact, testing showed that writing times were reduced from about 4000 to just 90 seconds with a dataset that produced 436GiB of output data (using 256 CPU cores).

Figure 3.10 shows how the *mergeHits* procedure (Figure 3.5, Line 18; and Figure 3.2, Line 13) achieves process-level mutual exclusion by creating a message queue on top of MPI one-

```
1  procedure mergeHits(rank: integer, procs: integer)
2    next := 0
3    count := 0
4    last := MPI_Fetch_and_op(&rank, MPI_REPLACE)
5    if last != -1 then
6      MPI_Send(&rank, to=last)
7      MPI_Recv(&count, from=last)
8    end if
9    mergeResults()
10   count := count + 1
11   if count < procs then
12     MPI_Recv(&next, from=MPI_ANY_SOURCE)
13     MPI_Send(&count, to=next)
14   end if
15 end procedure
```

Figure 3.10: Parallel-aware I/O procedure expressed in pseudocode

sided communications, point-to-point send and receive operations and a shared index (as in the dynamic decomposition scheme of Subsection 3.1.3). Initially, the index holds the special value -1, meaning no other process has passed past that point in the code.

Digging into the details of the procedure, first, every process atomically pulls the current value in the shared index and replaces it with its own rank (Line 4). If the pulled value is equal to -1 (Line 5), it means that the process is the first one to finish the interaction search loop and it is free to write its results (Line 9). Otherwise, *last* holds the value of the last process waiting in the queue to acquire the lock and write to the output file. Therefore, it must wait until *last* has finished writing its results calling the *MPI_Send* operation (Line 6).

Once the first process has finished writing its results, it will increment the value of *count* (i.e. how many processes have already written their results) in Line 10. Then, it will proceed to execute Lines 11 to 14, to release the lock on the file and notify the next waiting process that it can save its results. In this way, it will first receive the rank of the waiting process (Line 12), because it has no way of knowing who is waiting for the lock, and then update its *count* value calling *MPI_Send* (Line 13). Similarly, the first process waiting in the queue executes Line 6, to send its rank to the releasing process; and Line 7, to receive the updated value of *count*.

All processes continue to execute this repeating pattern of send and receive operations to gradually copy their results one by one. However, the last one to finish the interaction prediction step will receive a value of *count* equal to the number of processes in the MPI group minus one (Line 7). Therefore, after saving its predictions and executing Line 10, *count* will be equal to the number of processes and it will skip Lines 12 to 13. Thus, closing the

locking cycle, which, ultimately, means all processes have saved their results.

Chapter 4
Evaluation

Now that the high performant pRIblast algorithm has been studied, the set of extensive tests run to benchmark the novel application will be presented. Moreover, to give more depth and context to the aforementioned tests, the architecture of the supercomputing environment used to evaluate the tool will be described too.

4.1 Environment

In order to gain comprehensive insight of both the efficiency and scalability of the pRIblast algorithm, the state-of-the-art CITIC's[1] cluster, namely "Plutón"[2], was used as a testing ground. "Plutón" is a heterogeneous high performance computing cluster that consists of 25 computing nodes, which, as a whole, sum up to 512 CPU cores, 2.8TiB of main memory, 19 NVIDIA Tesla GPUs and 3 many-core Intel Xeon Phi hardware accelerators.

"Plutón" has a so called frontend node dedicated exclusively to be the entry point to the cluster. End users remotely connect to this server to compile programs and setup working environments for the applications that will later run in the high performance nodes. Furthermore, it serves as an interface to a scheduling software used by individuals to book computing resources to run their jobs. And lastly, the frontend node works as a Network Attached Storage (NAS) too, allowing computing nodes to seemingly read and write to files stored in it.

Computing nodes are physically installed in three racks as follows:

- **Rack #0**. 17 computing nodes (compute-0-0 to compute-0-16) with a total of 272 CPU cores, 1088GiB of memory, 17 NVIDIA Tesla GPUs (Kepler) and 3 many-core Intel Xeon Phi hardware accelerators.
- **Rack #1**. 2 computing nodes (computer-1-0 to compute-1-1) with 48 CPU cores and 256GiB of RAM.

[1] https://www.citic.udc.es
[2] https://pluton.dec.udc.es

compute-0-{0-16}	
CPU model	2 × Intel Xeon E5-2660 Sandy Bridge-EP
Clock speed	2.2GHz/3.0GHz
#Cores per CPU	8
#Threads per core	2
#Cores/Threads per node	16/32
L1/L2/L3 Cache	32KiB/256KiB/20MiB
Main memory	64GiB DDR3 1600MHz
Hard drive	1 × HDD 1TiB SATA3 7.2Krpm
Network interfaces	InfiniBand FDR & Gigabit Ethernet

Table 4.1: Rack #0 hardware specifications

- **Rack #2**. 6 computing nodes (compute-2-0 to compute-2-5) for a total amount of 192 CPU cores, 1536GiB of main memory and 2 NVIDIA Tesla GPUs (Turing).

In order to assess the pRIblast's capabilities, only the nodes within the Rack #0 were used. It is the rack with the highest amount of CPU cores, and therefore it allows to better evaluate the scalability of the algorithm. Table 4.1 shows the specs of the hardware installed within the rack. Note that there are two processors in each node. Hence, it is possible to use up to 16 CPU cores per node (or 32 threads using Intel's hyperthreading technology).

From the table above, it is also important to point out the network interfaces. Each node is both attached to an InfiniBand FDR network and to a Gigabit Ethernet LAN. In this sense, the InfiniBand FDR technology is used to mount the NAS file system (served by the frontend node) and to communicate MPI processes that may run in the nodes, offering extremely low latencies (1-2μs) and very high bandwidth (56Gbps). In any other case, the Gigabit Ethernet interface is used; for instance, to open a remote shell from the frontend server to a computing node.

As several users may want to execute resource-intensive workloads at the same time, the cluster is managed by a scheduling software (Slurm Workload Manager[3]) running in the frontend node. The software is in charge of assigning hardware to the users depending on their computing needs. Indeed, applications may take days to finish their execution. Therefore, Slurm implements a fair use system, trying to execute as many workloads at a time as possible. Applications, or jobs, sent to the scheduler are defined by many parameters, some of which are:

- **Number of processes**. It sets the number of processes that will be spawned to run a given computation.

[3] https://slurm.schedmd.com

- **Number of processes per node**. It is the amount of processes that will be spawned in each computing node.
- **Number of nodes**. It is possible so set a raw value or let Slurm decide. In this case, the software assigns as many nodes as necessary using the "number of processes" and "processes per node" values.
- **Number of cores**. It sets the number of CPU cores assigned to each process.
- **Memory**. It is possible to set a total value or a value per CPU core. In any case, it may not exceed the total amount of memory installed in a node.
- **Timeout**. It is mandatory to set a timeout value. Therefore, if an application exceeds this value, it will be cancelled immediately. Also, the timeout value allows Slurm to make better scheduling decisions. To improve the quality of service in "Plutón", the timeout value may not exceed 72 hours.
- **Exclusive**. If set to true, the software will not schedule other tasks in the nodes used by the job. It is very useful at the time of benchmarking applications.

No command syntax has been described in this thesis. However, the interested reader may check the online documentation[4].

The frontend node also serves as a compilation and setup environment for the high performance workflows. In this sense, the Lmod[5] module system is used to provide different software versions to end users. Lmod automatically handles the PATH and other environment variables, minimizing the configuration hell that may be to port workflows from local computers to the cluster. The modules used within the compilation and execution of tests in this thesis were:

- **CC**. It loads a C/C++ compiler and the corresponding standard libraries. Specifically, the GNU Compiler Colection (GCC)[6] v8.3.0 with support for OpenMP directives.
- **MPI**. It loads the set of libraries and programs used to compile and run MPI programs. It was OpenMPI[7] v3.1.4 the one used here.

Other programs used, but already loaded into the system, were Git[8] and GNU Make[9].

[4] https://slurm.schedmd.com/documentation.html
[5] https://lmod.readthedocs.io/en/latest
[6] https://gcc.gnu.org
[7] https://www.open-mpi.org
[8] https://git-scm.com
[9] https://www.gnu.org/software/make

Dataset	lncRNAs	RNAs	Size	Sequential execution	Output size
Lepi	730	1256	896.0KiB	7602.21s	2.52GiB
Ursus	935	3899	1.9MiB	16714.60s	6.07GiB
Droso	2266	3963	3.6MiB	53029.00s	12.29GiB
Anser	10422	10988	12.0MiB	1930426.07s	406.26GiB
Anas	14504	15061	18.0MiB	8765575.33s	1.42TiB

Table 4.2: Datasets used to assess the performance of the pRIblast algorithm. All sequential execution times were computed using the former RIblast algorithm, except for the Anser and Anas datasets, which were computed using pRIblast

4.2 Datasets

Five different, real and representative lncRNA datasets were chosen (from the Ensembl [14] genome browser) to evaluate the performance of the pRIblast algorithm. These datasets[10], described in Table 4.2, range from small and non diverse to very large and completely unbalanced. Note that lncRNAs are RNAs too. In total, there are as many sequences per dataset as RNAs. Also, the sequential execution times for the Anser and Anas datasets were computed *a posteriori*, executing the pRIblast algorithm in several "Plutón" nodes and adding all their computing times together. Neither is it feasible nor a good practice to run tests that would last 22 and 101 days respectively (and moreover they would exceed the "Plutón" timeout value).

Indeed, as it was described in Subsection 2.2.1, these datasets are stored using the FASTA file format. However, in order to use them as inputs for the pRIblast algorithm, it is mandatory to first preprocess them. In this sense, two different FASTA files must be created: one with all the RNAs (which includes the lncRNA sequences), used for the database construction step; and another one with just the lncRNAs, which is then used as an input for the lncRNA-RNA interaction prediction search[11].

As for the datasets per se, the first two (Lepi and Ursus) are rather small and easy to compute. That is, sequences tend to follow the length heuristic statement (although some exceptions apply) and, therefore, workload can be easily balanced. By contrast, the other three (Droso, Anser and Anas) are progressively larger and more diverse. Therefore, these are going to show important differences at the time of evaluating the three data decomposition algorithms developed within pRIblast. Ideally, it would have been a very important milestone to compute the human genome too (the algorithm is completely capable of handling it). Nevertheless, it contains so many lncRNA sequences that it exceeds "Plutón" computing capabilities.

[10] All datasets are available to download at `ftp://ftp.ensembl.org/pub/release-97/fasta`.

[11] See the pRIblast user guide's Apendix A, which includes, in Subsection A.3.1, a Bash script that automatically creates these two input files from a given FASTA file.

4.3 Results

In this section, the results of the tests ran to assess the scalability of the pRIblast algorithm will be presented. As the novel tool can exploit both explicit and implicit parallelism, benchmarking also assessed which configuration of processes per node and threads per process better suits the "Plutón" architecture. Furthermore, each batch of tests evaluated whether using Intel's hyperthreading technology yields any improvement in execution time.

For each dataset (see Table 4.2), a table and a logarithmic bar plot will be shown, alongside with a small explanation of the results obtained and the best per node configuration of processes and threads. Also, all reasoning and discussion will be made using the following metrics:

- **Processing units**. It is equal to the number of physical resources used in a parallel computation. Note that it refers to the number of physical resources, not logical. Even though it is possible to spawn two threads within one CPU core (i.e. hyperthreading), the number of physical resources is still one. So, to be crystal clear, it is the number of CPU cores used to speed up a computation. Specifically, in the "Plutón" cluster, the number of processing units, p, is computed as

  $$p = 16 \cdot \#\text{nodes}.$$

  Therefore, if one node is used to execute the pRIblast algorithm, p will be equal to 16. Similarly, if two nodes are used to execute the parallel tool, p will be equal to 32, and so on and so forth.

- **Speedup**. It is the improvement of runtime for a task executed in parallel. It is calculated with respect to the number of processing units. The formula is as follows:

  $$\text{speedup}(p) = \frac{T_{\text{sequential}}}{T_{\text{parallel}}(p)},$$

  where $T_{\text{sequential}}$ is the execution time of the sequential program and $T_{\text{parallel}}(p)$ the execution time of the parallel program using p processing units. Ideally, it takes values between 0 and p, but the following cases may arise:

  - $\text{speedup}(p) \in (0, 1]$. The parallel program does not improve the execution time at all. Indeed, if the value is less than one, the parallel program performs worse than its sequential counterpart.
  - $\text{speedup}(p) \in (1, p]$. The parallel program improves the execution time. In the best case scenario, $\text{speedup}(p)$ equals p (linear speedup), which means the task has been perfectly divided among all the processing units.

  - speedup$(p) \in (p, \infty)$. The parallel program exceeds the theoretical speedup (i.e. superlinear speedup). This may happen if the parallel code improves the memory usage; for instance, reducing cache misses, and, therefore, the execution time of the program.

- **Efficiency**. It is the ratio of the speedup to the number of processing units. It takes values between 0 and 1. Efficiency equal to 1 means that the parallel algorithm makes perfect usage of all the available resources. However, it can take values above 1 when the speedup is also above the number of processing units (i.e. superlinear speedup). It is calculated as follows:

$$\text{efficiency}(p) = \frac{\text{speedup}(p)}{p}.$$

- **Scalability**. Ideally, the efficiency of a parallel algorithm stays constant as the number of processing units increases. However, in practice this is rarely true. As p grows larger, communication latencies, for instance, become increasingly important and limit the maximum speedup of a parallel algorithm. Moreover, a parallel algorithm may never achieve perfect scalability if workload cannot be balanced between processing units. Therefore, scalability represents whether efficiency stays constant when the number of processing units increases.

4.3.1 Methodology

Before presenting results, it is important to point out the methodology followed throughout the experimental evaluation phase of the pRIblast algorithm. It will help to better understand the figures and tables drawn in the following subsections.

First of all, to assess the scalability of the algorithm, each dataset was run using 1, 2, 4, 8 and 16 computing nodes. That is, using 16, 32, 64, 128 and 256 processing units (i.e. CPU cores). In this sense, it was possible to obtain educated conclusions about the three data decomposition schemes developed within the tool: do they really make a difference at the time of computing representative datasets? Moreover, the time spent computing every sequence of all datasets was also recorded. Therefore, it was possible to assess whether or not the decomposition algorithms achieved the best possible speedups by simulating perfect executions (i.e. knowing *a priori* how much time it takes to compute each sequence).

Secondly, the pRIblast application can exploit both explicit and implicit parallelism. Therefore, for each node, five different configurations of number of threads per process were tested. Indeed, those are: 1p$\times$16t (i.e. 1 process per node with 16 threads each), 2p$\times$8t (i.e. 2 processes per node with 8 threads each), 4p$\times$4t, 8p$\times$2t and 16p$\times$1t. This allowed to acquire knowledge

```
1  function bestSpeedup(seqsTimes: *integer, p: integer) : integer
2      times := arrayOfZeros(p)
3      sort(seqsTimes, reverse=True)
4      for seq in seqsTimes do
5          minIdx := findMinIdx(times)
6          times[minIdx] := times[minIdx] + seq
7      end for
8      return findMaxValue(times)
9  end function
```

Figure 4.1: Heap-like procedure to compute the best possible speedup for a given lncRNA dataset and number of processes p

regarding which arrangement of threads per process better suits those architectures similar to that of "Plutón".

Lastly, for each configuration of threads per process, it was tested whether using Intel's hyperthreading technology yields any improvement in execution time.

So, in conclusion, for each dataset, $5 \times 5 \times 2 \times 3 = 150$ tests were conducted. In total, 750 tests. But actually, only 630 of those were successful. The Anas dataset is so large that it exceeds the "Plutón" timeout value if it is executed in less than 16 computing nodes. Still, 630 results are way too many results. Therefore, in the following subsections, only the best results per node count and data decomposition scheme (including a theoretical upper bound) will be shown in the plots and tables. And the main focus on those figures will be the speedup (no hyperthreading used). The rest of the parameters mentioned earlier (the configuration of threads per process and hyperthreading) will be evaluated in plain text.

The theoretical upper bound was computed thanks to the fact that (as it was mentioned before) the time spent computing every sequence of all datasets was recorded in a database. Thus, it was possible to calculate what was the best speedup achievable using the function in Figure 4.1. For each dataset and number of processes p, it assigns sequences, in descending order according to their computing time, to each computing resource using a heap-like procedure (i.e. in each iteration it gives a new sequence to the process which has done less work). However, it is important to take into account that these values are just theoretical upper bounds, and, indeed, it is possible that a given algorithm performs better in practice (for instance, if it reduces cache misses or it better hides the parallel-aware I/O operations in the pRIblast parallel execution pipeline).

4.3.2 Lepi

The first batch of tests run to assess the scalability of the pRIblast algorithm was against the Lepi dataset. It is the smallest of the datasets and its sequences tend to follow the length

Decomp. algorithm	1N	2N	4N	8N	16N
Pure block	596.36s	343.67s	245.11s	237.33s	232.60s
Area sum	657.65s	420.29s	252.82s	242.77s	233.65s
Dynamic	521.37s	289.22s	246.55s	221.79s	217.80s
Theoretical	475.14s	237.56s	181.00s	181.00s	181.00s

Table 4.3: Parallel execution times of the Lepi dataset using the three pRIblast data decomposition algorithms

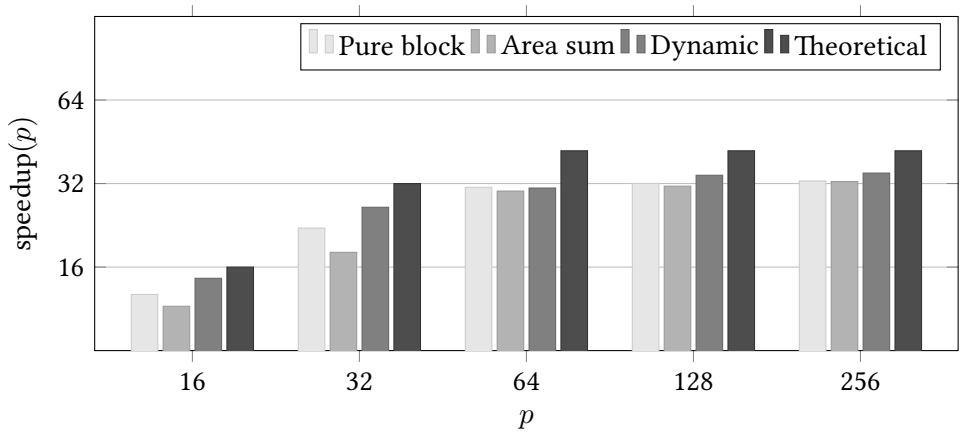

Figure 4.2: Speedups of the Lepi dataset using the three pRIblast data decomposition algorithms

heuristic statement (although some exceptions apply).

Examining Table 4.3 and Figure 4.2, it is possible to see that, even though the genome is fairly easy to compute, the three data decomposition algorithms achieve different execution times in the most representative tests. Those are the ones where 1 and 2 computing nodes are used, because it is possible to (theoretically) achieve a speedup$(p) = p$. Indeed, it is the dynamic algorithm the one that performs the best by far. The area sum algorithm overestimates the cost of computing sequences (this indicates that lcnRNAs with the same length do not exactly produce the same amount of work here) and the pure block procedure achieves somewhat better execution times, but simply out of pure luck. Remember that it is tightly coupled to the order in which sequences are defined in the input files.

As for the other tests (4, 8 and 16 computing nodes), they show that the scalability of the tool (in this particular scenario) is bounded by the execution time of one sole sequence. In fact, a sequence that takes 181.00s to compute. So, at the time of throwing in more resources, all decomposition schemes end up producing similar results. Anyway, the dynamic algorithm consistently proves to be faster than the other two procedures. However, there is still room for improvement. None of the three algorithms match the theoretical execution times for the dataset.

Lastly, hyperthreading yields better execution times only in those scenarios where the workload is not properly balanced among the computing resources (speedup(p) $\neq p$). And actually, they are in line with the theoretical values. Nevertheless, it slows down the runtime of the application at the time of executing the dataset with more than two computing nodes. Also, the static decomposition algorithms (pure block and area sum) are faster using more threads per process. The best execution times were achieved using 2p×8t and 4p×4t. By contrast, the dynamic scheme consistently gives better results using less threads per process (i.e. 8p×2t). This may be linked to the fact that the two static schemes are not so effective at the time of increasing the number of processes. That is, if there are more processes, the sequences are further divided. Therefore, there may appear processes which end up underutilized, because the workload has not been properly balanced between them.

4.3.3 Ursus

The Ursus dataset was the second one put to the test. As the previous one, it is a rather small dataset and workload is pretty much balanced among its sequences. However, the ratio of lncRNAs to RNAs is much smaller here (see Table 4.2). Therefore, the data decomposition algorithms are forced to make wise decisions in order to effectively balance the workload.

Examining Table 4.4 and Figure 4.3, it is possible to see that the dynamic scheme is the one that achieves the best results, followed by the pure block algorithm up to eight nodes. The area sum approach, once again, overestimates the cost of computing sequences. This shows that the area sum algorithm works best when the length heuristic (i.e. the probability of finding seeds is proportional to the length of the sequences) better matches the reality. However, if sequences are similar in length and produce slightly different workload (like both in this and the Lepi dataset), it does not work that well anymore. Indeed, a completely uninformed decomposition algorithm (i.e. the pure block) may even perform better.

Once again, throwing in more resources to a dataset whose scalability is bounded by the execution time of one single sequence only benefits those decomposition schemes making bad decisions. In fact, examining the results for 16 computing nodes, it is the only time where the area sum algorithm works just fine. Yet, the dynamic scheme still beats the other two procedures. Lastly, as with the Lepi dataset, there still exists room for improvement in prediction speed.

As for the hyperthreading and the configuration of threads per process, the same reasoning presented for the previous dataset applies here: hyperthreading yields better execution times if the workload is not evenly balanced among processing units. More threads per process works best for the pure block and area sum algorithms, because they are not able to correctly balance the workload when more processes are put into the mix.

Decomp. algorithm	1N	2N	4N	8N	16N
Pure block	1255.15s	644.95s	339.01s	179.81s	134.59s
Area sum	1291.76s	664.20s	366.35s	197.01s	122.41s
Dynamic	1173.68s	599.67s	310.66s	173.46s	119.29s
Theoretical	1044.66s	522.33s	261.16s	134.79s	134.79s

Table 4.4: Parallel execution times of the Ursus dataset using the three pRIblast data decomposition algorithms

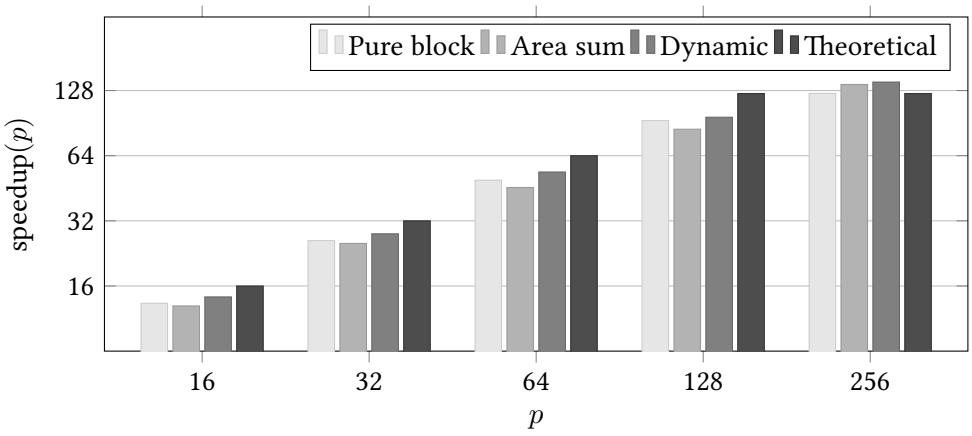

Figure 4.3: Speedups of the Ursus dataset using the three pRIblast data decomposition algorithms

4.3.4 Droso

The third set of tests was run against the Droso dataset. It is a rather large dataset and its sequences perfectly follow the length heuristic statement. As a consequence, the more advanced decomposition algorithms really make a difference here.

After carefully examining Table 4.5 and Figure 4.4, the reader can check that the area sum algorithm gives the best results this time. And moreover, those results slightly outperform the theoretical values sometimes. This comes from the fact that the length heuristic holds in this dataset, and the longest sequences produce the most workload here indeed. Following the area sum approach, it is the dynamic scheme, which performs well against this dataset too. However, the pure block procedure starts to fall short this time (see the differences in execution times), except when 16 computing nodes are used. In this last case, the presence of a significant amount of processing units, and the fact that the runtime is bounded by one only sequence that takes 636.20s to compute, clearly benefits this naive algorithm. There are so many CPU cores that, inevitably, logical blocks contain sequences with similar characteristics.

As the area sum and the dynamic algorithms perfectly balance the workload here, hyperthreading yields important improvements in execution times. However, when the scalability

Decomp. algorithm	1N	2N	4N	8N	16N
Pure block	4256.30s	2335.06s	1185.73s	723.61s	598.46s
Area sum	4128.65s	2077.56s	1081.39s	637.78s	595.05s
Dynamic	4152.61s	2085.56s	1065.50s	635.06s	621.98s
Theoretical	4141.16s	2072.90s	1038.20s	636.20s	636.20s

Table 4.5: Parallel execution times of the Droso dataset using the three pRIblast data decomposition algorithms

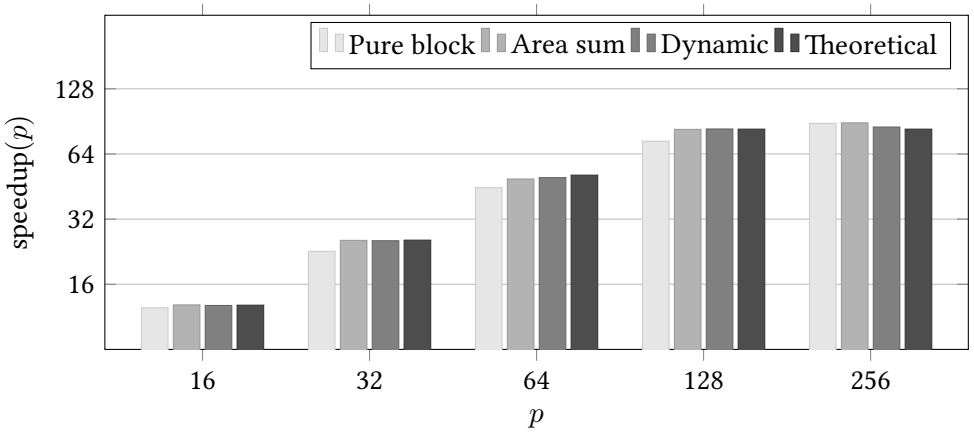

Figure 4.4: Speedups of the Droso dataset using the three pRIblast data decomposition algorithms

of the tool becomes bounded to that of the execution of one sequence (see 8 and 16 computing nodes), its performance starts to degrade. It even produces significantly slower runtimes. Similarly, in this extreme case, one thread per process (16p×1t) leads to better execution times. Again, the more elaborate data decomposition algorithms really make a difference in this scenario, and using more processes than threads reduces the execution time.

On the other hand, for the pure block procedure, hyperthreading only yields better results when the workload is not evenly balanced and the scalability is not bounded by one sole sequence. Likewise, more threads per process helps this decomposition algorithm: it cannot properly distribute workload among processes.

4.3.5 Anser

The Anser genome was the fourth dataset studied in this thesis. It is larger in size than any of the other datasets presented before. And actually, the database paging mechanism (see Subsection 3.2.1) was a must to process this FASTA file. There are so many RNAs in the database that memory rapidly becomes full with prediction candidates. Therefore, the database was divided into chunks of 2500 sequences each. Also, the parallel-aware I/O procedure (detailed

Decomp. algorithm	1N	2N	4N	8N	16N
Pure block	117877s	67188s	43229s	27684s	19977s
Area sum	113915s	61480s	33667s	20019s	14166s
Dynamic	105906s	58046s	33628s	21982s	15934s
Theoretical	105199s	52609s	26316s	17102s	14590s

Table 4.6: Parallel execution times of the Anser dataset using the three pRIblast data decomposition algorithms

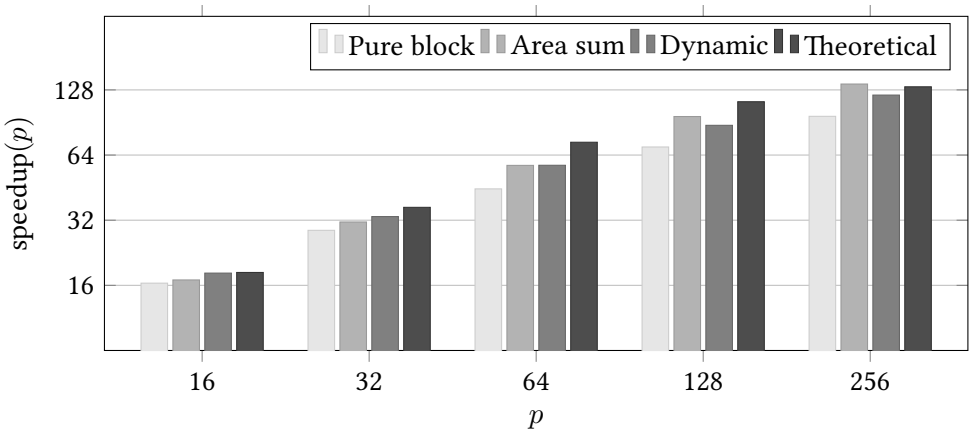

Figure 4.5: Speedups of the Anser dataset using the three pRIblast data decomposition algorithms

in Subsection 3.2.2) became significantly important here. It reduced writing times from 4000 to just 90s (using 256 processing units).

Particularly, this dataset is very interesting. Not because of its size or because it demonstrates how powerful the data decomposition algorithms and the optimizations developed in this thesis are. Instead, it is interesting because the length heuristic pretty much holds for all the sequences but one. And indeed, that sequence is a medium-small sequence that produces around 14590s of workload (0.75% of the total workload approximately). Therefore, this batch of tests presents very characteristic results.

Examining Table 4.6 and Figure 4.5, the first most noticeable result is that the pure block procedure is rather useless in comparison to the other two algorithms. There are so many sequences in the dataset that it is hard to obtain balanced blocks without no previous preprocessing of the input sequences. Also, it can be observed that the dynamic scheme works better than the area sum algorithm only when one, two and four computing nodes are used to run the pRIblast application. The reason for this being the aforementioned sequence:

On the one hand, the area sum algorithm assigns sequences to processes. That is, it gives them a responsibility: you must compute this concrete set of sequences. And moreover, following the length heuristic, the main focus is put on the longest sequences. So, it is not until

there are spare resources that the shortest sequences are processed. On the other hand, the dynamic algorithm does not assign responsibilities to processes. Processing units ask for a sequence to compute to the shared pool of lncRNAs every time they are free. In this sense, the first processes have computationally expensive sequences assigned, whereas the others skim fast through the smaller sequences to finish the computation as soon as possible. Therefore, in this scenario, the dynamic decomposition procedure performs better than the area sum algorithm, but only when there are no more than 64 processing units (i.e. the area sum is stuck processing bigger sequences first). Yet, when 8 and 16 computing nodes are used, the area sum procedure is faster, since it has spare resources to process the smallest sequences earlier (even earlier than the dynamic decomposition scheme).

Lastly, as workload is almost perfectly balanced here using the more advanced decomposition schemes, hyperthreading helps them to achieve faster runtimes. They even surpass the theoretical values sometimes. However, it fails to speed up the computation when 256 processing units are used, since the scalability is already bounded by one sequence (the medium-small sequence presented above). In the same line, more processes than threads also benefits the area sum and dynamic algorithms.

4.3.6 Anas

Finally, the pRIblast algorithm was tested against the Anas dataset. It is the largest genome of the five and, as the Anser file, it has a medium size sequence that limits the scalability of the algorithm when all 16 computing nodes are used. The database was also divided into chunks of 2500 sequences each to be able to process the dataset and, once again, the parallel-aware recomposition of results proved to be very powerful: the I/O procedure took only 392s for a 1.42TiB output file.

Results are drawn in Table 4.7 and Figure 4.6. As it was previously discussed, these plots only show results for 16 nodes. The dataset is so large that any other configuration of nodes exceeds the "Plutón" timeout value. Therefore, to compute the speedup for this dataset, the pRIblast algorithm also recorded the time spent predicting interactions for every sequence. In this sense, the function in Figure 4.1 was used to calculate the theoretical sequential time (i.e. $p = 1$), and so the speedup. Put in other words, taking the time spent computing in parallel all sequences of a dataset and summing it all together allows to obtain a rather good estimate of the sequential runtime of a genome.

One more time, it is the dynamic algorithm the one that performs the best. However, as the dataset pretty much follows the length heuristic statement, increasing processing units would benefit the area sum procedure. It would be able to process smaller sequences earlier, just like it happened with the Anser dataset.

As for the hyperthreading, it benefits the area sum and pure block algorithms. Their

Decomp. algorithm	1N	2N	4N	8N	16N
Pure block	-	-	-	-	99579s
Area sum	-	-	-	-	82692s
Dynamic	-	-	-	-	72462s
Theoretical	-	-	-	-	72462s

Table 4.7: Parallel execution times of the Anas dataset using the three pRIblast data decomposition algorithms

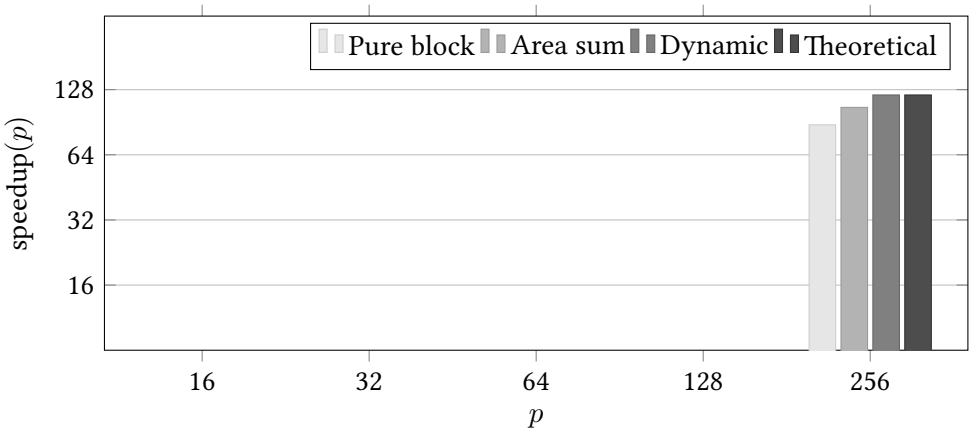

Figure 4.6: Speedups of the Anas dataset using the three pRIblast data decomposition algorithms

execution times are not, just yet, bounded by one sole sequence. Therefore, hyperthreading cannot hurt but help instead. Similarly, more threads per process helps these decomposition schemes. Their workload distribution cannot be done much more wisely increasing processes.

4.4 Early conclusion

Having discussed the results obtained during the testing phase of the pRIblast algorithm, the obvious conclusion is: the parallelization and optimization of pRIblast is a complete success. It is now possible to execute datasets that would have not been possible to process with the former RIblast application. The reason for this being that execution times have been reduced by orders of magnitude. But optimizations have played a key role too. For instance, the database paging mechanism has dramatically lowered the memory requirements to execute the tool.

Furthermore, comprehensive testing has allowed to make educated guesses regarding how and when to use each data decomposition algorithm the tool offers. These guesses are:

- Never use the pure block algorithm. It was developed in an early stage of the thesis. Its only purpose is to benchmark.
- Use the area sum algorithm when there are plenty of computing resources available (no hyperthreading needed). And, if it is possible to guarantee that the length heuristic holds, use more processes than threads (i.e. 8p×2t). In any other case, use more threads per process: 2p×8t or 4p×4t, for instance.
- Use the dynamic decomposition algorithm if there are not many computing resources available. If the dataset is small with respect to the number of processing units, hyperthreading will not hurt. Also, 8p×2t works best here.

Chapter 5

Planning, costs and methodology

This chapter focuses on presenting detailed information about the planning of the work developed throughout this thesis. First, the project will be divided into small and fine-grained tasks, and, secondly, these tasks will be plotted in a Gantt chart, together with their estimated duration and cost. Finally, an overview of the methodology chosen to build pRIblast will be presented.

5.1 Project plan

In this thesis, an incremental development model has been followed. Parallel implementations and optimizations were programmed one by one as early tests showed that significant improvements could be obtained. Therefore, the development phase mixes both analysis and implementation to progressively achieve better results.

5.1.1 Phase 1: non-recurring engineering. RIblast and the state-of-the-art

In this first phase of the project, extensive and carefully study of the RIblast algorithm was conducted. It allowed to understand the inner workings of the tool and to identify performance bottlenecks. Put differently: to find code that could take advantage of parallelism. Hence, the main focus here was put on reading and understanding the RIblast paper [7] (benchmarking results [10] [6] and similar tools [5] [4]) and comprehending its code, which is publicly available on GitHub. As a result, it was possible to draft out an MPI-OpenMP programming solution to speedup the algorithm. Key papers, such as [8] [9], were read to fully understand the importance of RIblast's application domain. Finally, a meeting was scheduled to present the study results and plan out the next phases of the thesis.

5.1.2 Phase 2: development of the parallel algorithms. pRIblast

Having fully understood the RIblast algorithm, the next step forward in this project was to develop the different pRIblast algorithms. The three data decomposition procedures were programmed following these steps:

1. Algorithm design and analysis.
2. Add MPI support.
3. Add OpenMP support.
4. Run the algorithm and verify outputs.
5. Early benchmarking against fractions of the Ursus and Droso datasets.

After all benchmarks were recorded within a spreadsheet, a meeting was scheduled to discuss results. Indeed, it was throughout this early testing stage that the algorithm showed memory and I/O issues. Therefore, before proceeding with more extensive testing in the "Plutón" cluster, the two optimizations presented in Section 3.2 (database paging and parallel-aware I/O) were developed and verified.

5.1.3 Phase 3: extensive benchmarking. "Plutón"

With three robust and parallel algorithms, it was time for in-depth benchmarking. The first step here was to select representative datasets to assess the performance of the different and optimized versions of the pRIblast application. Therefore, the Ensembl [14] genome browser was used to download the Lepi, Ursus, Droso, Anser and Anas datasets, after carefully comparing all the available FASTA files there.

Next, access to the "Plutón" cluster was granted. In this sense, the cluster's user guide was carefully read and a meeting was programmed to learn how to successfully send jobs to the Slurm scheduling software.

The third step was the development of scripts to automate the execution of all 750 tests run in this phase. It was tough work. As all executions were parametrized to achieve automation, scripts had to pass environment variables to computing nodes. Slurm makes it easy to perform it with a command-line parameter. However, it caused a very specific problem and MPI communications between computing nodes started to fail. Some days were spent figuring out possible solutions, and it was not until a thread dating from 2015 was read that the problem was finally resolved.

Finally, tests were executed and monitored from time to time. Once all tests were completed, results were recorded in a spreadsheet and a meeting was scheduled to present educated conclusions.

5.1.4 Phase 4: documentation. Thesis and miscellanea

Throughout the lifecycle of the project, notes, documents and spreadsheets were written to record important data. For instance, while the different data decomposition algorithms were developed, they were also extensively documented to be able to remember very specific implementation details at the time of writing this report. However, the main focus in this phase was to elaborate this thesis.

5.2 Project metrics

5.2.1 Time

In the end, the project took six months to complete. To be precise, it started on the first day of February 2021 and finished by the end of July 2021. However, it is important to note that all figures here are rough estimates. The project was interrupted several times because of continuous assessment during the term and final exams in May. And actually, these and other risks were taken into account at the time of first estimating the length of the project (around 300 man-hours). But anything can happen, and indeed some tasks were delayed because of unexpected events. For instance, as it was mentioned earlier, an error was encountered at the time of programming Bash scripts to automate "Plutón" executions. This event increased the duration of such task from around 10 to 20 hours, since it took several days to resolve the issue. Also, MPI one-sided communications are not studied in the degree. Therefore, reading documentation and programming one-sided "Hello world" applications delayed the project too. Or most importantly, as early tests showed that the parallel algorithm had to be optimized to be able to process large datasets, new tasks arose, leading to some more delays in the original project plan. However, they did not add up for a significant amount, and no specific actions were taken to correct the progress of the project.

Table 5.1 shows the approximate time spent on each one of the tasks described in the previous section. Figure 5.1 plots them in a Gantt chart, alongside with the most important dependencies that existed between them. Specifically, these dependencies were:

- **Finish-to-start (F-S)**. The predecessor ends before the successor can begin.
  - **Task a** $\rightarrow$ **Task c**. Before developing the first of the data decomposition algorithms (**Task c**), expert knowledge of the inner workings of the RIblast algorithm had to be acquired (**Task a**).
  - **Task c** $\rightarrow$ **Task d** $\rightarrow$ **Task e**. The development cycle followed was the incremental model. Therefore, to start developing the area sum procedure (**Task d**), the pure

Phase	Task	Time
1	a. In-depth study of the RIblast algorithm	25h
	b. State-of-the-art analysis	5h
2	c. Pure block algorithm	20h
	d. Area sum algorithm	15h
	e. Dynamic algorithm	20h
	f. Database paging optimization	20h
	g. Parallel-aware I/O procedure	20h
3	h. Select representative datasets	5h
	i. Read and understand the "Plutón" user guide	5h
	j. Bash scripting	20h
	k. Benchmarking	50h
4	l. Document the parallel algorithms	10h
	m. Document the optimizations	5h
	n. Record results within a spreadsheet	5h
	o. Thesis	95h
	Total	320h

Table 5.1: Approximate time spent on each project task

block algorithm (**Task c**) had to be finished. The same applies for **Task d** into **Task e**.

- **Task f** $\rightarrow$ **Task g**. In order to start optimizing the pRIblast I/O procedure (**Task g**), the database paging mechanism (**Task f**) had to be fully tested and functional. Recall that an incremental development cycle was followed.
- **Task i** $\rightarrow$ **Task j**. To develop the execution scripts (**Task j**), it was a must to understand the "Plutón" execution environment (**Task i**).
- **Tasks g**, **h** and **j** $\rightarrow$ **Task k**. All development had to be finished (**Task g**), datasets selected (**Task h**) and scripts programmed (**Task j**) to start benchmarking the pRIblast algorithm on the "Plutón" cluster (**Task k**).

- **Start-to-start (S-S)**. The predecessor begins before the successor can begin.
  - **Task a** $\rightarrow$ **Task b**. State-of-the-art analysis (**Task b**) could not start until some knowledge on bioinformatics had been developed (**Task a**).
  - **Task c** $\rightarrow$ **Task l**. To start documenting the parallel algorithms (**Task l**), development must have started (**Task c**).
  - **Task e** $\rightarrow$ **Tasks h** and **i**. To select datasets (**Task h**) and study the "Plutón" environment (**Task i**) may start some time before finishing the dynamic decomposition algorithm (**Task e**).

  - **Task f** → **Task m**. To start documenting optimizations (**Task m**), their development must have started (**Task f**).
  - **Task k** → **Task n**. To begin recording results within a spreadsheet (**Task n**), some benchmarking must have been conducted (**Task k**).

- **Finish-to-finish (F-F)**. The predecessor ends before the successor can end.
  - **Task e** → **Task l**. To finish development documentation (**Task l**), all algorithms have to be fully functional and production ready (**Task e**).
  - **Task g** → **Task m**. Likewise, to stop documenting optimizations (**Task m**), all optimizations must have been developed (**Task g**).
  - **Task k** → **Task n**. To finish recording benchmarking results (**Task n**), testing must be over (**Task k**).
  - **Tasks l**, **m** and **n** → **Task o**. To finish writing the thesis (**Task o**), documenting parallel algorithms (**Task l**), describing optimizations (**Task m**) and recording benchmarking results on the "Plutón" cluster (**Task n**) must have been done.

It is important to note that there is no F-S dependency between **Task e** and **Tasks f** and **g**. Those two tasks appeared to optimize the pRIblast application during the development phase. It was not possible to tell at the beginning of the project whether RIblast had memory issues or I/O bottlenecks.

5.2.2 Cost

Three people were involved in the development of the pRIblast algorithm: Iñaki Amatria Barral (Student), who was in charge of the analysis, programming and evaluation of the novel tool; and Jorge González Domínguez and Juan Touriño Domínguez (Advisors 1 and 2 respectively), who proposed the thesis topic and carefully supervised the progress of the project. Moreover, configurations of 1, 2, 4, 8 and 16 "Plutón" compute-0-X nodes were used to both benchmark and assess the scalability of the new parallel tool.

Table 5.2 shows the approximate dedication of each resource in the project. As a rough estimate, the student spent about 12 hours per week (from February to July) on the development of the project (i.e. around 2h each day). As for the advisors, dedication was computed taking into account meetings, mailing, messaging and reviewing of this thesis and other documents. Lastly, execution time in the "Plutón" cluster was calculated according to the experimental data obtained during Phase 3 (i.e. benchmarking).

Finally, Table 5.3 shows an approximate cost for the project, alongside with the hourly rate for each of the resources. Hardware cost was computed using the AWS EC2 pricing cal-

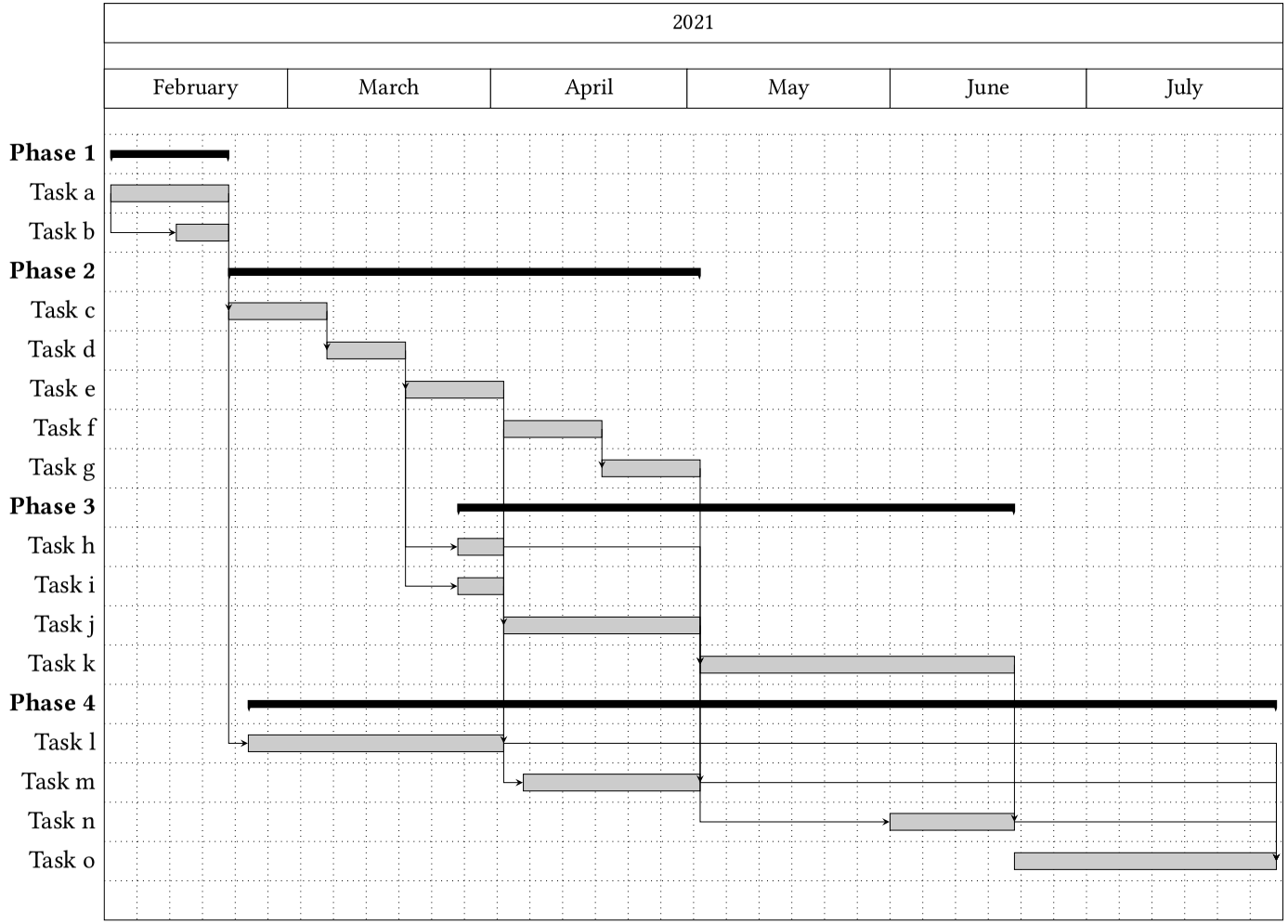

Figure 5.1: Gantt chart for the pRIblast project

Resource	Time
Student	320h
Advisor 1	35h
Advisor 2	25h
1 Node	970h
2 Nodes	500h
4 Nodes	290h
8 Nodes	165h
16 Nodes	815h

Table 5.2: Approximate dedication of each resource in the project

Resource	Hourly rate	Time	Cost
Student	35.00€/h	320h	11200€
Advisor 1	60.00€/h	35h	2100€
Advisor 2	60.00€/h	25h	1500€
1 Node	0.49€/h	970h	475€
2 Nodes	0.98€/h	500h	490€
4 Nodes	1.96€/h	290h	570€
8 Nodes	3.92€/h	165h	645€
16 Nodes	7.84€/h	815h	6390€
		Total	23370€

Table 5.3: Approximate cost for the project

culator[1], selecting an EC2 instance type with similar specifications to those of the computing nodes at "Plutón" (i.e. Linux Operating System, 16vCPUs and 64GiB of main memory).

5.3 Methodology

Diving into the development phase (Phase 2) of the pRIblast project, this section details all the important aspects of the methodology followed to obtain the final version of the software presented in this thesis.

As it was mentioned earlier in the chapter, an incremental development cycle was followed to build pRIblast (see Figure 5.2). This means that the software was developed iterating over the RIblast algorithm, and delivering a completely usable program after each successive version. Therefore, before starting any coding, the first step was to design the complete product (the three decomposition algorithms), taking as input the MPI-OpenMP draft sketched during Phase 1.

Once there was a high level design document, it was sliced into small chunks (i.e. one

[1] https://calculator.aws/#/createCalculator/EC2

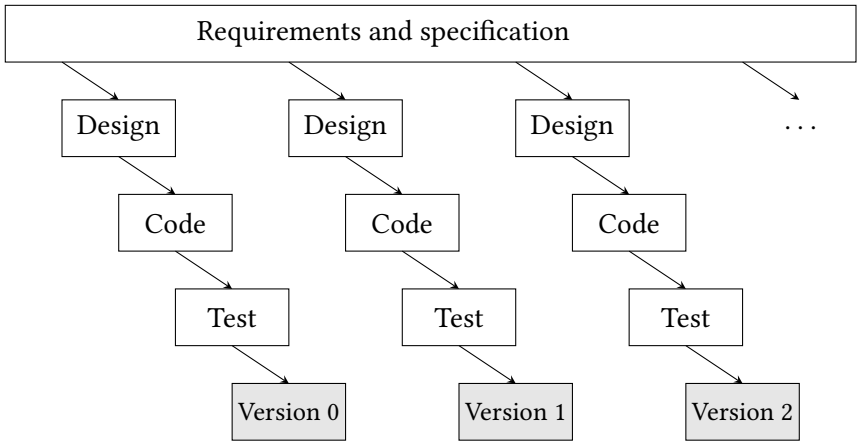

Figure 5.2: Incremental development model

chunk means one parallel algorithm) and they were set out to be built separately. That is, each chunk represents a new, independent and self-contained version of the product. So, ideally, the chunks could have been developed in parallel by different people (in practice it was impossible, since there was only one programmer in the team). Furthermore, chunk independence allowed easy and manageable identification of both risks and software bugs.

With each chunk defined, the process described in Subsection 5.1.2 was followed to develop each increment. First, more in-depth analysis and design was made, now considering language-dependent and library constraints, data types, procedures, functions and information flow. As a result, precise pseudocodes and diagrams (such as Figure 3.6) were obtained. Secondly, taking those figures as inputs, the RIblast algorithm was modified to accommodate the data decomposition algorithm in play, first adding MPI support, and, afterwards, OpenMP directives. Lastly, the new version of the software was run to verify outputs and benchmarked against small fractions of the Ursus and Droso datasets.

According to the incremental development cycle, as soon as all three increments were production ready, the software was complete. However, because benchmarking and testing showed that improvement could be made in terms of memory management and I/O procedures, it was necessary to plan out another incremental cycle to develop the two optimizations described earlier in Chapter 3 (Section 3.2). Again, it means producing a high level design document, slicing it up into small chunks, and, finally, implementing each one independently. This showed that the incremental development methodology may have not been a good choice to develop pRIblast. It does not easily adapt to unexpected changes in requirements. In this sense, an agile development cycle would have suited pRIblast a lot better, since it is a lot like inventing: discovering what and how you need as you go.

Chapter 6
Conclusions

To conclude, this last chapter draws both conclusions and future lines of work to further improve the pRIblast algorithm.

6.1 Conclusions

Roughly speaking, the main conclusion is that pRIblast is a huge success and proves to be a step forward towards comprehensive computational lncRNA-RNA interaction analysis. It does not only fulfill all starting goals, but it also has surpassed all initial expectations in terms of performance and scalability. And now, after thorough testing, the application is ready and set to help scientists all around the globe.

So, recalling Chapter 1, all objectives set out at the start of the project have been met. Is pRIblast capable of effectively and efficiently exploit all hardware in high performance computing facilities? Yes, it is. Indeed, as broadly extended parallel programming technologies have been used to develop the tool, it easily adapts to any cluster or supercomputer similar to that of Section 2.3. Moreover, pRIblast has been carefully designed to minimize communications and avoid unnecessary stress in interconnection networks.

Is benchmarking successful? Sure it is. Evaluation (Chapter 4) shows that pRIblast is capable of analyzing in less than 21 hours (using 256 processing units) a large dataset that would have needed around 101 days of computing time with the original tool. Furthermore, the outstanding results presented in the aforementioned chapter do not only come from the fact that RIblast has been parallelized. The parallel-aware I/O procedure developed to speed up the execution of large-scale datasets has proved to be increasingly important to process these volumes of data. And what is more, the two largest datasets (Anser and Anas) would have never been executed if the database paging mechanism had not been implemented, not even with the former RIblast application.

And last, has this thesis introduced the student to research/academic work? Yes, it has.

It has forced the student to take all the developed knowledge throughout his four years of bachelor's studies, and apply it to a completely unexplored and different discipline (i.e. bioinformatics). Also, solid an deep study of the domain had to be made before setting out any project plan or milestone. Finally, all relevant results and educated conclusions were recorded within a paper: this thesis.

6.1.1 Personal thoughts

I have enjoyed every minute spent on this thesis, except those fixing bugs and reading through compiling error transcripts. Those were not entertaining at all. But I am sure all of them have served a purpose.

This project has helped me to not only learn or dive deeper into topics I like (i.e. biology and parallelism), but to grow as a student/engineer too. I had never thought that I would read through extensive MPI documentation to learn and use one-sided communications all by myself. What is more, little did I know that I would genuinely enjoy reading through bioinformatics papers. But I have, and I am really proud of it.

So, to conclude, now that pRIblast is publicly available as free and open source software, I am eager to see what people are capable of achieving with it. For instance, will it help to further advance in cancer research? I do not know if that will ever happen, but if it does, I will be more than happy to think that I was a part of it.

6.2 Future lines of work

Even though comprehensive benchmarking has showed that pRIblast performs undoubtedly well when it is supplied with large datasets, it also asserts that there is still room for some improvement in workload distribution. Indeed, all reasoning behind the heuristics developed in this thesis is based on an assumption: "the number of seeds found in a sequence is proportional to its length". And while evaluation has proved that it somewhat matches reality, it has also proved that the novel tool falls a bit short when it does not (see Lepi and Ursus as the most representative tests). Also, because the assumption may or may not hold for some datasets, the pRIblast user guide provides the reader with a rather puzzling configuration of algorithms, processes and threads to ensure maximum performance is achieved. Therefore, as a next step to refine pRIblast and guarantee continuous improvement, domain-specific knowledge must be introduced to better rank sequences at the time of distributing workload. That is, to classify sequences using more informed and determinant parameters than their lengths. Ideally, developing a lightweight procedure that can be easily parallelized too.

Moreover, RIblast implements two major steps: database construction and RNA interaction search. Although the main focus in this thesis has been put on the second one (since it is

way more computationally expensive and it is executed more often), the database construction procedure can take advantage of parallel programming paradigms too. In this sense, a completely parallel execution workflow would be achieved, from initiation to completion.

Appendices

Appendix A

pRIblast user guide

A.1 Requirements

To compile and execute pRIblast, the following packages are required:

- GNU `Make`, which automatically determines which pieces of a large program need to be recompiled (v3.82).
- GCC, the GNU Compiler Collection (v7.3.0).
- `Open MPI`, an open source MPI implementation (v3.1.0).
- `OpenMP`, LLVM OpenMP runtime - dev package (v3.1).

Note that there exist different MPI implementations, such as OpenMPI or MPICH. Please, make sure your library has support for RDMA (i.e. one-sided communications) before compiling pRIblast. Also, it is possible to build the program with older GCC and OpenMP versions, but they will not be officially supported.

A.2 Compilation

First of all, download the source code from the official pRIblast repository[1]. Either use Git or download a copy from GitHub. Secondly, navigate to the `src` directory and choose which version of the parallel algorithm you would like to use (see below for further advice):

- `pure-block`: naive pure block decomposition algorithm.
- `area-sum`: greedy algorithm which decomposes sequences according to their size.
- `dynamic-decomp`: share all algorithm, all processes see the same set of sequences.

[1] https://github.com/amatria/pRIblast

And lastly, navigate to the folder of the algorithm you wish to use and let GNU Make automatically compile it for you. As a result, there will be a newly created binary file named pRIblast in your current working directory.

A.3 Execution

To execute pRIblast, fetch the MPI runtime interface as follows

```
mpirun -np <p> -x OMP_NUM_THREADS=<t> pRIblast <options>
```

where `<p>` is the number of processes that will exist in the MPI group and `<t>` the number of threads spawned per MPI process.

As for the program options, RIblast's official repository[2] provides a fairly detailed list of available execution modes (i.e. database construction and RNA interaction search) and per mode flags. However, pRIblast implements new options to have fine-grained control over the execution of the parallel algorithm. These options are:

`(db)` `-c <uint>`, sets the database chunk size.
`(ris)` `-p <path>`, sets a local path for fast writing of temporary output files.
`(ris)` `-t <0 | 1>`, debug execution.

A.3.1 Execution example

Suppose you want to execute the `area-sum` algorithm in a 16-node multicore cluster using the Droso dataset, a database chunk of 500 sequences and 1 process per node with 16 threads each. Furthermore, there exist a local, temporary disk attached to every node located in `/tmp/scratch`. To do so, first download the drosophila dataset from the Ensembl genome browser[3]. Secondly, use the script `misc/preprocess.sh`

```
./preprocess.sh /path/to/drosophila.fa
```

to split the FASTA file into two different datasets: one with all RNAs (used to build the target database), and an lncRNA only (used to search interactions). Third, create the fragmented database running the pRIblast database construction step

[2] https://github.com/fukunagatsu/RIblast
[3] ftp://ftp.ensembl.org/pub/release-97/fasta/

```
mpirun -np 1 \
       pRIblast db -i /path/to/db-drosophila.fa \
                   -o /path/to/db-drosophila \
                   -c 500
```

And finally, execute the interaction search step on 16 nodes issuing the following command

```
mpirun -np 16 -x OMP_NUM_THREADS=16 \
       pRIblast ris -i /path/to/ris-drosophila.fa \
                    -d /path/to/db-drosophila \
                    -o /path/to/out-drosophila.txt \
                    -p /tmp/scratch
```

A.3.2 Configuration of threads, processes and algorithms

To achieve maximum performance, use the pRIblast algorithms as follows:

- Do not use the pure block algorithm. Its only purpose is to benchmark.
- Use the area sum algorithm when there exist plenty of compute resources with respect to the dataset size. No hyperthreading needed. And if the length heuristics holds, use more processes than threads (i.e. 8p×2t). In any other case, use more threads per process: 2p×8t or 4p×4t, for instance.
- Use the dynamic algorithm if the number of computing resources is small with respect to the dataset size. Hyperthreading will not hurt here. 8p×2t works best.

List of Acronyms

GCC GNU Compiler Colection. 35

GPGPU General Purpose Graphical Processing Unit. 10

I/O Input and Output. 9

lncRNA long noncoding RNA. 1

MPI Message Passing Interface. 10

NAS Network Attached Storage. 33

OpenMP Open Multi-Processing. 13

RDMA Remote Direct Memory Access. 12

SIMD Single Instruction Multiple Data. 6

SPMD Single Program Multiple Data. 9

Bibliography

[1] S. Azzi, W. A. Habib, and I. Netchine, "Beckwith-Wiedemann and Russell-Silver syndromes: from new molecular insights to the comprehension of imprinting regulation," *Current Opinion in Endocrinology, Diabetes and Obesity*, vol. 21, pp. 30–38, 2014.

[2] L. Natarelli, L. Parca, T. Mazza, C. Weber, F. Virgili, and D. Fratantonio, "MicroRNAs and long non-coding RNAs as potential candidates to target specific motifs of SARS-CoV-2," *Non-Coding RNA*, vol. 7, p. 14, 2020.

[3] O. Wapinski and H. Y. Chang, "Long noncoding RNAs and human disease," *Trends in Cell Biology*, vol. 21, pp. 354–361, 2011.

[4] I. Antonov, A. Marakhonov, M. Zamkova, and Y. Medvedeva, "ASSA: fast identification of statistically significant interactions between long RNAs," *Journal of bioinformatics and computational biology*, vol. 16, p. 1840001, 2018.

[5] F. Alkan, A. Wenzel, O. Palasca, P. Kerpedjiev, A. Rudebeck, I. H. P. Stadler, and J. Gorodkin, "RIsearch2: suffix array-based large-scale prediction of rna-rna interactions and sirna off-targets," *Nucleic Acids Research*, vol. 45, p. e60, 2017.

[6] S. U. Umu and P. P. Gardner, "A comprehensive benchmark of RNA–RNA interaction prediction tools for all domains of life," *Bioinformatics*, vol. 33, pp. 988–996, 2017.

[7] T. Fukunaga and M. Hamada, "RIblast: an ultrafast RNA-RNA interaction prediction system based on a seed-and-extension approach," *Bioinformatics (Oxford, England)*, vol. 33, pp. 2666–2674, 2017.

[8] K. Lee, K. Leung, S. L. Ma, H. C. So, D. Huang, N. L. Tang, and M. Wong, "Genome-wide search for SNP interactions in GWAS data: algorithm, feasibility, replication using Schizophrenia datasets," *Frontiers in Genetics*, vol. 11, p. 1003, 2020.

[9] C. Huang, D. Leng, S. Sun, and X. D. Zhang, "Re-analysis of the coral Acropora digitifera transcriptome reveals a complex lncRNAs-mRNAs interaction network implicated in Symbiodinium infection," *BMC Genetics*, vol. 20, p. 48, 2019.

[10] I. Antonov, E. Mazurov, M. Borodovsky, and Y. Medvedeva, "Prediction of lncRNAs and their interactions with nucleic acids: benchmarking bioinformatics tools," *Bioinformatics*, vol. 20, pp. 551–564, 2018.

[11] D. J. Lipman and W. R. Pearson, "Rapid and sensitive protein similarity searches," *Science*, vol. 227, pp. 1435–1441, 1985.

[12] The MPI Forum. (2015) MPI: A Message Passing Interface (v3.1). Accessed: 09/01/2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[13] OpenMP Architecture Review Board. (2020) OpenMP API 5.1 Specification. Accessed: 09/01/2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf

[14] K. L. Howe et al., "Ensembl 2021," *Nucleic Acids Research*, vol. 49, pp. 884–891, 2021.