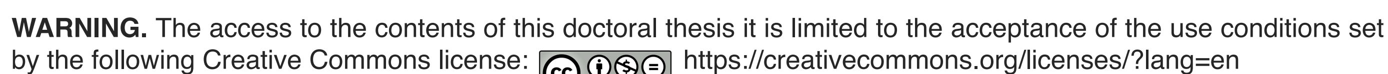
ADVERTIMENT. L'accés als continguts d'aquesta tesi queda condicionat a l'acceptació de les condicions d'ús establertes per la següent llicència Creative Commons: http://cat.creativecommons.org/?page_id=184

ADVERTENCIA. El acceso a los contenidos de esta tesis queda condicionado a la aceptación de las condiciones de uso establecidas por la siguiente licencia Creative Commons: http://es.creativecommons.org/blog/licencias/

WARNING. The access to the contents of this doctoral thesis it is limited to the acceptance of the use conditions set by the following Creative Commons license:  https://creativecommons.org/licenses/?lang=en

Universitat Autònoma de Barcelona

Escola d'Enginyeria

Departmament d'Arquitectura

de Computadors i Sistemas Operatius

Parallelizing Population Genetics Applications

Thesis submitted by **Carlos Montemuiño Sosa** in fulfillment of the requirements for the degree of Doctor by the Universitat Autonòma de Barcelona. This work has been developed in the Computer Architecture and Operating Systems Department of the Autonomous University of Barcelona under the advice of Dr. Porfidio Hernández Budé and Dr. Sebastián Ramos-Onsins.

Bellaterra, January 2021

Parallelizing Population Genetics Applications

Thesis submitted by Carlos Montemuiño Sosa in fulfilment of the requirements for the degree of Doctor by the Universitat Autonoma de Barcelona, under the supervision of Dr. Porfidio Hernádez Budé and Dr. Sebastián Ramos-Onsins.

Bellaterra, January 2021

Author

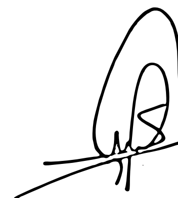
DNI 23867070Q
Carlos Montemuiño Sosa

Thesis Advisors

PORFIDIO HERNÁNDEZ BUDÉ - DNI 18414251Z

Firmado digitalmente por PORFIDIO HERNÁNDEZ BUDÉ - DNI 18414251Z
Nombre de reconocimiento (DN): c=ES, sn=HERNÁNDEZ BUDÉ, givenName=PORFIDIO, serialNumber=IDCES-18414251Z, cn=PORFIDIO HERNÁNDEZ BUDÉ - DNI 18414251Z
Fecha: 2021.01.05 12:27:28 +01'00'

Dr. Porfidio Hernádez Budé



DNI 35056252C
Dr. Sebastián Ramos-Onsins

Acknowledgments

My wife, Karina, who put aside her academic and professional career and was ready for playing a single mother role in my worst moments of stress, and whose support and encouragement helped me not to give up.

My sons, Julieta and Santiago, who do not understand very well why their daddy is spending so much time in non-leisure activities but behave as adults embracing disagree & commit. I am very proud of them.

My mother Nair, my sister Norma, and my brother-in-law Jorge, whose devotion to helping me is almost infinite.

To my father, Gualberto, who always dreamed of working on science projects and inspired me to join science.

My nephews Juan Manuel and Florencia, whose continuously quest for academic and professional improvement, served as inspiration.

My advisors, Porfidio Hernández Budé and Sebastián Ramos-Onsins, who always believed in me more than I did. Without their continued guidance, inspiration, insights, and encouragement, this research would not be possible.

I extend my gratitude to all the great people I had the privilege to work with throughout the investigation. In particular, thanks go out to Gonzalo Vera, Anna Sikora, Antonio Espinosa, Juan Carlos Moure, Eduardo Cesar, Dolores Rexachs, and Tomàs Margalef.

Abstract

With the increasing availability of genome-scale data for genetic research, molecular population geneticists need to work with more complex models, which cannot be done in a time-fashion using the standard coalescent methods. This scenario led to the development of several alternative numerical simulation applications. Despite the ever-increasing access to High Performance Computing (HPC) clusters in the academy, it is not being leveraged in the field of population genetics. Parallelizing existing applications is hard to achieve by developers without a comprehensive understanding of the HPC, and new applications only take advantage of multiprocessing capabilities from a single computer.

This thesis proposes a technique to parallelize coalescent applications and effectively use all the available processing power from an HPC cluster. We use a strategy to reduce the intra-node communications in the message-passing paradigm. This solution allows for getting better scalability for coalescent applications that require generating millions of replicas. As a result, population geneticists can use the standard coalescent tools for running Approximate Bayesian Computation (ABC) analysis without relying on less accurate applications.

We have evaluated our strategy parallelizing the de facto standard coalescent application and run experiments at genome-scale in a real HPC cluster. We have obtained significant performance gains in tuning different aspects of our approach, leading to a 4x speedup over our initial parallelization, which accounted for a 50x speedup over the reference coalescent application.

Keywords: HPC, parallel programming, population genetics, coalescence, sequential Markov coalescent

Resum

Amb la creixent disponibilitat de dades a escala de l'genoma per a la investigació genètica, els genetistes de poblacions moleculars han de treballar amb models més complexos, el que no pot fer-se en un temps determinat utilitzant el mètode coalescent estàndard. Aquest escenari va dur a el desenvolupament de diverses aplicacions alternatives de simulació numèrica. Tot i l'accés cada vegada més gran a les agrupacions de computació d'alt rendiment (HPC) a l'acadèmia, no s'està aprofitant en el camp de la genètica de poblacions. L'establiment de paral·lels entre les aplicacions existents és difícil d'aconseguir pels desenvolupadors sense una comprensió completa de la HPC, i les noves aplicacions només aprofiten les capacitats de multiprocessament d'una sola computadora.

En aquesta tesi es proposa una metodologia per establir un paral·lelisme entre les aplicacions coalescents i utilitzar eficaçment tota la potència de processament disponible d'un grup d'HPC. La metodologia introdueix una estratègia per reduir les comunicacions intra-node en el paradigma de pas de missatges. Aquesta solució permet obtenir una millor escalabilitat per a les aplicacions coalescents que requereixen la generació de milions de rèpliques. Com a resultat, els genetistes de poblacions poden utilitzar les eines coalescents estàndard per executar l'anàlisi de Computació Bayesiana Aproximada (ABC) sense dependre d'aplicacions menys precises.

Hem avaluat la nostra estratègia establint un paral·lelisme amb l'aplicació coalescent estàndard de facto i executant experiments a escala de l'genoma en un conglomerat HPC real. Afinant diferents aspectes de la nostra metodologia, hem obtingut importants guanys de rendiment, donant lloc a una velocitat de 4x per sobre de la nostra paral·lelització inicial, que representava una velocitat de 50x per sobre de l'aplicació coalescent de referència.

Paraules clau: HPC, programació paral·lela, genètica poblacional, coalescència, coalescència seqüencial de Markov

Resumen

Con la creciente disponibilidad de datos a escala del genoma para la investigación genética, los genetistas de poblaciones moleculares tienen que trabajar con modelos más complejos, lo que no puede hacerse en un tiempo determinado utilizando el método coalescente estándar. Este escenario llevó al desarrollo de varias aplicaciones alternativas de simulación numérica. A pesar del acceso cada vez mayor a las agrupaciones de computación de alto rendimiento (HPC) en la academia, no se está aprovechando en el campo de la genética de poblaciones. El establecimiento de paralelos entre las aplicaciones existentes es difícil de lograr por los desarrolladores sin una comprensión completa de la HPC, y las nuevas aplicaciones sólo aprovechan las capacidades de multiprocesamiento de una sola computadora.

En esta tesis se propone una metodología para establecer un paralelismo entre las aplicaciones coalescentes y utilizar eficazmente toda la potencia de procesamiento disponible de un grupo de HPC. La metodología introduce una estrategia para reducir las comunicaciones intra-nodo en el paradigma de paso de mensajes. Esta solución permite obtener una mejor escalabilidad para las aplicaciones coalescentes que requieren la generación de millones de réplicas. Como resultado, los genetistas de poblaciones pueden utilizar las herramientas coalescentes estándar para ejecutar el análisis de Computación Bayesiana Aproximada (ABC) sin depender de aplicaciones menos precisas.

Hemos evaluado nuestra estrategia estableciendo un paralelismo con la aplicación coalescente estándar de facto y ejecutando experimentos a escala del genoma en un conglomerado HPC real. Afinando diferentes aspectos de nuestra metodología, hemos obtenido importantes ganancias de rendimiento, cuadruplicando el speedup de nuestra paralelización inicial, la cual representaba una mejora de 50x sobre la aplicación coalescente de referencia.

Palabras clave: HPC, programación paralela, genética poblacional, coalescencia, coalescencia secuencial de Markov

Table of Contents

1 Introduction 1

1.1 The Standard Coalescent and Simulation 1

1.2 High Performance Computing 2

1.2.1 Multiprocessors and Parallel Architectures 3

1.2.2 Architectural Models 3

1.2.3 Multicore and Manycore Processors 5

1.2.4 Processor-Memory Performance Gap 6

1.2.5 Parallelization and Programming Models 7

1.2.6 Programming Paradigms 11

1.2.7 Performance Evaluation 12

1.3 Related Studies 14

1.4 Motivation 16

1.5 Objectives 17

1.6 Contributions 17

1.7 Organization of the Thesis 18

2 Population Genetics and Computer Simulation 19

2.1 Population Genetics 19

2.2 Computer Simulation 20

2.2.1 Forward-in-time Simulation 20

2.2.2 Coalescent Simulation 21

2.2.3 Approximate Methods 24

3 Parallelizing Coalescent Applications 26

3.1 Application Characterization 26

3.2 Coarse Grain Parallelization 28

3.3 Communication Patterns and Memory Management 30
3.4 Approaching Multilocus Analysis with the Hierarchical Manager-Worker 31
4 Experimental Results 34
4.1 Coarse-Grain Approach 34
4.1.1 Experimental Setup 34
4.1.2 Discussion 35
4.2 Hierarchical Manager-Worker 37
4.2.1 Experimental Setup 37
4.2.2 Discussion 37
4.3 Multilocus Applications 41
4.3.1 Experimental Setup 41
4.3.2 Discussion 41
5 Conclusions and Future Work 43
5.1 Future Work 44
5.2 List of Publications 45
6 Bibliography 46
Appendix A – msparsm 54
Pre-requisites 54
How to Build 54
How to Use 54
Appendix B – mlcoalsim-v2 55
Quick Start 55
Enabling MPI in the IDE 55
How to Build 55
How to Run the Examples 55

List of Figures

Figure 1. Schematic representation of Hudson's ms process to generate a set of replica samples.... 22

Figure 2. Data structure of the ARG node. 23

Figure 3. Simplified process for validating an evolutionary model with ABC. 24

Figure 4. Processor speed evolution along the years (log scale). 6

Figure 5. Impact on the spatial space to generate one single replica. 26

Figure 6. Impact on the temporal space, using the same setup as Figure 5....................................... 27

Figure 7. a) Time spent to generate the ARG, distributed into the algorithm's steps, and I/O processing. b) Maximum resident set size as a function of recombination and mutation parameters. 27

Figure 8. Manager-Worker topology from msPar. 28

Figure 9. Manager-Worker topology from *msParSm*. 31

Figure 10. *mlcoalsim* is parallelized using a hierarchical manager-worker model. 32

Figure 11. Speedup and efficiency of *msPar* compared against Hudson's *ms*.................................... 35

Figure 12. Average resident set size consumed by each MPI process when running the experiments with *mspar*... 36

Figure 13. Speedup analysis of case study 1 using Hudson's *ms* as a baseline.................................. 38

Figure 14. Obtained speedup for case study 1. 38

Figure 15. Obtained speedup for case study 2. 39

Figure 16. Average consumed memory for case study 2.. 39

Figure 17. Speedup analysis of *enhanced parallelization* when compared against the original *mlcoalsim*... 42

CHAPTER 1

1 Introduction

Computer science can be divided into several areas, each one intended to address problems from different disciplines, spanning biology, chemistry, physics, and even cognitive science. Regardless of the domain, a commonality is going after ever more resource-intensive problems, where the resource can be computational, memory, storage, or a combination of them. One of such disciplines is bioinformatics, where High Performance Computing (HPC) is used in many fields, for example, molecular biology, genomics, and population genetics, just to name a few.

The cost of genome sequencing has substantially reduced over the past quarter-century, with total costs[1] figures per Megabase of DNA sequence dropping below $0.008 in 2020 vs. $5200 in 2001. [1]. This cost reduction had contributed to democratizing the use of whole-genome data in genetic research, especially in the evolutionary and population genetics field.

Population genetics plays a significant role in human genetic research, linking hypothesis on sequence variation with empirical observations. Computer simulation software has long been used to explore analytically intractable genetic models [2].

Continuous advances in numerical simulation and the wide availability of computational resources allow researchers to use numerical simulation to test mathematical models in virtual populations, and even analyze genetic data [[3], [4]]. Consequently, a plethora of simulators are available, each tailored to a specific scenario, forcing molecular population geneticists to choose a simulator depending on the research being conducted.

In this chapter, we present an overview of coalescent based-simulation applications and HPC systems, the challenges this kind of application poses for parallelization, related studies, and we conclude by presenting our thesis proposal.

1.1 The Standard Coalescent and Simulation

Research in the genetics population field was supported by mathematical modeling for over 90 years. Empirical testing of theoretical models is practically impossible for organisms with

[1]. Including labor, utilities, sequencing instruments, and indirect costs.

long generation times. Computer simulation software has traditionally been used to explore analytically intractable genetic models, providing insights about how patterns of genetic variation within and between populations have been shaped.

Existing simulation applications from population genetics fall into two classes, *forward-in-time* and *backward-in-time*. The latter, also known as the *standard coalescent*, is the mainstream process used in molecular population genetics because of its efficiency and flexibility [5].

Several software simulation applications have been developed along the years, being Hudson's *ms* [6] the most widely coalescent simulator [2]. The essence of this application is applying the Monte Carlo method to generate an Ancestral Recombination Graph[2] (ARG) for an initial sample of chromosomes and then placing random mutations over the resulting ARG. The ARG is constructed by stochastically simulating evolutionary events (such as coalescence, migration, or recombination) occurring on the ancestral material of sampled alleles backward in time until the Most Recent Common Ancestor (MRCA) to all the alleles is found [7]. This application is computational intractable when handling very large samples (i.e., genome-scale data sets) and recombining genomic regions, and complex population substructures [[8], [9]].

1.2 High Performance Computing

High Performance Computing (HPC) is used to solve large problems via supercomputers, fast networks, and massive storage devices.

The lazy programmer era of waiting for faster hardware to improve the performance of a program is over. If we want to exploit the new generation of processors, we must write parallel programs [10].

Most applications have not been structured to exploit parallelism [11], nor have they been designed for HPC and multicore hardware. Population genetics's applications (e.g., Hudson's ms and variants) are not an exception.

Parallelizing an existing application requires more work than just rewriting the code to use some parallel library (e.g., OpenMP, MPI, CUDA, etc.). This is just the tip of the iceberg. Achieving sustained gains of performance at scale requires understanding HPC system components' interactions, both from software and hardware, algorithms, libraries, programming interfaces, and many more aspects.

[2] An ancestral recombination graph is a directed graph, where each node represents a chromosome contributing some genetic material to the present-day sample.

Implementing a parallel version of an existing application is not different. The developer needs to have a complete understanding of all these interactions. Thus she can identify the limiting factors for scalability and make the right trade-offs during the implementation. Besides the complexity associated with grasping how all the moving pieces work together, this process will probably differ depending on the application domain.

In this section, we focus on the factors that need to be considered when parallelizing coalescent applications. We begin with multiprocessors, the parallel architecture models, and the multicore/manycore architectures, which are the essence of any HPC cluster these days. We continue with a description of the processor-memory-performance gap, which is of particular interest for coalescent applications. Then, we cover the programming models and parallelization techniques. Finally, we review the performance evaluation topic, where we provide definitions for the performance metrics of interest.

1.2.1 Multiprocessors and Parallel Architectures

We adhere to the definition provided by Hennessy and Patterson in [10]:

> *A multiprocessor can be defined as computers consisting of tightly couple processors whose coordination and usage are typically controlled by a single operating system and that share memory through a shared address space.*

Multiprocessors exploit thread-level parallelism (TLP) through two different software models, *parallel processing*, and *request-level parallelism*. The former means dividing a problem into tasks and execute these tasks simultaneously, while the latter refers to the execution of an independent process that may originate from different users. Both models rely on multi-threading, a technique to execute multiple threads in an interleaved fashion on a single multiple-issue processor. The amount of work assigned to each thread is usually dubbed as the *grain size*. It is essential to differentiate TLP from instruction-level parallelism (ILP), as a thread may comprise hundreds to millions of instructions, making it possible to exploit data-level parallelism with TLP [10].

1.2.2 Architectural Models

Many parallel architectures have been proposed to address parallel processing, leading to parallel computers' classification according to different factors. These classifications, usually called *taxonomies*, help us to think about solving problems in parallel computing.

Michael Flynn proposed and developed a classification over 50 years ago, focusing on how processing elements interact with data [12], often referred to as *Flynn's taxonomy*. This classification classifies a parallel computer according to two independent dimensions,

instruction, and *data*. Each dimension can have only *single* or *multiple* states, leading to the following definitions:

- SI (Single Instruction): All processors execute the same instruction.
- MI (Multiple Instruction): Different processors *may* execute different instructions.
- SD (Single Data): All processors operate on the same data.
- MD (Multiple Data): Different processors *may* operate on different data.

Then, a taxonomy of four paradigms is provided:

- Single Instruction - Single Data (SISD)
- Multiple Instruction - Single Data (MISD)
- Single Instruction - Multiple Data (SIMD)
- Multiple Instruction - Multiple Data (MIMD)

SISD is a paradigm that makes sense for computers with a single processor. The current tendency is to have processor architectures with over one processing unit, even in the low-end processor range (e.g., RaspberryPi). *This paradigm may be considered obsolete.*

MISD implies several processors, each receiving different instructions operating on the same data stream. The output of one processor becomes the input of the next processor. It is an extremely rare paradigm often classified by the community as impossible to implement [13]. Other authors argue that neural networks and data flow machines are examples of this architecture [14].

SIMD hardware allows us to work on multiple data items in parallel with a single instruction. For example, adding a constant value to each element of an array in parallel. Processing units following the SIMD paradigm are referred to as SIMD units or *vector units* [15].

When seen from a software viewpoint, this paradigm can be reformulated as SPMD, where all processors use the same program, and they operate on multiple data streams. For example, a climate model is a *single program*, comprising an atmospheric, ocean, and ice component with independent data and instructions (i.e., *multiple data*) in each component.

SIMD may be inefficient in case of load imbalance, which could degrade the performance because processors become idle while waiting for other processors to finish their work [16].

Another source of inefficiency is *branch divergence*. For example, let's consider the following pseudo-code containing a for-loop over an array, where a computation is done depending on a branch condition:

```
for i=0, n
   if A(i) > 0 then
      A(i) = A(i) + 1
```

```
      else
         A(i) = A(i)*2
    end
```

In this case, only one instruction can be sent at a time in SIMD. Therefore, utilization is decreased by 50% since the system needs to issue instructions for both branches.

Finally, we have the MIMD paradigm, which describes almost all parallel computers today, including personal computers. Unlike SIMD, which has synchronous and deterministic execution, MIMD can be synchronous or asynchronous, deterministic, or non-deterministic.

Flynn's taxonomy is not the only work for classifying parallel architectures. Other classifications based on different factors exist—for example, Feng's classification and Handler's classification [13]. Feng's classification differentiates parallel architectures based on the number of bits processed in parallel. Handler's classification categorizes parallel architectures on the number of processor control units (PCU), the number of arithmetic control units (ACU), and the number of bit-level circuits (BLC). Finally, other authors proposed modifications to Flynn's taxonomy. For example, Duncan's taxonomy includes pipelined vector processors [17].

1.2.3 Multicore and Manycore Processors

The industry has shifted away from designing higher speed processors, which are not power-efficient in computation. Still, transistors are invested in adding more execution units[3] on a single chip. As shown in Figure 1, the trend for designing higher speed processors has stopped around 2005 [11].

---

[3]. We also interchangeably refer them as **cores** throughout the document.

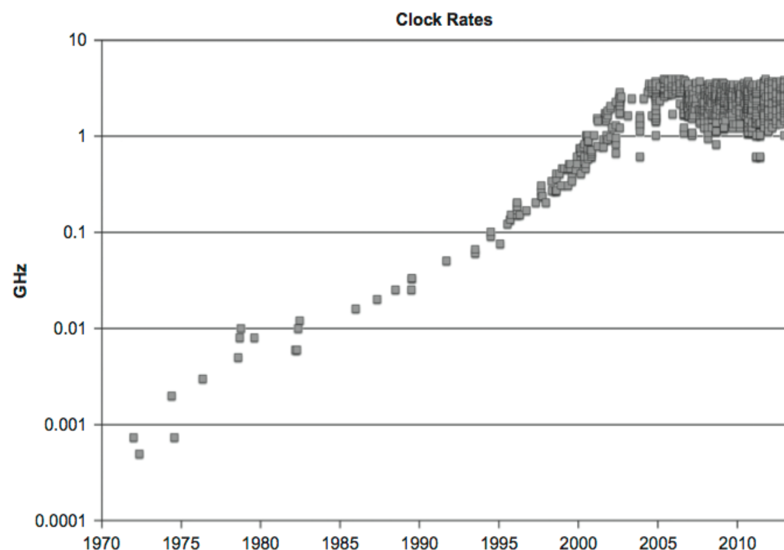


Figure 1. Processor speed evolution along the years (log scale). This figure shows how processors are not following the increasing speed trend since 2005. (c) 2013 Jim Jeffers and James Reinders, used with permission.

The multi-threading strategy eventually evolved into cloning the processor to full extent, moving from one large and fast superscalar processor to smaller and not as powerful *multicore processors*. Increasing power and silicon costs grew faster than performance, influencing both the clock frequency and the number of cores that can be put into a single chip. This led to the *manycore architecture*, which is often designed with many tiny *cores*, offering a massive parallelism level that can be exploited by proper code designed to run on such architecture. If all cores are similar, then the architecture is referred to as *homogeneous manycore architecture*. Otherwise, it is referred to as *heterogeneous manycore architecture* [[10], [15]].

In the last two decades, manycore and multicore processor architectures have rapidly gained tremendous popularity, becoming the state of practice. Although there is no formal distinction between manycore and multicore, the common consensus is that manycore processors contain a larger number of physical cores to achieve a higher degree of explicit parallelism. For example, Intel's Xeon Phi Knights Landing architecture (code name KNL), with up to 72 processor cores on a single chip [18]. Multicore processors typically contain fewer but faster cores designed to deliver high performance for both serial and parallel applications. One such example is the SkyLake-SP (code name SKX) processor family, with up to 28 processor cores per socket [19].

1.2.4 Processor-Memory Performance Gap

Processors are much faster than memory, making them sit idle waiting for data, especially with processors using in-order execution. This is commonly known as the memory wall [20], and frequently reported in the literature as the *processor-memory performance gap* [21]. The processor-memory performance gap is a limitation, also referred to as the *von Neumann*

bottleneck, which especially dominates the performance of applications with workloads that exhibit memory-intensive behaviors [[22], [23]].

Much research has been done on reducing memory access latencies, such as lockup-free caches, prefetching, speculation (superscalar computing), and multi-threading. Many of these architectural solutions entailed increasing bandwidth requirements, which ultimately became a more fundamental impediment to high performance [24]. Other approaches have been taken to solve the memory bandwidth issue, such as the Process-in-memory (PIM) technology, and three-level cache hierarchies. This last approach addresses the bandwidth requirements for high-end processors with multiple cores, which have more significant bandwidth requirements than for single cores. For example, the Intel Core i7 6700 processor has a total peak demand bandwidth of 409.6 GiB/s [10].

1.2.5 Parallelization and Programming Models

Designing and developing parallel programs has traditionally been a manual process, but approaching the parallelization in a fully automated way is also an option. A parallelizing compiler analyzes the source code, identifies opportunities for parallelism, often addressing loops as the most frequent target for automatic parallelization, and finally converts the serial program into a parallel program. A semi-automated approach could consist of using compiler directives to help the compiler in the parallelization process. Chances to embrace automatic parallelization (either entirely or semi-automated) are high in case of time or budget constraints. However, several important caveats might apply to automatic parallelization, such as erroneous results and performance degradation.

On the other hand, manually developing parallel programs is a time consuming, complex, and error-prone process. Additionally, there is not a strictly mechanical process we can follow. However, all is not lost. A methodology comprising four steps, known as Foster's methodology [25], can be leveraged to develop parallel programs: partitioning, communication, agglomeration, and mapping.

In the *partitioning* step, computation is divided into small tasks, and focus is put on identifying tasks that can be executed in parallel. Then, communication needs among tasks are identified in the *communication* step. During *agglomeration*, tasks can be aggregated into a single composite task, if it makes sense. Finally, tasks are assigned to processes/threads (including aforementioned composite tasks) in the *mapping* step, in a way that communication is minimized, and work is balanced across the processes/threads. We observe that partitioning and communication steps focus on exploiting parallelism, while agglomeration and mapping steps focus on performance gain.

The potential performance obtained from the parallelization process does not end with Foster's methodology, but it heavily depends on the programming model we use.

MIMD architectures can roughly be divided into shared memory, distributed memory, and hybrid architectures. From a developer's point of view, a shared memory computer can be depicted as a collection of processors connected to global memory. On the other hand, a distributed memory computer can be seen as a collection of processors, each with local memory and an interconnect used to communicate with other processors. Finally, architectures are combining both shared memory and distributed memory components. We will review these three classifications in the following sections, where we detail what programming models make more sense to use in each one of them.

1.2.5.1 Shared Memory

In *shared memory* (SM) architectures, the memory is globally accessible to all available processors, which may have local copies of a global memory subset, using a cache coherency technique to maintain copies' consistency.

Significant advantages from this architecture have a global address space, leading to fast data sharing between processing elements (i.e., reduced communication latency), without explicit communication. This kind of architecture is sometimes called uniform memory access (UMA), since all processors have a uniform latency from memory, regardless of the memory organization.

Disadvantages from this architecture include potential bottlenecks in the shared memory to the CPU path, forcing developers to implement synchronization properly.

Another potential problem with SM architectures is ***false sharing***, which especially applies to SMP system, where each processor has a local cache. False sharing occurs when threads running on different processors refer to different locations within the same cache line (e.g., array data structures) [26]. Unlike true sharing, which can be addressed by programmatic synchronization constructs, false sharing may not be visible within the application [27]. Its associated performance penalty could significantly degrade performance. Thus several tooling for false detection has been produced, usually requiring heavy instrumentation and could incur high overhead [[28], [29]].

POSIX threads (or Pthreads) and OpenMP are both APIs for SM programming, being the latter the dominant language today [30].

OpenMP is an Application Program Interface (API) that defines a set of *compiler directives* used to achieve multi-threaded shared memory parallelism. It follows an explicit

programming model known as the *fork-join model*, giving the programmer full control over the parallelization process. Thread communication is achieved by sharing variables [31].

The fork-join model comprises a master thread executing sequentially until a parallel region is found. A set of threads is then created (i.e., forked), which are executed in parallel. When all threads complete the work, they synchronize (i.e., join), and the master thread continues executing sequentially. The number of threads is independent of the number of processors. If we have more threads than available processors, then all threads will not run in parallel, but one after another. When there are more processors than threads, we will have idle processors, which translates into bad efficiency.

1.2.5.2 Distributed Memory

Distributed memory (DM) architectures can be depicted as a collection of processor-memory pairs connected by a network [31]. Unlike SM architectures, DM has different memory latency and bandwidth, depending on whether a processor is accessing the memory locally or from another processor. This architecture is also called non-uniform memory access (NUMA) [10].

This type of architecture has several advantages. For example, the memory is scalable with the number of processors. Another advantage is the ability to make use of commodity parts, which makes this architecture potentially cost-effective.

On the other hand, the developer is responsible for many of the details of the inter-process communication. However, it could also be seen as an advantage, as the developer has full control of operations flow. Another potential disadvantage is the added complexity of distributed data structures.

When a process (or program) running on one processor-memory pair requires data located in a different processor-memory pair, then a technique known as *message passing* is used.

Communication is often an essential aspect of performance and correctness in distributed memory systems. Messages are like handshakes, involving two partners, a *sender* and a *receiver*.

Sending a message is relatively slow, with startup times (i.e., latency) taking thousands of cycles. Typically, once the message has started, the additional time per byte (i.e., bandwidth) is relatively small.

Some useful approaches may be taken for reducing the effect of latency:

- Reduce the number of messages by mapping communicating entities on the same processor.
- Bundle small messages having the same sender and receiver, and send a combined message instead.

- Avoid processes asking for data, but send data early as soon as it is ready.

There are other aspects of message passing systems worth considering. If communication is *blocking*, we can read *deadlock* if processors cannot proceed until message communication is finished. This can be avoided by carefully coordinating all the processors, but a most straightforward approach to prevent deadlock is using *non-blocking* communication, which allows a processor to send a message before the *receive* operation is finished.

Most recent versions of MPI provide one-sided communication operations that might boost performance. It avoids handshaking for sending messages, but a process can take an item from memory and puts it into the memory of a different process, running on whatever node.

One of the most known implementations of message passing is the Message Passing Interface (MPI). There are many MPI implementations, being Open MPI [[32], [33]], between the most widely used today in HPC.

1.2.5.3 Hybrid Architectures

Most parallel computers today combine both SM and DM components, making what is known as *hybrid architecture*. Placing accelerators, such as Graphics Processing Units (GPUs) and CPUs, with shared access to the main memory, makes hybrid architectures complex to program.

Accelerators and GPUs play an essential role in supercomputers. For example, coprocessors account for 56% of peak FLOPs in the Tianhe-2A supercomputer, and GPUs provide around 98% of peak FLOPs in the Summit supercomputer. Although very important, accelerators and GPUs are not mandatory for achieving the highest performance. The counterexample is the Fugaku supercomputer, which captured the TOP500's first place with 100% ARM-based architecture [34].

GPUs are massively parallel multiprocessors based on the manycore architecture, with hundreds of cores per processor, combining SIMD fine-grain parallelism and slightly coarse-grained MIMD. Leveraging the parallelism from GPUs require executing mutually independent tasks over large data entries. Not all problems are data-parallel. Therefore, depending on the algorithm, its highly parallel structure does not necessarily lead to better efficiency than general-purpose CPUs.

Different levels of parallelism can be achieved by combining programming paradigms. For example, OpenMP can be combined with MPI for maintaining sustained performance gains at a finer granularity. If accelerators like GPUs are present, then parallelization can be further exploited via CUDA or compiler directives. The drawback is that programmers must know several languages, resulting in code harder to debug, analyze, and maintain.

The greater availability of hybrid architectures is entailing a different parallelization paradigm. It is hard leveraging both SM and DM architectures by solely using OpenMP, MPI, CUDA, or similar programming languages. The hybrid programming strategy is getting more and more traction, being MPI + OpenMP a popular choice.

1.2.6 Programming Paradigms

Regardless of the parallel architecture, distribution, and control of work between processors are not always the same, but programming paradigms heavily influence it.

One such paradigm is *divide-and-conquer* [14], a strategy that iteratively partitions a problem into subtasks of the same size, achieving the smallest possible tasks. It comprises three operations, *divide*, *compute*, and *join*, leading to a tree type program structure.

Another well-known paradigm is *pipelining*, a useful technique for improving throughput, defined as task completion rate per unit time. Following a functional decomposition, a program is divided into subtasks, where subtask one must be completed to start the next subtask.

Another fundamental strategy is the *manager-workers* pattern[4] [35]. The basic idea is having a *manager* keep track of many tasks and a set of *workers* performing the tasks. The manager assigns tasks to the workers, which communicates back the results once the task is done.

The manager has a different meaning depending on the architecture. With DM, a manager is usually a processor, while in the case of SM, a manager can be a shared data structure (e.g., a queue).

Sometimes the manager may become a bottleneck because of the number of messages sent to the workers. A naïve approach is sending many tasks at once. Another approach uses multiple manager/worker teams, with some communication between managers, where every worker might play the manager role.

In DM architectures, we have the situation that a worker might run out of tasks or have too many. Therefore, the worker needs to communicate, either asking for more work or asking for help in doing the assigned tasks.

Another problem with the manager-worker paradigm lives in load balancing. We cannot get high performance if we are always waiting for some slow worker.

---

[4]. Usually referred as “master-worker” or “master-salve” in the literature.

1.2.7 Performance Evaluation

We cannot evaluate performance by only focusing on individual factors, but we need to evaluate the application-system interplay.

When determining how efficient the parallel code is, we need to consider how the time is spent. For example, the time spent in communications to other processors, time waiting for a message to be received, or the wasted time waiting for other processors.

We need an unbiased metric for determining whether program A's performance is better than program B's performance.

Hennessy and Patterson provide a formula for the speedup metric in their seminal textbook [10] that we can adapt to our needs. Instead of comparing a task using an enhancement versus a task without using such an enhancement, we compare a parallel version of a program versus its serial form. First, we define serial and parallel time:

- *SerTime*(n) = Time of **best serial program** to solve A for input of size n.
- *ParTime*(n, p) = Time of the parallel program to solve A for input of size n, using p processors.

1.2.7.1 Amdahl's Law and Speedup

Amdahl’s law is a formulation showing that given an enhancement applied to a program, the improvement gains we can get are limited by the program's fraction that does not contain the enhancement [36]. Amdahl suggested that general-purpose parallel computing was not viable, and effort should focus on improving the serial part of programs.

We can redefine our previous formulas based on Amdahl’s concepts. Let s be the fraction of time on operations that are performed serially (i.e., a fraction of ParTime(n, p)):

$$ParTime(n,p) \geq SerTime(n)\left[s + \frac{1-s}{p}\right]$$

Therefore, we could define the speedup as:

$$Speedup(n,p) = \frac{1}{s + \frac{1-s}{p}}$$

Additionally, we can formulate the speedup is limited to the serial fraction s, irrespectively of how many processors are used:

$$0 < Speedup <= 1/s$$

Amdahl’s law has been put in question by researchers in the parallel computing community. For example, Gustafson [37] questioned Amdahl’s law's validity as he reported near-linear

speedup when working with multiprocessors. His work turned out into an alternative of the speedup formula, coined "scaled speedup"[5], and also referred to as Gustafson's Law:

$$Scaled speedup = p + s(1 - p)$$

Suggesting that Amdahl's law is inappropriate in the context of MPP, as Gustafson did, is not valid, as shown by Yuan Shi [38], but both laws are equivalent.

We rely on the total execution time as the basis for performance evaluation:

$$Speedup(n, p) = \frac{SerTime(n)}{\mathrm{ParTime}(n, p)}$$

1.2.7.2 Linear and Superlinear Speedups

If speedup equals p, then we say the program has linear speedup, but if the speedup is bigger than p, we say the program has super-linear speedup.

Reasons for linear speedup can be attributed to overheads, such as communication, synchronization, input/output and memory access, and load imbalance.

Using Amdahl's law for scalability analysis presents two fundamental challenges:

- Determining what the serial fraction is.
- Having the same number of total instructions for both serial and parallel programs.

There is a special kind of serial algorithm, known as non-structure persistent (NSP). Depending on the inputs, at least one parallel version requires fewer instructions than the best serial implementation [38]. Therefore, we can achieve super-linear speedup as a side-effect of the parallelization process. Another valid reason for super-linear speedup can be attributed to the larger memory capacity of parallel computers. When a big problem is put on a small amount of RAM, many data will probably be sent back and forth to the disk. Having much more RAM increases the chances of having a higher fraction of the program in RAM instead of disk, which translates into better performance.

1.2.7.3 Scalability and Efficiency

We create applications for solving problems that cannot be manually addressed, either because of time restrictions, spatial restrictions, or both. If we focus on the spatial aspect, we find that increased load is common for performance degradation [39].

---

[5]. Gustafson used $\boldsymbol{N}$ to denote the number of processors.

Scalability is a term we use to describe an application's ability to cope with larger problem sizes [31] and qualify the performance degradation.

When dealing with HPC and parallel program performance, scalability is associated with a program's ability to use more resources in parallel [40]. In our case, we are interested in using more processors. We find two scalability types: **weak scalability** and **strong scalability**, which we can tightly couple to the speedup's evolution as we add more processing elements [41]. This makes the so called **efficiency** metric, which helps us to determine what *good enough performance* means when linear speedup cannot be achieved [31]:

$$Efficiency(n,p) = \frac{SerTime(n)}{p(ParTime(n,p))}$$

If we simultaneously increase the problem size and the number of processors involved, and the efficiency is kept fixed, the application has weak scalability. Suppose we keep the problem size fixed while increasing the number of processors, and the efficiency is kept fixed. In that case, the application is said to have strong scalability (what Amdahl considered in his research).

We could be tempted to assume that achieving weak scalability is easier than strong scalability, but this could become a fallacy depending on the architecture. For example, depending on the memory hierarchy we have, the working dataset could not fit in a multicore processor's last level cache. In such a case, the efficiency could be worse for weak scalability than strong scalability [42].

1.3 Related Studies

The coalescent theory provides an efficient framework for the study of molecular diversity within and between species that revolutionized the field of population genetics. Since the first application's appearance implementing the standard coalescent, Hudson’s *ms* [6], population geneticist started to switch from empirical and historical approaches to a theoretical approach leveraging numerical simulation.

Genetic diversity is studied under many different evolutionary and demographic scenarios. Depending on the model's emphases, a wide variety of applications for simulating data had emerged over the years. Some applications put the focus only on selection, recombination, demographics, population structure, migration. Others try to address biologically realistic models by handling several evolutionary processes at the same time.

We start with Hudson’s *ms*, which allows recombination, migration, variable population size, and produce a genealogy, but does not cover different mutation models nor variable recombination rates. *SPLATCHE* [43] simulates diversity taking environmental heterogeneity

into account, but does not model selection nor recombination. *SelSim* [44] allows modeling both selection and recombination but leaves out variable population size and produces no genealogy. *Simcoal2* [45] includes the possibility to simulate datasets with arbitrary recombination rates between partially linked *loci*, enabling the possibility of multiple coalescent events per generation and complex migration patterns, but lack of selection.

Good examples of improvements on top of Hudson's *ms* are *mlcoalsim* [46], including selection. Another one is msHot [47], which incorporates crossover and gene conversion hotspots.

Finally, we may find applications including codon or amino-acid evolution models, such as *NetRecodon* [48].

Next-generation sequencing (NGS) [49] has democratized access to genome data, making genome-scale data commonplace in genetic research. Disagreements between expectation and observation started to be reported, such as discrepancies in linkage disequilibrium predictions [50]. This situation sparked the need to model more complex evolutionary scenarios [8] (e.g., hotspots of recombination). Simulating large regions of DNA became computationally intractably by coalescent simulators, a scenario that is exacerbated when using large recombination rates. Then, approximations to the coalescent for working at genome-level with large recombination rates were propose. Most relevant examples are the sequentially Markov coalescent (SMC) [51], with *fastsimcoal2* [52] as the reference implementation, and *MaCS* [53].

Several works have explored optimizations to the standard coalescent, resulting in variants that can manage genome-scale data. We find alternative implementations of the exact coalescent, including algorithmic and data structure optimizations, such as *scrm* [54] and *msprime* [55]. Moreover, we also find approximations to the coalescent, such as the sequentially Markov coalescent and the Markov Chain Monte-Carlo.

Approximated methods solve the issue of working at genome-level with large recombination rates; however, these algorithms have been reported to suffer a loss of accuracy and fail to capture important evolutionary events. For example, when sampling populations separated by reduced gene flow [56]. Another reported issue relates to ignoring type 2 recombination events (from Marjoram and Wall's classification [51]), which may impact the mean and variance of most recent common ancestor times when simulating long sequences [57]. Consequently, approximate methods have not been wide-adopted in the population genetics community, where Hudson's *ms* is still the reference application.

1.4 Motivation

Plummeting DNA sequencing costs has democratized access to hundreds of millions of reference genomes, rocketing the study of genomic variation in the last years. Population genetics is no longer restricted to a small set of anonymous *loci*[6], but it utilizes genomic data more often.

Research in population genetics is driven through three different lines: (i) research in statistics and methods to measure variability; (ii) develop software for the analysis of high throughput data; (iii) perform computational simulations of evolutionary models to analyze and interpret the available data.

Population genetics is hypothesis-driven science. The coalescent standard is the dominant paradigm used to generate *in silico* data that is later used for hypothesis testing. Moving from genetic data to whole-genome data makes the population geneticists increasingly rely on High Performance Computing (HPC) clusters.

Generation of genealogy samples is of little use if it cannot be done in a viable time-frame. The ability to efficiently compute demographic and mutational parameters is critical as genomic data become increasingly large. For example, the computational time is a decisive factor for many studies, such as Approximate Bayesian Computing (ABC), which requires millions of simulations to analyze complex evolutionary models [58].

Designing applications targeting HPC environments platforms is nontrivial. Parallel programming requires in-depth knowledge of multicore and manycore architectures, skills in several programming paradigms (e.g., MPI, CUDA, and OpenMP), and it makes debugging and reproducibility harder. On the other hand, introducing new applications designed from scratch is also nontrivial. Proving that a new application produces accurate data as existing reference applications is very challenging.

By definition, coalescent applications are based on a stochastic process, where replica samples are independently generated from each other, exhibiting embarrassingly parallel properties. This characteristic opens the door for parallelization, leading to potential improvements in the execution time. Some research has been done on exploiting multi-threading, such as *mlcoalsim* [[46], [48]], but it did not consider the other major issue from coalescent applications: they are memory eager.

[6] Loci is the plural of *locus*, which refers to the place on a chromosome where an *allele* resides. An *allele* is the bit of DNA at that place. A *locus* can be seen a template for an *allele*. Thus, an *allele* is an instantiation of a locus.

Despite parallelizing an application can alleviate the execution time and lead to manage bigger problems than can be achieved with reference applications (i.e., Hudson's *ms*), it still does not fully solve the scenario of working at genome-scale. A population geneticist is at a crossroad, deciding whether to use a reference coalescent application, potentially tuned to run in parallel on a fat node, or taking the risk and use an approximate application such as *MaCS*.

Considering all aspects described above, this thesis's motivation is proposing a blueprint for parallelizing coalescent applications, enabling population geneticists to work at genome-scale. Our strategy is to parallelize the reference coalescent application and take advantage of the full computational potential of HPC systems commonly available in research departments. Therefore, population geneticists can potentially take a step forward in their analysis without stepping out of the comfort zone of working with the standard coalescent.

1.5 Objectives

This thesis proposes a blueprint for parallelizing standard coalescent applications, allowing molecular population geneticists to experiment with complex evolutionary models with long genomic regions and large recombination rates. We focus on message passing and HPC systems to accomplish this goal.

With this scope, we have developed the following specific objectives:

- Identify the reference coalescent applications used in population genetics, targeting both singlelocus and multilocus analysis.
- Conduct a performance analysis of coalescent applications. The complexity of evolutionary models is a determining factor regarding compute and memory resources. The objective is to determine which parameters have the most impact on memory, compute, and execution time, regardless of whether the experimentation is done at a genetic or genomic level.
- Design and implement a parallelization strategy for the reference application.
- Coalescent applications do not look different in structure. Features and optimizations aside, the two-step algorithm for generating a replica must remain. Therefore, we have the stretch goal of extending our parallelization approach to other coalescent applications. From the many options out there, we believe that parallelizing *mlcoalsim* will add value to the population genetics community. We will be delivering parallelized versions for both single-locus and multi-locus analysis and paving the way to apply the same parallelization approach to further coalescent.
- Include our parallelized coalescent applications in the ngasp platform [59].

1.6 Contributions

We have designed and implemented a technique to parallelize standard coalescent applications used for singlelocus analysis. The technique includes a strategy to reduce the computing and communication gap using a masterless master-worker approach. As a result, this technique reduces the total execution time of standard coalescent applications, increasing the efficient use of an HPC system's computational resources. Thus, it is possible to run faster coalescent simulations generating whole-genome genealogies with long recombination rates. The software is publicly available at https://github.com/cmontemuino/msparsm.

We adapted and applied the parallelization strategy to *mlcoalsim*, an application used for multilocus analysis. The code is publicly available and can be used as a starting point for one of the open lines of investigation: https://github.com/cmontemuino/mlcoalsim-v2.

This work is included in the *ngasp* [59] package, spanning software to analyze high throughput data, such as analyzing genome variability using NGS data. We contribute toward understanding the effects of selection due to domestication by scanning the levels and patterns of genomic variation in commercial interesting organisms such as *Sus scrofa* (pig), *Cucumis genus* (melon, cucumber), or *Prunus genus* (peach, almond).

1.7 Organization of the Thesis

The organization of this thesis is as follows. In Chapter 2, we provide a primer and population genetics and how its dependency on computer simulation. We present the proposed parallelization approach in Chapter 3. We show the iterations to improve our solution's scalability, and then we finish with a proposal to parallelize a multilocus coalescent application. Chapters 4 covers the experimental results, where we describe the experimental setup and provide insights about our findings. We provide our conclusions and open lines of investigation in Chapter 5. We conclude in Chapter 6 with the bibliography.

Chapter 2

2 Population Genetics and Computer Simulation

In this chapter, we start introducing the population genetics field. We discuss approaches to scientific research, and especially the relevance of computer simulation. Finally, we describe the coalescent standard and approximate methods in detail.

2.1 Population Genetics

Knowing the genetic structure of any species is utopic. It would require a complete description of every individual's genome and spatial location at one instant of time. As soon as new individuals are born, this description gets incomplete. Natural populations are always changing. Population genetics is a theoretical area of genetics that does not try to get a complete description of species but focuses on the evolution of few *loci* in a population [60].

Population genetics is concerned mainly with genetic variation within species. This variation results from finding different nucleotides at a specific site of independently sampled *alleles*, known as single nucleotide polymorphisms (SNPs). Population geneticist's quest is finding what evolutionary forces could have led to such divergence between individuals within the same species. For example, the identification and characterization of SNPs that confer increased susceptibility to complex human diseases, such as Type 2 diabetes and breast cancer [61].

Geneticists study population-level change in three ways: empirical, historical/comparative, and theoretical [4].

The empirical approach involves observing living populations over time and direct experimentation with the variables that affect population variation. The most obvious limitation of this approach is studying organisms with long generation times.

The historical/comparative approach consists of observing differences within existing populations and trying to infer population histories. This method is limited by various assumptions, such as the constant-rate molecular clocks or the neutrality of synonymous polymorphisms[7].

Finally, the theoretical approach involves studying how hypothetical populations might behave in *forward-time*, starting from an initial population, or in *backward-time*, starting from the observed sample and in the present generation. The main limitation of this

[7] Also called mutations in the literature.

mathematical modeling is that it requires significantly simplifying the proposed models. Thus, mathematical models need to be tested. One way to do this is by returning to the empirical approach and study living populations through many generations to validate the theory. As commented already, the empirical approach is not practical. An alternative is generating *in silico* data and test the models in virtual populations. It is for this reason that numerical computer simulation is used in population genetics.

2.2 Computer Simulation

Population geneticists use a variety of computer simulation applications. Each of these applications has been designed with different modeling emphases, covering just a subset of evolutionary and demographic scenarios.

Existing simulation applications fall into two classes, *forward-in-time* and *backward-in-time*. The *forward-in-time* approach is designed to start from an initial population and track its evolution under various genetic models, over multiple generations, with samples usually drawn from either the final or one of the last several generations. The *backward-in-time* approach also referred to as the *standard coalescent* [62], takes a lineage approach. For each gene, a sample of copies is followed back in time to the most recent common ancestor (MRCA). Then it introduces genetic information into the generated genealogy, such as mutations, going forward from the MRCA up to the current generation.

2.2.1 Forward-in-time Simulation

This simulation strategy allows to track an entire population of individuals (and all their chromosomes) and observe population properties at any generation. This high accuracy comes at a high computational cost. Execution times are quadratic with respect to the population size. Therefore, most forward simulators can simulate small populations. For example, 10400 individuals in humans have become very challenging already [63]. Its application is usually preferred in studies that require investigating the whole population's evolution or alternative when working at a short-time scale [[2], [61]].

There are many forward-in-time simulation applications (see [8] for a comprehensive list). The oldest ones are FPG and EASYPOP [64]. FPG simulates a population of constant size undergoing various evolutionary processes, such as mutation, recombination, natural selection, and migration. It allows for a total genome length of up to 1000 segments, each limited to 32 SNPs, limiting its usage to model a genome of up to 3.2 Mbp[8]. EASYPOP [65] has been optimized for maximum speed and makes a way more efficient use of the memory.

[8] Mega base pairs

It allows managing thousands of SNPs, as long as there is enough memory in the machine where it executes.

Another classic forward-in-time simulator are *simuPOP* [66] and SFS_CODE [67]. SimuPOP provides a general-purpose simulation environment that can be used to simulate various chromosome types (e.g., autosome, chromosome X and Y) and arbitrary mating schemes. However, running complex models requires the geneticist to write its own macros (in the Python language). SFS_CODE is the state-of-the-art forward simulator [63]. It can be run on an HPC cluster by manually partitioning the experiment and aggregating the output results when processing finishes.

Memory management is one of the most fundamental issues of forward-in-time simulators. Dynamic memory allocations invoked in the inner parts of nested loops quickly dominate runtimes. Recent research has been proposed to alleviate the memory allocation issue. For example, *Libgdrift* [68] implements several performance code optimizations, such as reusing allocated memory and using array-like structures to leverage reference principles' spatial and temporal locality.

Finally, the use of GPU processing has been proposed for certain types of forward-in-time simulations. For example, GO Fish [69] is a CUDA-based implementation targeting the single-locus Wright-Fisher forward algorithm, which is considered embarrassingly parallel.

2.2.2 Coalescent Simulation

Contrary to the individual-based approach of forward-in-time simulation, the coalescent works on a sample DNA fragment's genealogy. Therefore, the coalescent is more computationally efficient than forward-in-time simulations. The difference in execution time gets more significant when population sizes are large relative to sample size [2].

One crucial feature from the standard coalescent is that recombination events are not independent. It significantly impacts runtimes, making it scale more than quadratically with the genetic region's length [51].

Hudson's *ms* [6] is the reference implementation of the standard coalescent. This application generates many independent replica samples (i.e., genealogies) under different parameters users specify to represent a model under study. Besides being very flexible from a feature point of view and computationally intractable when simulating long genomic regions, Hudson's *ms* does not support all possible scenarios required by geneticists. For example, including recombination and gene conversion hotspots (e.g., *msHot* [47]) and allowing selection and analysis of multi-locus data in linked and independent regions (e.g., *mlcoalsim* [46]).

In Figure 2, we see the global replica process generation of Hudson's *ms*. At the bottom, each replica sample is independently and sequentially generated by an orchestration module split up into two independent steps: Genealogy Construction and Mutation Assignment. Genealogy construction is driven by three routines simulating a random set of recombination, migration, and coalescence events. The mutation assignment step works over the ARG produce by step 1, generating the different segregation loci to produce the final chromosomes. The stochastic feature of the coalescence process is based on the Monte-Carlo method.

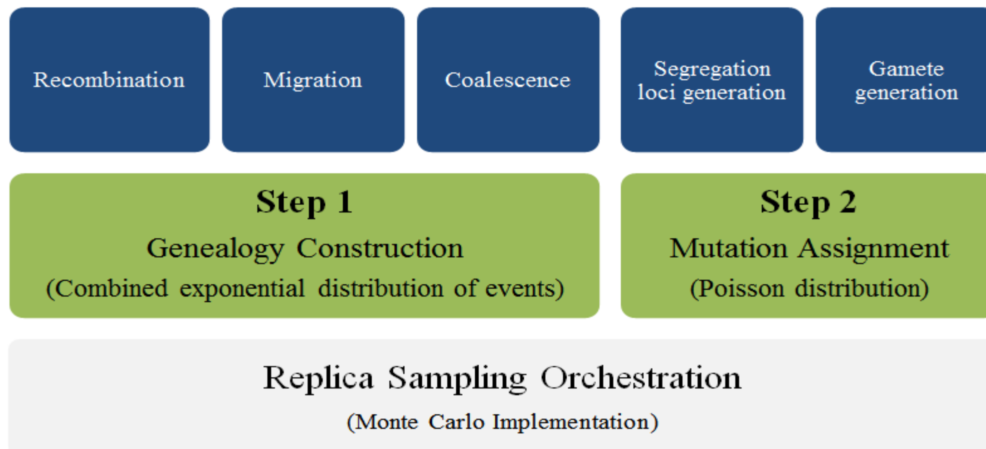


Figure 2. Schematic representation of Hudson's ms process to generate a set of replica samples.

The first step is to build the ARG and put it into an array-based data structure. In Figure 3, we show how the ARG nodes represent each chromosome's segment's history. Each node points to the start segment. Then, a parent-child structure is used to represent the history. The ARG is shared by the three-event routines from step 1 (recombination, migration, and coalescence), updating it as soon as one event occurs.

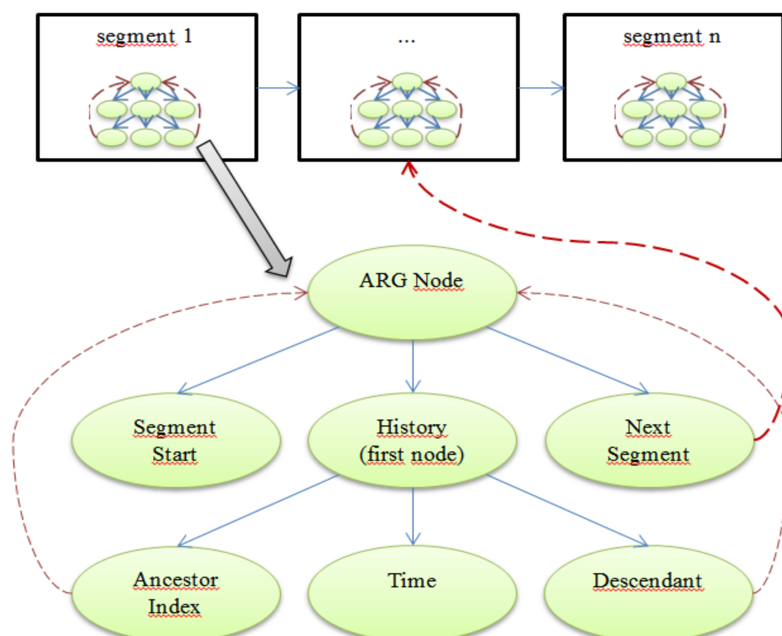


Figure 3. Data structure of the ARG node. Given n samples, the first n nodes are the tips of the tree. The remaining node is the ancestral to the sampled chromosomes.

The complete ARG structure is the input of step 2, where mutations are randomly assigned (following a Poisson distribution), and the replica samples data is finally produced. Hudson's *ms* uses several global shared variables. One such variable maintains the ARG state, which is updated by the recombination and mutation steps. Another variable is used to maintain all the replica samples used to generate an output file when the algorithm finishes generating all the request replicas.

The coalescent is a stochastic process that requires millions of simulations to obtain probability distributions for each parameter included in a proposed evolutionary model. Many of the algorithms using the coalescent for parameter estimation use Markov chain Monte Carlo (MCMC) sampling.

Research in molecular population genetics does not end with the genealogies generated by a coalescent application. A usual next step is validating the evolutionary model used as input of the coalescent application and ultimately determine what model is the best fit. It requires an analysis that is an iterative and nonlinear procedure, including sampling, markers, population inference, and evolutionary coalescent model testing steps [70].

Two statistical techniques are used for estimating parameters, namely Maximum Likelihood (ML) and Bayesian inference. Bayesian inference uses prior knowledge of the parameter values to obtain probability distributions, while ML uses single-point estimates with confidence intervals. Analytical validation can be applied to species with simple evolutionary history, but more complex species (e.g., the human species) may require to make use of

Approximated Bayesian Computation (ABC) [71]. In Figure 4, we can see an overview of the validation process using ABC.

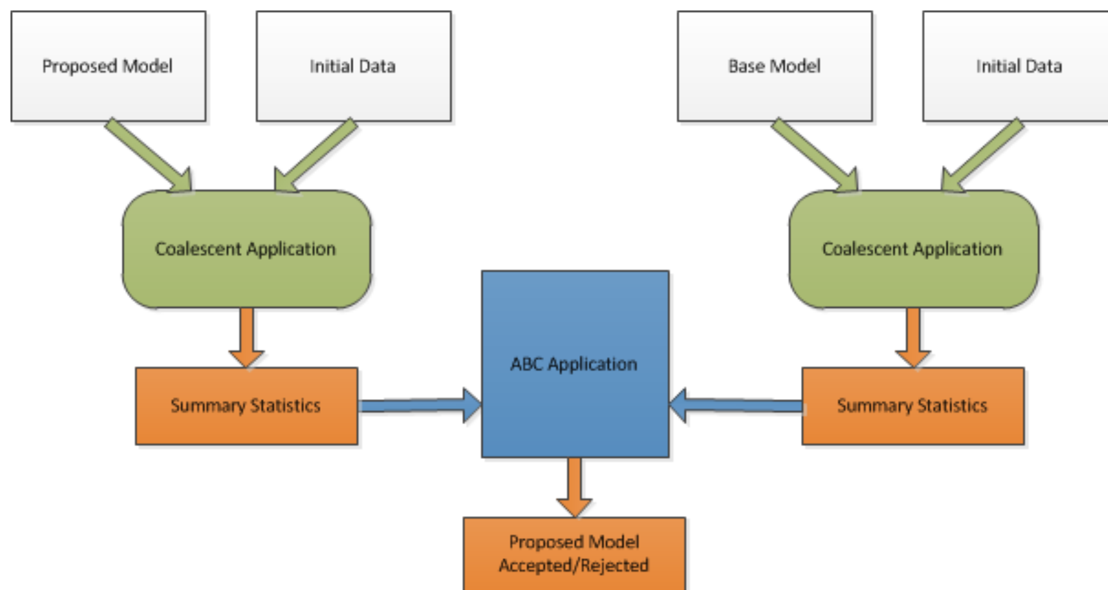


Figure 4. Simplified process for validating an evolutionary model with ABC.

The general idea consists of proposing an evolutionary model; choosing summary statistics capturing the structure of the data; simulate millions of replicas that are equivalent to the observed data in several populations and sampled individuals; filtering simulations with similar summary statistics to the observed data; selecting the model with the highest probability; estimating parameter value, and conducting quality controls to decide whether the proposed model is either accepted or rejected.

2.2.3 Approximate Methods

The standard coalescent suffers when simulating whole-genome data with large recombination. More computationally efficient approximated approaches have been proposed, namely the Sequentially Markov Coalescent (SMC) [72] and Markovian Coalescent Simulator (MaCS) [53].

SMC assumes that the recombination process operates as a Markov process, allowing it to scale linearly with the genetic region's length. The improvement in execution time comes at the cost of approximating the full ARG, incurring in loss of accuracy when compared against the standard coalescent.

MaCS is a modification of the SMC algorithm sitting between SMC and the standard coalescent. It is slower than the SMC algorithm, but it produces virtually identical data to the standard coalescent data. *MaCS* is way faster than Hudson's *ms*, especially when simulating

large genetic regions. For example, simulating a total of 10 million replicas with a genetic region of 100 Mbp, a sample size of 100, and a scaled recombination rate parameter of 100, *MaCS* has been reported to execute in less than 3 minutes, while Hudson's *ms* required 13 days to complete [53].

Approximated methods can suffer a loss of accuracy for specific evolutionary models. For example, SMC can be a poor approximation when modeling features such as the length of admixture blocks. On the *MaCS* side, while it is possible to adjust its accuracy by increasing a parameter. If the increase goes beyond a specific limit, then it can result in a worse approximation [55]. Other reported loss of accuracy includes sampling populations separated by reduced gene flow [56] and potential issues associated with ignoring type 2 recombination events (from Marjoram and Wall's classification [51]), which may impact the mean and variance of the most recent common ancestor times when simulating long sequences [57].

Alternative implementations of the exact coalescent also include algorithmic and data structure optimizations, such as *scrm* [54] and *msprime* [55], which do not exhibit all the issues mentioned earlier to *SMC* and *MaCS*.

While SMC (and variants) based applications scale well in terms of simulating genome-scale sequences, they do not scale well in terms of sample size. Therefore, tools implementing the standard coalescent are still being released—for example, *msms* [73] and *msprime* [55].

CHAPTER 3

3 Parallelizing Coalescent Applications

This chapter presents our proposal for parallelizing coalescent applications, which focuses on exploiting coarse-grain parallelism with the manager-worker paradigm and minimizing the communications with a masterless approach.

3.1 Application Characterization

Unveiling computation and I/O hotspots is the first step to determine whether a fine grain or coarse grain parallelization approach makes more sense. In this section, we want to show how computation is distributed among the algorithm steps and the sequential application's output process to determine whether a fine-grained parallelization approach could make sense.

Depending on the target species, a molecular population geneticist may require to use many evolutionary parameters. Each parameter has a potential impact on the temporal, or spatial space, or both. As we already mentioned in previous chapters, it is already known that coalescent applications suffer from large DNA regions with large recombination rates. As suggested by domain experts, we include the mutation rate and initial and the number of chromosomes.

We show the impact of each parameter in Figure 5 and Figure 6.

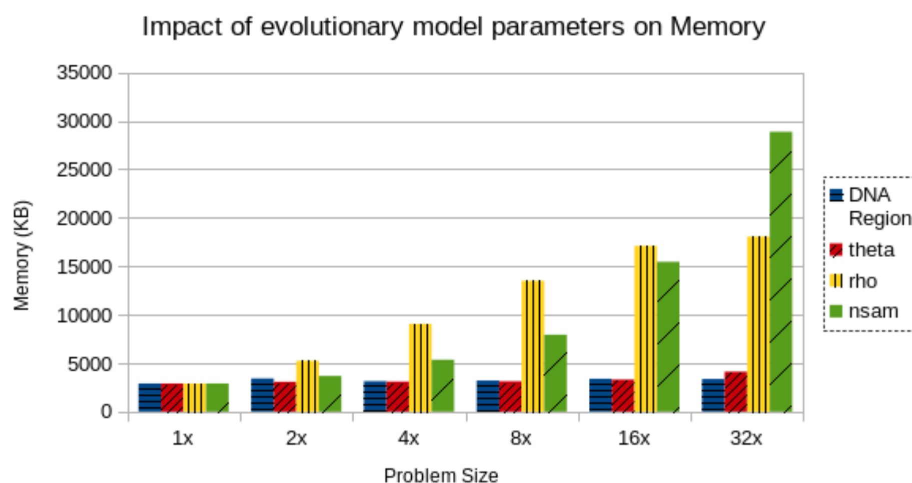


Figure 5. Impact on the spatial space to generate one single replica, changing the DNA region size, mutation rate (*theta*), recombination rate (*rho*), and the number of chromosomes (*nsam*). An initial experiment is taken as a baseline, and then each one of the evolutionary parameters is double in size. We can observe how *nsam* has a more significant impact than other parameters.

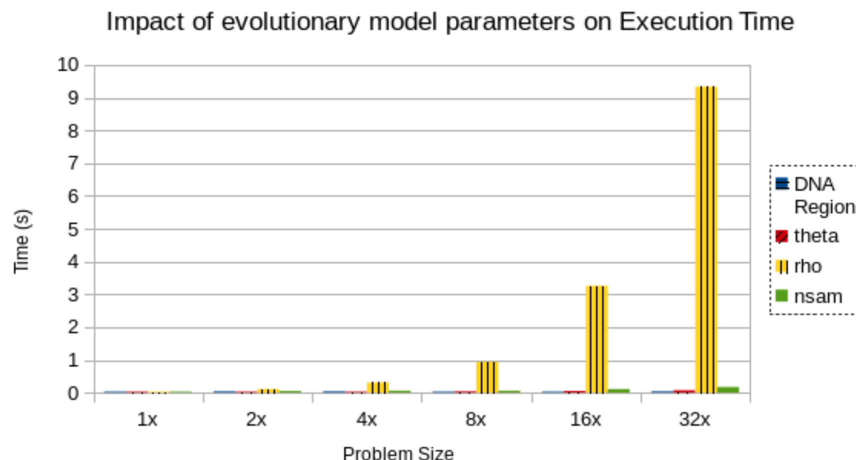


Figure 6. Impact on the temporal space, using the same setup as Figure 5

As there is no biological interest in working with a large number of chromosomes, and given that we know in advance that the target is working at genome-scale, then we proceeded to measure the time and memory consumption for recombination and mutation rates only.

Figure 7 (a) shows how much time Hudson's *ms* application spends in computing and I/O processing to generate the samples, given a fixed mutation rate and a variable recombination rate. In Figure 7 (b), we characterize the memory consumption, but this time we variate both recombination and mutation rates.

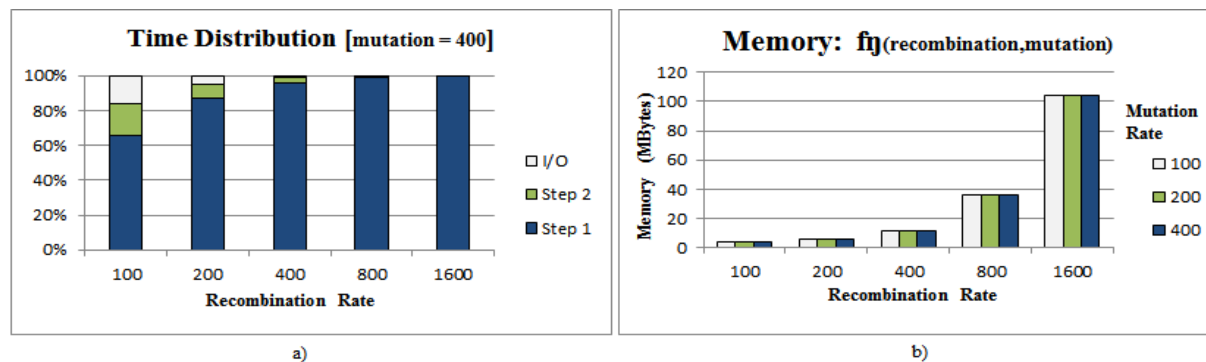


Figure 7. a) Time spent to generate the ARG, distributed into the algorithm's steps, and I/O processing. b) Maximum resident set size as a function of recombination and mutation parameters.

We found the recombination rate is the evolutionary parameter that most significantly impacts both total execution time, and memory consumption. The mutation rate determines how bit the output file will be, translating into more I/O time processing. It becomes negligible when the recombination rate increases over 400.

3.2 Coarse Grain Parallelization

For analysis requiring millions of replica samples, each independently generated, translates into sequentially running steps 1 and 2 from *ms* millions of times. The result of each of these computations does not depend on the results from any other computation. This is an embarrassingly parallel problem [74].

The high degree of data dependency among routines from step 1 (see Figure 2) and between steps 1 and 2 prevents using a fine grain parallelization approach without completely refactoring the source code. We propose to apply the manager-worker pattern to parallelize Hudson's *ms* application, which is a natural fit for Monte Carlo applications [[75], [76]].

Our first approach to parallelize Hudson's *ms* was *msPar* [77]. We show the manager-worker model in Figure 8.

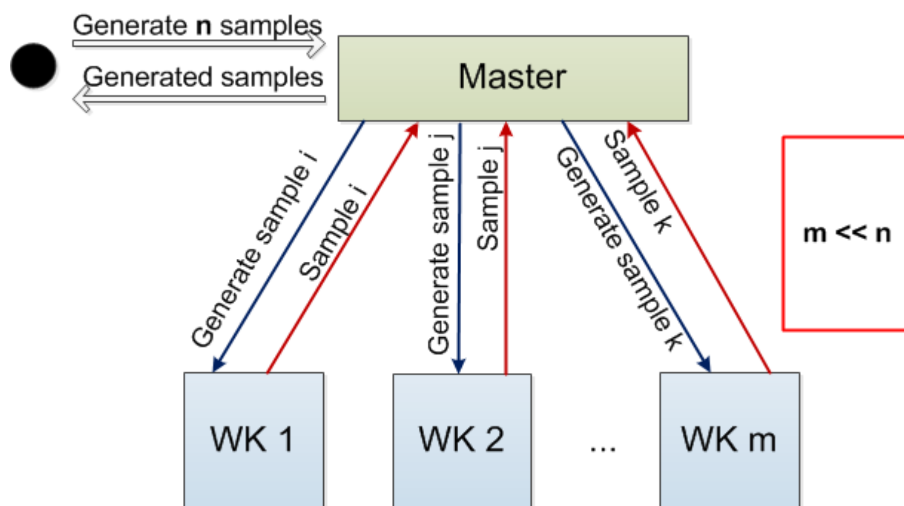


Figure 8. Manager-Worker topology from msPar. The manager process iterates over the available workers, asking them to generate a replica sample. Each worker executes the original Hudson's ms logic and sends back a message with the generated replica. The manager aggregates all the received replicas and generates the final output.

The replica sampling orchestration from Fig. 7 is assigned to the manager with the following algorithm:

```
managerProcessingLogic() {
   initialize and distribute RNG 9seeds to workers
   while there are replicas to generate
      find an idle worker
      if there is an idle worker
         assign replica generation to worker
      else
         retrieve generated replica from workers
   end
}
```

The manager also maintains a pool with idle workers. In the beginning, all workers are idle and the pool is full. When one replica is assigned to one idle worker, the worker is removed from the pool. Moreover, after one worker has generated its assigned replica, it sends a message to the manager, and the manager adds this worker back to the idle pool. When the manager reads the generated replica sent by workers, it determines if more replicas need to be generated and send a signal to the worker to let it know whether to wait for more request, or stop working.

Each worker is waiting for a request of replica generation, and then run both step 1 and 2 to generate one replica and then transmits it back to the master. Following is the pseudo-code for worker's processing:

```
workerProcessingLogic() {
   receive seeds and initialize local RNG
   read evolutionary parameters
   while worker is active {
      run step 1
      run step 2
      send replica to manager
      receive activation signal
   end
}
```

The input arguments and output remains unchanged from the original application, but we include additional arguments to setup the parallelization strategy. If the target environment consists of *m* cores, the parallelization is performed by dividing the N replicas evenly amongst *m-1* cores, and the remaining core plays the role of the manager.

The manager-worker is implemented with the MPI programming model. We spawn worker processes and map them to hardware processors in the system, each one using its local memory. We specifically use the Open MPI parallel library [32].

[9] Random Number Generator

The manager-worker's quality of random number streams is guaranteed by using the RNG (Random Number Generator) twice. The RNG seeds specified as input parameters are used to initialize the manager's RNG. Then the manager generates a set of random numbers ultimately used by each worker as seeds to initialize their RNG.

3.3 Communication Patterns and Memory Management

Modeling complex organisms with large recombination rates translates into an exponential increase in memory consumption. If we work with large genomic regions, then the situation gets even worse.

When generating several millions of replicas, for example, in the case of feeding ABC experiments, in addition to the time spent on generating the replicas, the higher number of messages flowing through the manager-worker can limit the scalability of our parallel approach.

We modified the initial manager-worker implementation, improved memory management at the MPI process level, and improved communication patterns to ease communication overhead. The resulting implementation is *msParSm* [78].

We use the Cross Memory Attach (CMA) mechanism to improve the intra-node communications [79]. We use the OpenMPI's *vader* BTL [80], whose design is influenced by the single-copy RDMA-like capabilities provided by XPMEM, implementing both SEND and RDMA transfer protocols. *Vader* BTL allows MPI processes to directly read/write data to a different process's virtual memory space without passing through the kernel space. Thereby, latency for exchanging messages between workers and managers co-located on the same node could be lowered since one copy operation can be saved. This mechanism contrasts with the other one used by *msPar*, known as *sm* BTL (shared-memory BTL). The *sm* BTL mechanism follows a copy-in/copy-out pattern: when an MPI process X sends a mes sage to a process Y, the message is first copied from the X's buffer to the shared memory, and then the receiver (i.e., the process Y) copies the message from the shared memory into a buffer.

To take further advantage, we have changed the manager-worker topology used by *msPar*. In *msParSm*, we have one manager process per compute node (hereafter referred to as node manager) and one global manager process that coordinates the M node managers (where M is the number of compute nodes). Each manager node coordinates the co-located worker processes, returning an aggregated message with all of the generated samples back to the global manager. It is important to note that there will be one node hosting two manager processes: the global manager and the node manager. We show a simplified version of the manager-worker topology in Figure 9.

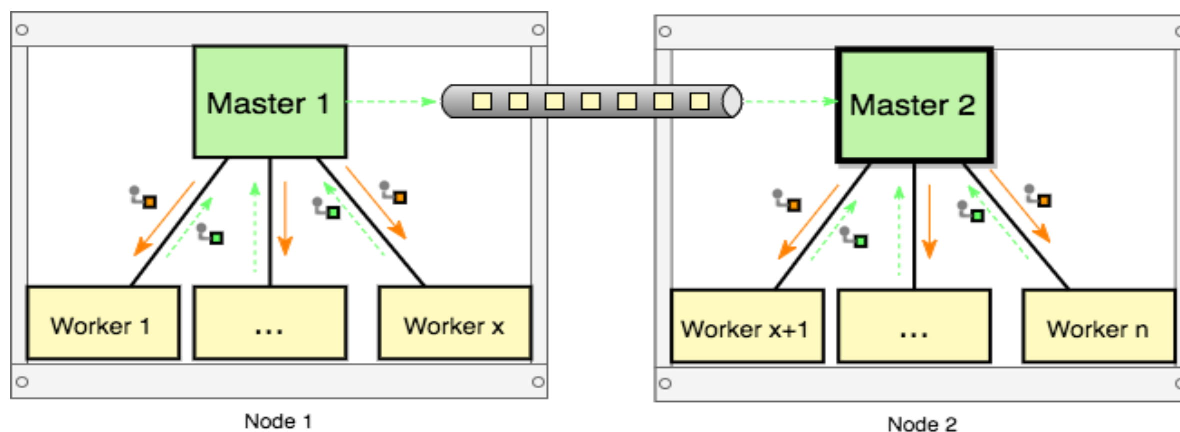


Figure 9. Manager-Worker topology from *msParSm*.

Using non-blocking MPI collectives, we also enabled to generate while waiting for local worker communications. Thereby, the potential loss of processing power compared to *msPar* is compensated as much as possible. In a setup with M compute nodes and P cores per node, in *msPar*, we have a total number of M*P – 1 dedicated worker processes, while in *msParSm,* the number is M*(P – 1) – 1.

For the use case of geneticists using a single node (e.g., a single fat node or even a workstation), the application will not use a global manager, but a single manager that is also going to generate replica samples.

Another improvement we made is allowing workers co-located with the global master to directly output the generated samples, thus saving MPI point-to-point communications with the node manager.

3.4 Approaching Multilocus Analysis with the Hierarchical Manager-Worker

Multilocus data cannot be easily analyzed using Hudson's *ms* simulator, forcing geneticists to use a different application. We analyzed the *mlcoalsim* application [46] because it is based on Hudson's *ms*. Therefore, we continue working on software that is accepted by academia.

The *mlcoalsim* simulator is a modification of Hudson's *ms* that uses MPI to parallelize the generation of replicas per *locus*. It follows the same approach as Hudson's *ms* to regarding memory structures. In Hudson's *ms*, one ARG is generated for each generated genealogy, while *mlcoalsim* generates as many ARGs as *loci* per genealogy, optionally in parallel. The ARG is maintained into a global shared variable updated by all the routine steps, exhibiting the same dependency issue as Hudon's *ms*.

While Hudson's *ms* compute is structured as a two-level nested loop, *mlcoalsim* is structured as a three-level nested loop. The first level loop iterates over *loci,* and the inner two-level loop encapsulate Hudson's *ms* logic. Current *mlcoalsim*'s parallelization consists of implementing

a simple manager-worker. It spawns as many MPI processes as *loci* required per genealogy (i.e., the first level loop). The evolutionary model passed as an input file is parsed and put into a memory structure, which is broadcasted to all the involved MPI processes. Each MPI process runs the full Hudson's *ms* logic (i.e., the inner two-level loop), and the generated genealogy is sent to a manager process using MPI blocking point-to-point communications. Finally, the resulting information is aggregated, and the manager process generates several output files.

The number of loci per genealogy limits the maximum degree of parallelization of mlcoalsim. It cannot use more MPI processes than the number of *loci*. For example, if we have a compute node with 48 cores, the application will not run if we specify to spawn 48 MPI process, but the loci number is 10. We are forced to indicate using 10 cores only, leading all the other available cores to remain idle. Consequently, the efficiency of *mlcoalsim* can significantly degrade.

In Figure 10, we show how *mlcoalsim* is parallelized following the same approach as *msParSm*.

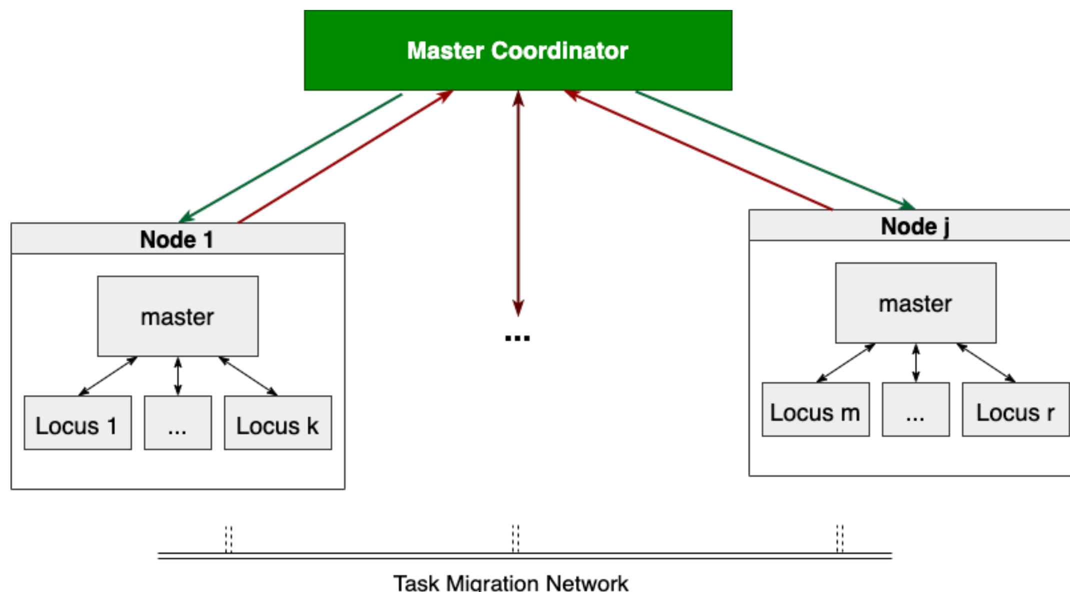


Figure 10. *mlcoalsim* is parallelized using a hierarchical manager-worker model. It allows to run simulations on several nodes, and also to work with a larger number of *loci* than available cores.

The higher manager coordinator allows us to run experiments on an HPC cluster and approach larger problems. The manager node lifts the limitation of using a larger number of cores than specified *loci* per genealogy replica. We show more details about how to run this new version in Appendix B.

The parallelization of *mlcoalsim* propels the use of an HPC cluster to generate multiple replicas in parallel, but it also introduces new questions. Preventing load imbalance implies having HPC nodes generating genealogies from different replicas and *loci*. Processing time and memory consumption for each genealogy are not necessarily homogenous. Determining

the right chunk size might require addressing dynamic policies and get feedback about the underlying cluster usage to maximize its utilization. Finding the right trade-off between scheduling overhead and load imbalance might be challenging.

Our parallelization of Hudson's *ms* has used a homogeneous HPC cluster, and we specifically made requests to get dedicated nodes while running our experiments. In the case of *mlcoalsim*, targeting a heterogenous cluster, using a non-dedicated node assignation policy, or both, might require contemplating using a weighting algorithm. A potential complementary approach could be starting with an initial chunk size and reduce it throughout the execution. An extensive review of the self-scheduling literature is required: [81], [82], [83], [84], [85], [86].

CHAPTER 4

In this chapter, we evaluate and discuss the different parallelization approaches presented in chapter 3.

4 Experimental Results

4.1 Coarse-Grain Approach

In this section, we evaluate the coarse grain parallelization using the manager-worker paradigm. We show that *msPar* can achieve significant speedup values and much better execution times on an HPC cluster than Hudson's *ms* version.

4.1.1 Experimental Setup

We evaluate the coarse grain manager-worker implementation's performance by executing *msPar* on an HPC cluster, allocating up to three nodes in exclusive mode. Each node is configured with two Intel Xeon X5660. The X5660 is a six-core processor with Hyper-Threading Technology, running at 2.80 GHz, 12MB L3 cache, and 96 GB of dynamic random access memory (DDR-RAM). This configuration gives us 12 physical processors per compute node, but we can get 24 logical processors per node due to the Hyper-Threading technology.

We designed a suite of test cases to be simple enough (i.e., without population structure), considering only recombination, mutation, genetic region, and sample size, focusing our attention on the performance evolution when the recombination ratio changes. We decided to use a scaled[10] mutation rate of 640, which being not too high; it is big enough to let the mutation assignment step to get some computation.

We selected a sample size of 200 chromosomes and a region of 10e6 bp (base pairs), considered by geneticists as big for genetic analysis and quite close to what is required in the multilocus genomic analysis [53].

We use Hudson's *ms* as a *baseline* application that serves as a basis for comparison. We run Hudson's *ms* in one of the nodes, with exclusive mode allocation enabled.

Suppose evolutionary parameters do not change from one replica to another (as in this setup). In that case, differences in the execution time and memory consumption for each replica

---

[10]. All neutral parameters are given by per-site rates 4N, where N is the current population size.

generation will be negligible. For the recombination rate, we decided to start with a scaled value of 2560 because, combined with the other parameters, it makes *ms* consume enough memory (1.3 GB). We doubled it until we reached 10240, which is considered a considerable recombination rate.

The number of generated replicas was set in 528 for two reasons: first, we wanted ms to execute during enough time (2.75 hours in the minimal setup) for measuring. Second, it allows our manager-worker application to distribute work fairly well in the case of 72 workers.

4.1.2 Discussion

In this section, we present the performance metrics obtained during the experimentation. We investigate the speedup and efficiency as a function of increasing numbers of processors.

We oversubscribed the MPI processes per node from 12 to 24. This directly impacts the efficiency numbers, as it will be taken into account the 24 logical processors due to the Hyper-Threading technology, instead of the 12 physical processors in each compute node.

In Figure 11 (a), we show the normalized speedup of *msPar* compared to the sequential version.

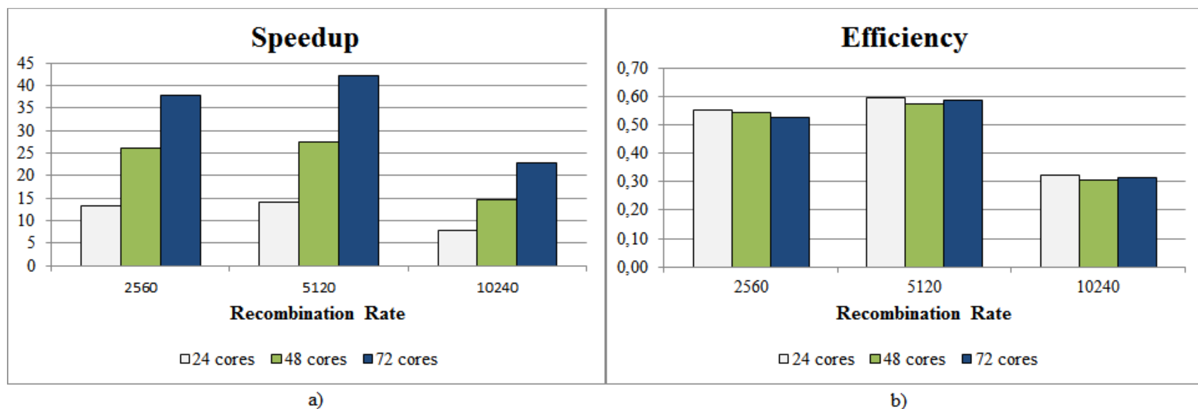


Figure 11. Speedup and efficiency of *msPar* compared against Hudson's *ms*. a) Speedup obtained with 24 cores (1 node), 48 cores (2 nodes), and 72 cores (3 nodes). b) Efficiency showing how well the parallelization goes as long as more cores are used for computation.

We observe degradation in performance for the case of maximal recombination rate. As we can observe in Figure 12, each MPI process consumes ~16 GB to generate a genealogy with a maximum recombination rate (i.e., 10240). Therefore, having 24 MPI processes running with this problem size implies exhausting the physical node memory (~380 GB consumed vs. 96 GB available). A performance penalty because of page swapping can explain the performance degradation. Another explanation for the performance degradation can be attributed to the increase of inter-node communications.

In Figure 11 (b), we show the efficiency obtained as long as more cores are used for computation. We observe a significant drop-off when generating genealogies with maximum recombination rate, which is aligned with our previous observation regarding the speedup.

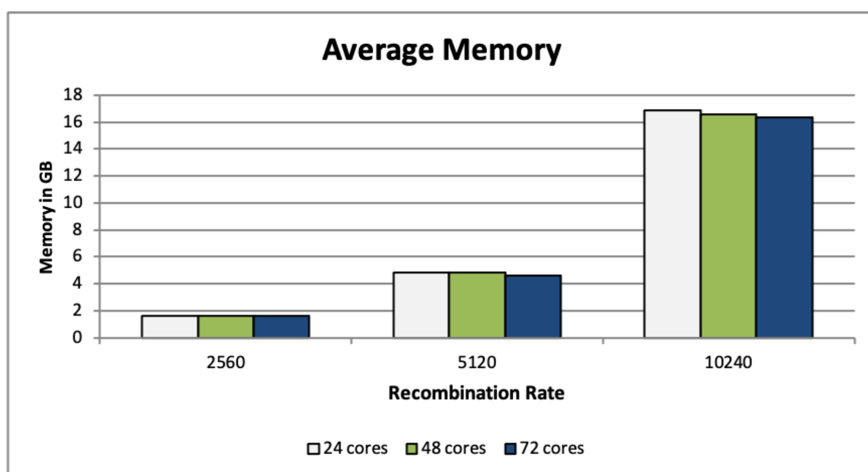


Figure 12. Average resident set size consumed by each MPI process when running the experiments with *mspar*.

The most important contribution for the geneticist is the reduction of the overall running time. Table 1 shows the overall running time *ms* and *msPar* take to complete different problem sizes. The first column of the table is the recombination rate used (i.e., the problem size). The second column indicates the execution time of *ms*. The next three columns show the execution time of *msPar*, using 72, 48, and 24 cores.

Table 1. Execution times (in 'd' days, 'h' hours, and 'm' minutes) for both *ms* and *msPar*.

			msPar	
Recombination	ms	72	48	24
2560	2h45m	4m	6m	12m
5120	24h40m	35m	53m	1h44m
10240	7d18h5m	8h11m	12h36m	24h2m

From this table, we see that a significant improvement is achieved regarding the time a researcher should wait for getting results.

4.2 Hierarchical Manager-Worker

This section shows the performance and scalability gains resulting when using a hierarchical manager-worker and taking advantage of the vader BTL mechanism.

We compare our solution to the fastest application that implements the sequential Markov coalescent algorithm, MaCS [53]. This application approximates the calculation of recombination. It achieves execution times in orders of magnitude lower than Hudson's *ms,* especially with large recombination rates. Additionally, *MaCS* can work with recombination rates larger than Hudson's *ms*.

4.2.1 Experimental Setup

Evaluation is done on an HPC cluster, where we allocate up to eight nodes in exclusive mode. Each node is configured with two Intel Xeon X5660. The X5660 is a six-core processor with Hyper-Threading Technology, running at 2.80 GHz, 12MB L3 cache, and 96 GB of DDR-RAM. This configuration gives us 12 physical processors per compute node, making a maximum compute power of 96 cores (we did not enable Hyper-Threading in this setup).

We compare *msParSm* with Hudson's *ms* and *MaCS* in terms of execution time, using two case studies configured with fixed population mutation rate values (i.e., $\theta = 4N\mu$, where N is the current population size and μ is the mutation rate per site). The first case approximates the estimated human average variability per nucleotide. In contrast, the second case matches the estimated D. *melanogaster* average ($\theta = 0.001$ and 0.005 values, meaning that 1 and 5 are differences between two random individuals for every 1000 sites, respectively).

The number of generated replicas is set at 300, each one with 100 chromosomes (sample size). The genetic region contains 10e6 bp, and population recombination per site ($4N\rho$, where ρ is the mutation rate between two contiguous sites) ranges from 0.0005 to 0.08. Both evolutionary models exhibit no population genetic structure.

4.2.2 Discussion

We evaluated the performance of *msParSm* by analyzing the speedup and efficiency as a function of increasing numbers of compute nodes.

In Figure 13, the speedup of *msParSm* using Hudson's *ms* as a baseline is shown, which was registered in case study 1. We observe a high parallel efficiency of *msParSm* and a substantial increase in speed concerning sequential Hudson's *ms* version. Nevertheless, for the case of maximal recombination (Figure 13 (d)), we observe a slight performance degradation. If we look at Figure 14 (d), then we observe the node memory was mostly exhausted by the MPI

processes (90 GB in use out of 96 GB available). It does suggest the performance degradation is the result of page swapping.

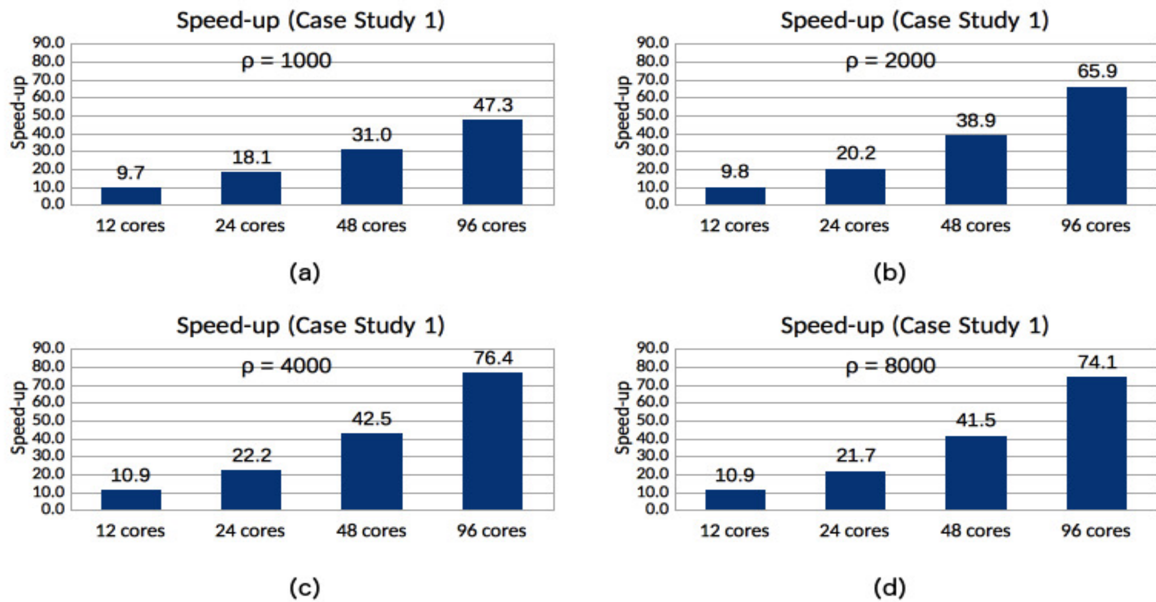


Figure 13. Speedup analysis of case study 1 using Hudson's *ms* as a baseline. The x-axis counts the total number of physical cores per allocated node (i.e., 12 cores for 1 node, 24 cores for 2 nodes, etc.). Different recombination rates are registered in subfigures a) to d), starting with 4Nρ = 1000 in (a), and doubling it through the remaining subfigures (c to d) until reaching scaled recombination of 4Nρ = 8000.

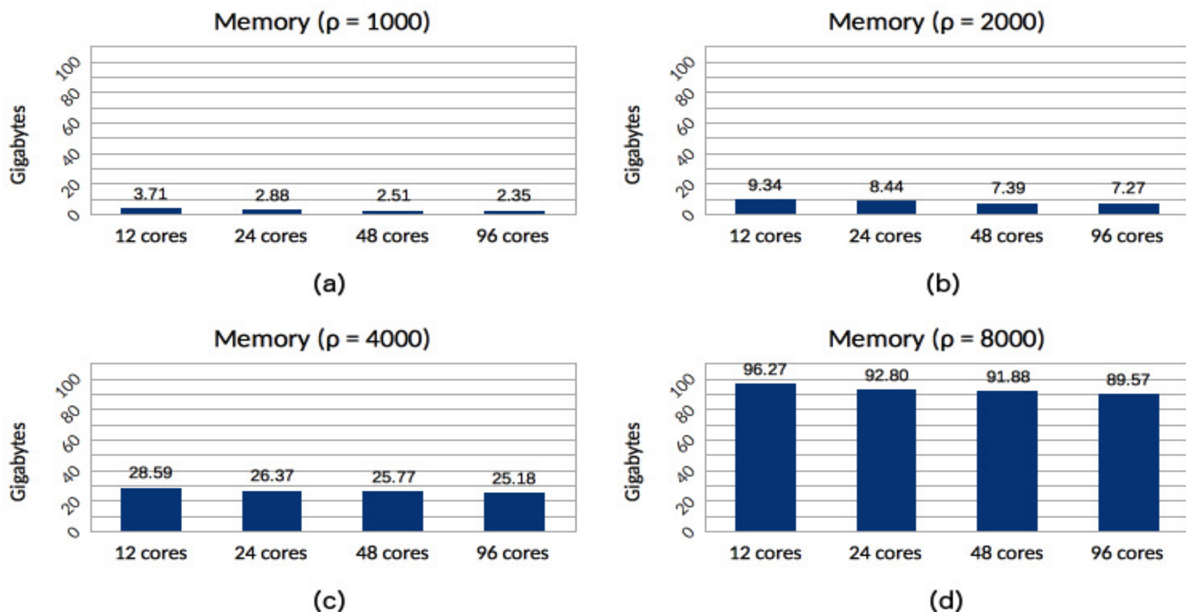


Figure 14. Obtained speedup for case study 1. Grouping is the same as Figure 13.

In Figure 15, we see a similar behavior as for case study 1. We observe a worse performance for a lower recombination rate (Figure 15 (a)), but performance for other recombination rates

are similar. An explanation for this behavior is associated with the message payload exchanged between MPI processes. The average message with sample data is 4.5 times higher for case study 2, suggesting that the time spent in communication is less relevant as long as the recombination rate increases.

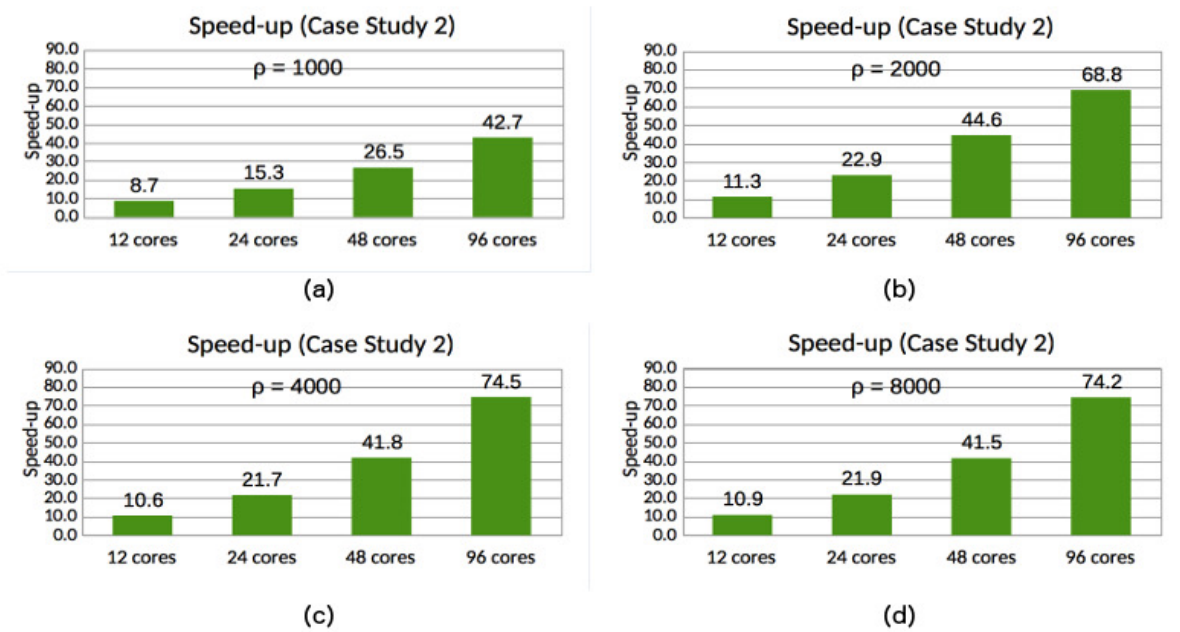


Figure 15. Obtained speedup for case study 2. Grouping is the same as Figure 13.

In Figure 16, we can observe how the node's main memory is exhausted when the maximal recombination rate is reached (Figure 16 (d)), regardless of the number of involved cores, explaining why the efficiency is impacted.

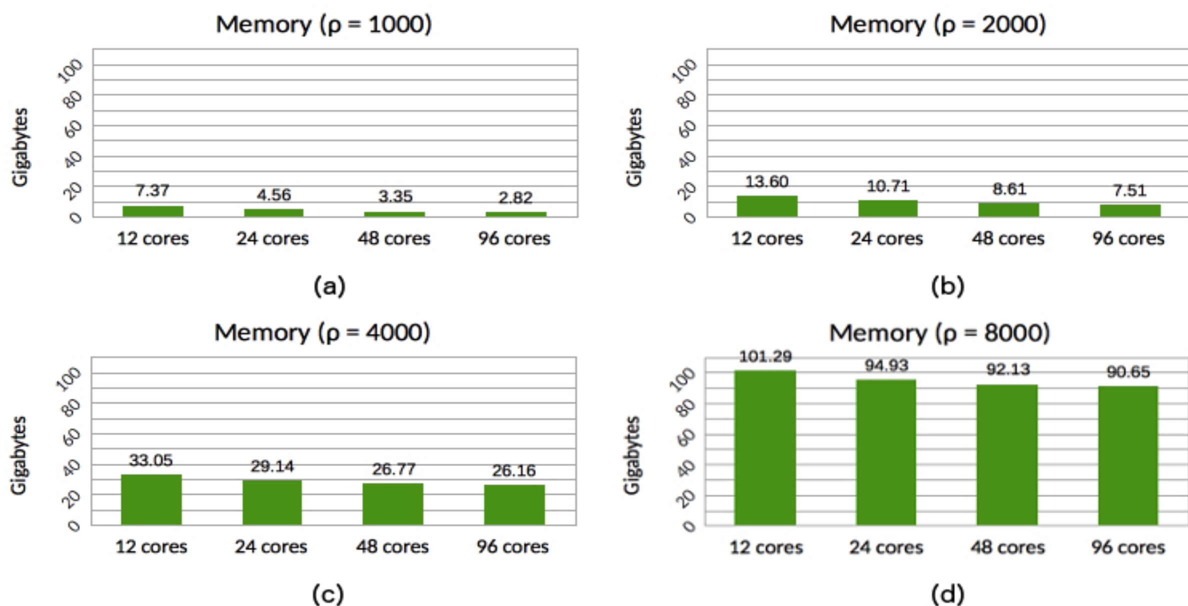


Figure 16. Average consumed memory for case study 2, following the same setup as in Figure 15

Table 2 shows the average time Hudson's ms, MaCS, msPar, and msParSm spent running both case studies using different problem sizes (i.e., recombination rate values) is given. We observe how the execution time of msParSm is orders of magnitude faster than Hudson's ms, faster by a factor than msPar, and significantly better than MaCS in most configurations, mainly when the mutation rate is higher.

Table 2. Comparison of average time cost (in minutes) between *ms*, *MaCS*, *msPar*, and *msParSm*. The execution times of both *msParSm* and *msPar* are grouped in function of the number of cores used for computation. A combination of font styles and the background color is used to facilitate the reading: values in italics are related to *msPar*, while bold style is used for *msParSm*; shaded table cells indicate cases that the execution time of both *msParSm* and *msPar* are worse than *MaCS*.

Rho	ms	MaCs	msParSm / msPar			
			96 cores	48 cores	24 cores	12 cores
Case study 1						
1000	3.81	3.82	**0.08** / *0.44*	**0.12** / *0.44*	**0.21** / *0.64*	**0.40** / *0.59*
2000	28.77	7.16	**0.45** / *2,75*	**0.76** / *2.75*	**1.47** / *3.91*	**3.05** / *3.42*
4000	322.80	13.76	**4.23** / *19.57*	**7.60** / *26.07*	**14.57** / *28.48*	**29.64** / *30.52*
8000	2605.58	26.86	**35.15** / *149.48*	**62.78** / *214.36*	**120.02** / *223.98*	**240.12** / *249.5*
Case study 2						
1000	3.95	5.80	**0.22** / *1.14*	**0.21** / *1.22*	**0.26** / *1.61*	**0.44** / *1.37*
2000	34.85	9.14	**0.51** / *3.33*	**0.78** / *3.53*	**1.52** / *4.89*	**3.09** / *4.23*
4000	320.30	15.88	**4.30** / *20.02*	**7.67** / *27.61*	**14.78** / *29.43*	**30.12** / *32.03*
8000	2602.00	29.32	**35.05** / *151.35*	**62.75** / *222.60*	**119.03** / *224.18*	**239.47** / *249.27*

Although the parallel execution of msParSm is behind MaCS in the case of large recombination rates, the scalability of the presented msParSm is superior to msPar, as the number of computing resources increases, with the possible limitation of the available memory on each node.

4.3 Multilocus Applications

This section shows the performance and scalability gains resulting when applying the hierarchical manager-worker from section 3.3 to a coalescent application intended for multilocus analysis. We show that the same strategy used in *msParSm* can be applied to the *mlcoalsim* simulator.

4.3.1 Experimental Setup

Evaluation is done as a microbenchmark in a single computer, configured with an Intel i7-9750H six-core processor, running at 2.60GHz, 12MB L3 cache, and 32 GB of DD-RAM.

We compare the parallel version with *mlcoalsim* in terms of execution time. The case study simulates five loci from an autosome and five loci from X-chromosome in a mammal species. The population analyzed suffered a bottleneck defined in prior 1 after a time defined in prior 2. The number of generated replicas is set at 10e6, each one with 100 chromosomes.

4.3.2 Discussion

We evaluated the performance of *mlcoalsim* by analyzing the speedup as a function of increasing numbers of cores. This application is already capable of running in parallel when generating multilocus replicas. We applied the same hierarchical manager-worker approach from *msParSm* to *mlcoalsim*. This new version focuses on minimizing the intra-node communications but also to leverage an HPC cluster for parallelizing the generation of genealogies (*mlcoalsim* parallelization is at loci-level)

In Figure 17, we observe that *mlcoalsim* struggles in effectively using all the available computing power when enabling MPI processing. In contrast, the enhanced parallel version achieves better speedups as more cores are used.

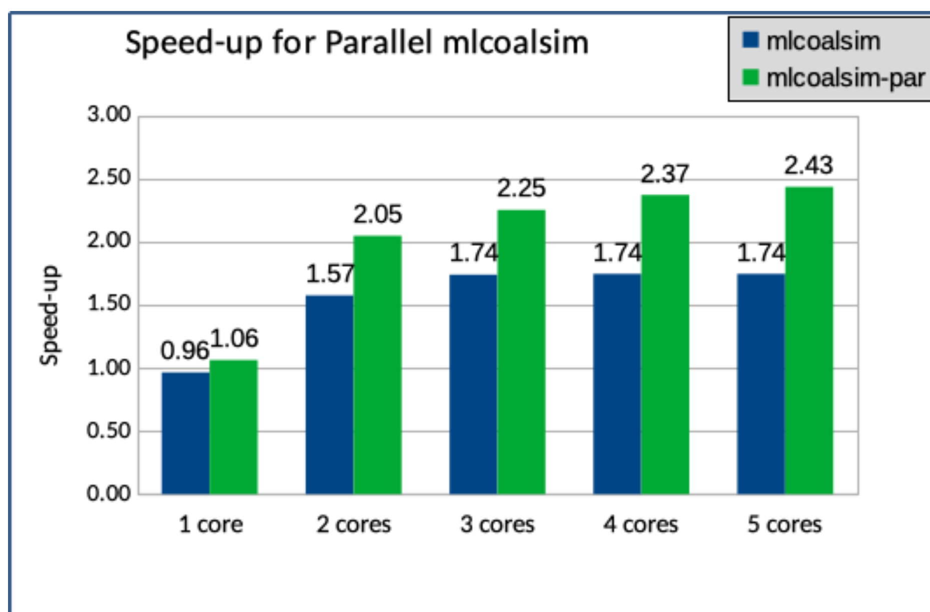


Figure 17. Speedup analysis of *enhanced parallelization* when compared against the original *mlcoalsim*. Blue bars correspond to the speedup of *mlcoalsim* when enabling processing with MPI. Green bars correspond to the speedup obtained by parallelizing *mlcoalsim* with the approach as with *msParSm*.

Integrating the hierarchical manager-worker into *mlcoalsim* is more straightforward than in the case of Hudson's *ms*. Its poor scalability can be attributed to sub-optimal use of the Open MPI library and memory management. Our parallel version includes several improvements regarding pointers (especially char arrays), and it also uses the *vader* BTL for one-sided communications.

This application reflects our initial rationale regarding how applications are usually parallelized in the population genetics field: *mlcoalsim* includes a parallelization approach but is not designed to use all the power of an HPC cluster. The utilization of many nodes would require splitting the data, run several application instances, and finally aggregate the results. Additionally, it has the limitation that the number of *loci* cannot be bigger than the number of available cores. Therefore, even splitting the data into chunks, using a heterogenous cluster, could be challenging if one of the allocated nodes has fewer cores than the *loci* being modeled.

CHAPTER 5

5 Conclusions and Future Work

The increasing availability of whole-genome data provides an unprecedented opportunity to understand species' history and geographic structure. Molecular population geneticists have been striving to design alternative algorithms to the coalescent and generate synthetic data in a viable time-frame. Such alternatives, which enable to experiment at genome-scale, use approximations that suffer from loss of accuracy. Additionally, these applications fail to effectively work on HPC clusters, mainly because of the inability to create parallel software, either explicitly by a developer or automatically by a compiler. Parallel programs are hard to develop, debug, and maintain. Thus, the adoption of parallel computing in the population genetics field has not been mainstream.

This thesis represents a step towards parallelizing coalescent applications that benefit from coarse grain parallelism. In this thesis, we have designed a manager-worker framework that extracts parallel performance from the de facto exact coalescent implementation, Hudson's *ms*. We have shown that parallelizing the application using the manager-worker paradigm can consistently obtain parallel speedups without compromising the original application's accuracy and flexibility. This thesis makes the following contributions:

- We have shown that the manager-worker paradigm effectively parallelizes coalescent applications, especially when each chunk's computation is large enough to hinder the communication overhead. Our initial application, *msPar*, allows the geneticists to work with moderated complex evolutionary models in an HPC cluster, getting the same accuracy from the exact coalescent process.
- We have implemented a second version, *msParSm*, that uses a hierarchical manager-worker and optimizations for intra-node communication. This approach achieves high parallel efficiency figures (>70%) when working with large recombination rates. Using this application, we can outperform *MaCS* in most cases, even when running the application in one single node.
- We showed that parallelizing a coalescent application can be done without modifying the core logic. This approach applies to applications that first generate genealogies and then throw down mutations in a separate process. However, the method may also apply to applications that approximate the coalescent by merging these two processes, such as *cosi2* [87].
- We have applied the same parallelization technique to a different coalescent simulator, *mlcoalsim*. In essence, the minimum effort needed to be done is separating the logic

into compartments, apply a manager-worker model, and iterate with logic inside the worker.

-

5.1 Future Work

This thesis gives rise to some open lines and future work:

- Explore load balance on heterogeneous HPC clusters. Reasons for the limited scalability of the presented approaches can be associated with communication overhead, synchronization loss, false sharing, NUMA locality, bandwidth bottlenecks, etc. While all these are potential reasons for poor scalability, load imbalance may have a more profound impact, especially for generating replicas with *priors* data, and running experiments on heterogeneous clusters. The performance issues associated can be mitigated by developing a solution for efficiently distributing the work based on the underlying HPC system's heterogeneity.
- Try using a fully distributed (i.e., master-less) approach for parallelization. This approach consists of having no manager process and all processing elements assuming the role of workers. A decentralized system could potentially provide more accurate dynamic scheduling and better load balancing.
- Explore the parallelization with a hybrid programming approach. Given the increasing availability of multicore and manycore architectures in HPC clusters, it is more than doubtful if running one MPI process per cores is appropriate to exploit parallelism. The hybrid parallel programming strategy often combines MPI with OpenMP, where MPI is used for internode communication, and OpenMP for parallelization within the node. An interesting line of study is determining if using the MPI-3 shared memory programming model is better than the MPI/OpenMP approach, especially in manycore architectures.
- Test the parallelization strategy with *multilocus* applications. Multilocus applications provide more parallelization opportunities than the standard coalescent (e.g., *mlcoalsim*, an extension to Hudson's *ms)*. A potential line of investigation could be designing a hierarchical manager-worker model that combines the aforementioned master-less approach with hybrid programming, and potentially load balancing strategies when using a heterogeneous HPC cluster.
- Explore parallelizing *forward-in-time* applications, which can be used for understanding the effects of selection. Contrasting to coalescent applications used to study a population sample, *forward-in-time* applications focus on studying the whole

population. Fine-grain parallelization might be a better approach to parallelize *forward-in-time* applications.

5.2 List of Publications

The work presented in this thesis has reported the following publications:

- **C. Montemuiño, A. Espinosa, J.-C. Moure, G. Vera-Rodríguez, S. Ramos-Onsins, and P. H. Budé, "msPar: A Parallel Coalescent Simulator," in Euro-Par 2013: Parallel Processing Workshops, D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, Eds. Springer Berlin Heidelberg, 2013, pp. 321–330.**
  This work focuses on parallelizing the de facto coalescent application using a coarse grain approach, implementing the manager-worker model. We showed that it is possible to run the application on an HPC cluster, and obtain acceptable speedups when working with long genomic regions.

- **C. Montemuiño, A. Espinosa, J. C. Moure, G. Vera, P. Hernández, and S. Ramos-Onsins, "Approaching Long Genomic Regions and Large Recombination Rates with msParSm as an Alternative to MaCS," Evol Bioinform Online, vol. 12, pp. 223–228, Oct. 2016, doi: 10.4137/EBO.S40268.**
  This work proposes the use of a hierarchical manager-worker to overcome the scalability issues from *mspar*. This paper shows the performance gains obtained from the manager-worker approach, and the optimizations to lower the intra-node communication overhead.

6 Bibliography

[1] K. Wetterstrand, “DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP),” *Genome.gov*, 2020. www.genome.gov/sequencingcostsdata (accessed Dec. 10, 2020).

[2] S. Hoban, G. Bertorelle, and O. E. Gaggiotti, “Computer simulations: tools for population and evolutionary genetics,” *Nat. Rev. Genet.*, vol. 13, no. 2, pp. 110–122, Feb. 2011, doi: 10.1038/nrg3130.

[3] J. Sanford, J. Baumgardner, W. Brewer, P. Gibson, and W. ReMine, “Mendel’s Accountant: a biologically realistic forward-time population genetics program,” *Scalable Comput. Pract. Exp.*, vol. 8, no. 2, Jan. 2001, doi: 10.12694/scpe.v8i2.407.

[4] J. Sanford and C. Nelson, “The Next Step in Understanding Population Dynamics: Comprehensive Numerical Simulation,” in *Studies in Population Genetics*, M. C. Fust, Ed. InTech, 2012.

[5] Y. Kim and T. Wiehe, “Simulation of DNA sequence evolution under models of recent directional selection,” *Brief. Bioinform.*, vol. 10, no. 1, pp. 84–96, Jan. 2009, doi: 10.1093/bib/bbn048.

[6] R. R. Hudson, “Generating samples under a Wright-Fisher neutral model of genetic variation,” *Bioinforma. Oxf. Engl.*, vol. 18, no. 2, pp. 337–338, Feb. 2002.

[7] R. Hudson, “Gene genealogies and the coalescent process,” *Oxf. Surv. Evol. Biol.*, vol. 7, pp. 1–44, 1991.

[8] A. Carvajal-Rodríguez, “Simulation of Genes and Genomes Forward in Time,” *Curr. Genomics*, vol. 11, no. 1, pp. 58–61, Mar. 2010, doi: 10.2174/138920210790218007.

[9] T. Yang, H.-W. Deng, and T. Niu, “Critical assessment of coalescent simulators in modeling recombination hotspots in genomic sequences,” *BMC Bioinformatics*, vol. 15, p. 3, Jan. 2014, doi: 10.1186/1471-2105-15-3.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th edition. Morgan Kaufmann, 2017.

[11] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[12] M. Flynn, “Very High-Speed Computing Systems,” *Proc. IEEE*, vol. 54, pp. 1901–1909, Jan. 1967, doi: 10.1109/PROC.1966.5273.

[13] S.S.Jadhav, *Advanced Computer Architecture and Computing*. Technical Publications, 2009.

[14] F. Gebali, *Algorithms and Parallel Computing*. John Wiley & Sons, 2011.

[15] R. Rahman, *Intel Xeon Phi coprocessor architecture and tools: the guide for application developers*. Springer Nature, 2013.

[16] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: the Terasys massively parallel PIM array," *Computer*, vol. 28, no. 4, pp. 23–31, Apr. 1995, doi: 10.1109/2.375174.

[17] R. Duncan, "A survey of parallel computer architectures," *Computer*, vol. 23, no. 2, pp. 5–16, Feb. 1990, doi: 10.1109/2.44900.

[18] S. Ramos and T. Hoefler, "Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 297–306, doi: 10.1109/IPDPS.2017.30.

[19] S. M. Tam *et al.*, "SkyLake-SP: A 14nm 28-Core xeon® processor," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, Feb. 2018, pp. 34–36, doi: 10.1109/ISSCC.2018.8310170.

[20] Wm. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995, doi: 10.1145/216585.216588.

[21] D. Patterson *et al.*, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar. 1997, doi: 10.1109/40.592312.

[22] J. Suh, C. Li, S. P. Crago, and R. Parker, "A PIM-based multiprocessor system," in *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, Apr. 2001, p. 6 pp.-, doi: 10.1109/IPDPS.2001.924932.

[23] M. Naylor and C. Runciman, "The Reduceron: Widening the von Neumann Bottleneck for Graph Reduction Using an FPGA," in *Implementation and Application of Functional Languages*, Berlin, Heidelberg, 2008, pp. 129–146, doi: 10.1007/978-3-540-85373-2_8.

[24] A. Kagi, J. R. Goodman, and D. Burger, "Memory Bandwidth Limitations of Future Microprocessors," in *23rd Annual International Symposium on Computer Architecture (ISCA'96)*, May 1996, pp. 78–78, doi: 10.1109/ISCA.1996.10002.

[25] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[26] M. L. Scott and W. J. Bolosky, "USENIX SEDMS IV, 1993," *Proc. USENIX Symp. Exp. Distrib. Multiprocessor Syst.*, vol. 57, p. 41, 1993.

[27] "Avoiding and Identifying False Sharing Among Threads," *Intel*, Nov. 02, 2011. https://www.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html (accessed Dec. 21, 2020).

[28] T. Liu and E. D. Berger, "SHERIFF: precise detection and automatic mitigation of false sharing," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, New York, NY, USA, Oct. 2011, pp. 3–18, doi: 10.1145/2048066.2048070.

[29] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe, "Dynamic cache contention detection in multi-threaded applications," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, New York, NY, USA, Mar. 2011, pp. 27–38, doi: 10.1145/1952682.1952688.

[30] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.

[31] P. Pacheco, *An Introduction to Parallel Programming*. Elsevier, 2011.

[32] E. Gabriel *et al.*, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Berlin, Heidelberg, 2004, pp. 97–104, doi: 10.1007/978-3-540-30218-6_19.

[33] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, 1996.

[34] J. Dongarra, "Report on the Fujitsu Fugaku system," University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06, 2020.

[35] H.-W. Loidl *et al.*, "Comparing Parallel Functional Languages: Programming and Performance," *High.-Order Symb. Comput.*, vol. 16, no. 3, pp. 203–251, Sep. 2003, doi: 10.1023/A:1025641323400.

[36] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, New York, NY, USA, Apr. 1967, pp. 483–485, doi: 10.1145/1465482.1465560.

[37] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988, doi: 10.1145/42411.42415.

[38] Y. Shi, “Reevaluating Amdahl’s law and Gustafson’s law,” *Comput. Sci. Dep. Temple Univ. MS 38-24*, Nov. 1996.

[39] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O’Reilly Media, Inc., 2017.

[40] A. Supalov, A. Semin, C. Dahnken, and M. Klemm, *Optimizing HPC Applications with Intel Cluster Tools: Hunting Petaflops*. Apress, 2014.

[41] S. Pllana and F. Xhafa, *Programming Multicore and Many-core Computing Systems*. John Wiley & Sons, 2017.

[42] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Enhanced: The Hardware/Software Interface*. Morgan Kaufmann, 2014.

[43] M. Currat, N. Ray, and L. Excoffier, “splatche: a program to simulate genetic diversity taking into account environmental heterogeneity,” *Mol. Ecol. Notes*, vol. 4, no. 1, pp. 139–142, 2004, doi: https://doi.org/10.1046/j.1471-8286.2003.00582.x.

[44] C. C. A. Spencer and G. Coop, “SelSim: a program to simulate population genetic data with natural selection and recombination,” *Bioinformatics*, vol. 20, no. 18, pp. 3673–3675, Dec. 2004, doi: 10.1093/bioinformatics/bth417.

[45] G. Laval and L. Excoffier, “SIMCOAL 2.0: A Program to Simulate Genomic Diversity over Large Recombining Regions in a Subdivided Population with a Complex History,” *Bioinformatics*, vol. 20, no. 15, pp. 2485–2487, Oct. 2004, doi: 10.1093/bioinformatics/bth264.

[46] S. E. Ramos-Onsins and T. Mitchell-Olds, “Mlcoalsim: Multilocus Coalescent Simulations,” *Evol. Bioinforma.*, vol. 2007, no. 3, pp. 0–0, Mar. 2007, doi: 10.4137/EBO.S0.

[47] G. Hellenthal and M. Stephens, “msHOT: modifying Hudson’s ms simulator to incorporate crossover and gene conversion hotspots,” *Bioinformatics*, vol. 23, no. 4, pp. 520–521, Feb. 2007, doi: 10.1093/bioinformatics/btl622.

[48] M. Arenas and D. Posada, “Coalescent simulation of intracodon recombination,” *Genetics*, vol. 184, no. 2, pp. 429–437, 2010.

[49] M. Baker, “Next-generation sequencing: adjusting to data overload,” *Nat. Methods*, vol. 7, no. 7, pp. 495–499, Jul. 2010, doi: 10.1038/nmeth0710-495.

[50] S. F. Schaffner, C. Foo, S. Gabriel, D. Reich, M. J. Daly, and D. Altshuler, “Calibrating a coalescent simulation of human genome sequence variation,” *Genome Res.*, vol. 15, no. 11, pp. 1576–1583, Nov. 2005, doi: 10.1101/gr.3709305.

[51] P. Marjoram and J. D. Wall, “Fast ‘coalescent’ simulation,” *BMC Genet.*, vol. 7, p. 16, Mar. 2006, doi: 10.1186/1471-2156-7-16.

[52] L. Excoffier, I. Dupanloup, E. Huerta-Sánchez, V. C. Sousa, and M. Foll, “Robust Demographic Inference from Genomic and SNP Data,” *PLOS Genet.*, vol. 9, no. 10, p. e1003905, Oct. 2013, doi: 10.1371/journal.pgen.1003905.

[53] G. K. Chen, P. Marjoram, and J. D. Wall, “Fast and flexible simulation of DNA sequence data,” *Genome Res.*, vol. 19, no. 1, pp. 136–142, Jan. 2009, doi: 10.1101/gr.083634.108.

[54] P. R. Staab, S. Zhu, D. Metzler, and G. Lunter, “scrm: efficiently simulating long sequences using the approximated coalescent with recombination,” *Bioinformatics*, vol. 31, no. 10, pp. 1680–1682, May 2015, doi: 10.1093/bioinformatics/btu861.

[55] J. Kelleher, A. M. Etheridge, and G. McVean, “Efficient Coalescent Simulation and Genealogical Analysis for Large Sample Sizes,” *PLOS Comput Biol*, vol. 12, no. 5, p. e1004842, May 2016, doi: 10.1371/journal.pcbi.1004842.

[56] A. Eriksson, B. Mahjani, and B. Mehlig, “Sequential Markov coalescent algorithms for population models with demographic structure,” *Theor. Popul. Biol.*, vol. 76, no. 2, pp. 84–91, Sep. 2009, doi: 10.1016/j.tpb.2009.05.002.

[57] Y. Wang *et al.*, “A new method for modeling coalescent processes with recombination,” *BMC Bioinformatics*, vol. 15, p. 273, 2014, doi: 10.1186/1471-2105-15-273.

[58] M. Arenas, “Simulation of Molecular Data under Diverse Evolutionary Scenarios,” *PLoS Comput. Biol.*, vol. 8, no. 5, May 2012, doi: 10.1371/journal.pcbi.1002495.

[59] “ngasp,” *Center for Research in Agricultural Genomics*. https://bioinformatics.cragenomica.es/projects/ngaSP/#/home (accessed May 09, 2019).

[60] J. H. Gillespie, *Population Genetics: A Concise Guide*, 2nd edition. Baltimore, Md: Johns Hopkins University Press, 2004.

[61] X. Yuan, D. J. Miller, J. Zhang, D. Herrington, and Y. Wang, “An Overview of Population Genetic Data Simulation,” *J. Comput. Biol.*, vol. 19, no. 1, pp. 42–54, Jan. 2012, doi: 10.1089/cmb.2010.0188.

[62] J. F. C. Kingman, “The coalescent,” *Stoch. Process. Their Appl.*, vol. 13, no. 3, pp. 235–248, Sep. 1982, doi: 10.1016/0304-4149(82)90011-4.

[63] A. J. Aberer and A. Stamatakis, “Rapid forward-in-time simulation at the chromosome and genome level,” *BMC Bioinformatics*, vol. 14, p. 216, 2013, doi: 10.1186/1471-2105-14-216.

[64] A. Carvajal-Rodríguez, “Simulation of Genomes: A Review,” *Curr. Genomics*, vol. 9, no. 3, pp. 155–159, May 2008, doi: 10.2174/138920208784340759.

[65] F. Balloux, “EASYPOP (Version 1.7): A Computer Program for Population Genetics Simulations,” *J. Hered.*, vol. 92, no. 3, pp. 301–302, 2001.

[66] B. Peng and M. Kimmel, “simuPOP: a forward-time population genetics simulation environment,” *Bioinforma. Oxf. Engl.*, vol. 21, no. 18, pp. 3686–3687, Sep. 2005, doi: 10.1093/bioinformatics/bti584.

[67] R. D. Hernandez, “A flexible forward simulator for populations subject to selection and demography,” *Bioinformatics*, vol. 24, no. 23, pp. 2786–2787, Dec. 2008, doi: 10.1093/bioinformatics/btn522.

[68] V. Sepulveda, R. Solar, A. Inostrosa-Psijas, V. Gil-Costa, and M. Marin, “Towards rapid population genetics forward-in-time simulations,” in *2017 Winter Simulation Conference (WSC)*, Dec. 2017, pp. 2672–2683, doi: 10.1109/WSC.2017.8247993.

[69] D. S. Lawrie, “Accelerating Wright–Fisher Forward Simulations on the Graphics Processing Unit,” *G3 Genes Genomes Genet.*, vol. 7, no. 9, pp. 3229–3236, Sep. 2017, doi: 10.1534/g3.117.300103.

[70] N. J. Grünwald and E. M. Goss, “Evolution and Population Genetics of Exotic and Re-Emerging Pathogens: Novel Tools and Approaches,” *Annu. Rev. Phytopathol.*, vol. 49, no. 1, pp. 249–267, 2011, doi: 10.1146/annurev-phyto-072910-095246.

[71] M. A. Beaumont, W. Zhang, and D. J. Balding, “Approximate Bayesian Computation in Population Genetics,” *Genetics*, vol. 162, no. 4, pp. 2025–2035, Dec. 2002.

[72] G. A. T. McVean and N. J. Cardin, “Approximating the coalescent with recombination,” *Philos. Trans. R. Soc. B Biol. Sci.*, vol. 360, no. 1459, pp. 1387–1393, Jul. 2005, doi: 10.1098/rstb.2005.1673.

[73] G. Ewing and J. Hermisson, “MSMS: a coalescent simulation program including recombination, demographic structure and selection at a single locus,” *Bioinformatics*, vol. 26, no. 16, pp. 2064–2065, Aug. 2010, doi: 10.1093/bioinformatics/btq322.

[74] T. G. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*. Pearson Education, 2004.

[75] J. Basney, R. Raman, and M. Livny, *High Throughput Monte Carlo*. 1999.

[76] G. Shao, F. Berman, and R. Wolski, “Master/slave computing on the Grid,” in *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000) (Cat. No.PR00556)*, May 2000, pp. 3–16, doi: 10.1109/HCW.2000.843728.

[77] C. Montemuiño, A. Espinosa, J.-C. Moure, G. Vera-Rodríguez, S. Ramos-Onsins, and P. H. Budé, "msPar: A Parallel Coalescent Simulator," in *Euro-Par 2013: Parallel Processing Workshops*, D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, Eds. Springer Berlin Heidelberg, 2013, pp. 321–330.

[78] C. Montemuiño, A. Espinosa, J. C. Moure, G. Vera, P. Hernández, and S. Ramos-Onsins, "Approaching Long Genomic Regions and Large Recombination Rates with msParSm as an Alternative to MaCS," *Evol. Bioinforma. Online*, vol. 12, pp. 223–228, Oct. 2016, doi: 10.4137/EBO.S40268.

[79] J. Vienne, "Benefits of Cross Memory Attach for MPI libraries on HPC Clusters," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, New York, NY, USA, Jul. 2014, pp. 1–6, doi: 10.1145/2616498.2616532.

[80] "FAQ: Tuning the run-time characteristics of MPI sm communications," Aug. 25, 2015. https://www.open-mpi.org/faq/?category=sm (accessed Mar. 30, 2017).

[81] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 5, pp. 662–675, May 1986, doi: 10.1109/TSE.1986.6312961.

[82] I. Riakiotakis, F. M. Ciorba, T. Andronikos, and G. Papakonstantinou, "Distributed dynamic load balancing for pipelined computations on heterogeneous systems," *Parallel Comput.*, vol. 37, no. 10, pp. 713–729, Oct. 2011, doi: 10.1016/j.parco.2011.01.003.

[83] A. T. Chronopoulos, R. Andonie, M. Benche, and D. Grosu, "A class of loop self-scheduling for heterogeneous clusters," in *Proceedings 2001 IEEE International Conference on Cluster Computing*, Oct. 2001, pp. 282–291, doi: 10.1109/CLUSTR.2001.959989.

[84] C.-T. Yang and S.-C. Chang, "A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters," in *Computational Science — ICCS 2003*, Berlin, Heidelberg, 2003, pp. 1079–1088, doi: 10.1007/3-540-44864-0_112.

[85] F. M. Ciorba, T. Andronikos, I. Riakiotakis, A. T. Chronopoulos, and G. Papakonstantinou, "Dynamic multi phase scheduling for heterogeneous clusters," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, Apr. 2006, p. 10 pp.-, doi: 10.1109/IPDPS.2006.1639308.

[86] F. M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou, and A. T. Chronopoulos, "Enhancing self-scheduling algorithms via synchronization and

weighting," *J. Parallel Distrib. Comput.*, vol. 68, no. 2, pp. 246–264, Feb. 2008, doi: 10.1016/j.jpdc.2007.07.003.

[87] I. Shlyakhter, P. C. Sabeti, and S. F. Schaffner, "Cosi2: an efficient simulator of exact and approximate coalescent with selection," *Bioinformatics*, vol. 30, no. 23, pp. 3427–3429, Dec. 2014, doi: 10.1093/bioinformatics/btu562.

Appendix A – msparsm

Full code at: https://github.com/cmontemuino/msparsm

The *msParSm* application is an evolution of *msPar*, the parallel version of the coalescent simulation program *ms*, which removes the limitation for simulating long stretches of DNA sequences with large recombination rates without compromising the accuracy of the standard coalescence.

Pre-requisites

- Linux GNU Compiler 4.9.1 (or greater)
- OpenMPI 1.10.1 (other releases in the branch 1.10 should be fine)
- Version 1.8.x could potentially be fine, but please notice that *msParSm* was not thoroughly tested with such a version.
- CMake 3.5.1 (or greater) **OR** GNU Make 3.81 (or greater)

How to Build

There are two ways of building *msParSm*: CMake and Make. If you have installed CMAKE with a version greater than 3.5.0, go with CMake; otherwise, you should use Make.

```
cmake <src-path> -DCMAKE_INSTALL_PREFIX=<install-path>
make install
```

Binary files will be put into the `bin` folder (which is already *git ignored*).

How to Use

Usage is the mostly the same as with traditional *ms*, but you need to run it through *OpenMPI*. Next example will run the application using 4 threads:

```
mpirun -n 4 bin/msparsm 10 20 -seeds 40328 19150 54118 -t 100
-r 100 100000 -I 2 2 8 -eN 0.4 10.01 -eN 1 0.01 -en 0.25 2 0.2
-ej 3 2 1 -T > results.out
```

Appendix B – mlcoalsim-v2

Full code at: https://github.com/cmontemuino/mlcoalsim-v2

Quick Start

This project requires both OpenMPI and CMake.

If you are running on Mac OSX and you happen to use Homebrew, then you might want to check a custom for installing OpenMPI: https://github.com/cmontemuino/homebrew-custom .

Enabling MPI in the IDE

If you open the project with a contextual IDE, for example, CLion, then you will notice that all the code related to MPI will not be "clickable." One way to resolve this is by setting the environment variable `WITH_MPI`. You just need to provide whatever value to it.

How to Build

Please refer to the maskfile.md file that contains a section for building the project.

How to Run the Examples

Several examples are provided in the `examples` folder. You can run most of them in the following way:

```
mpirun -np 4 build/mlcoalsimXmpi_ZnS
examples/example00/Example1locus_1pop_mhit0_rec100.txt
build/Example1locus_1pop_mhit0_rec100.out

# Without MPI:
# build/mlcoalsimX
examples/example00/Example1locus_1pop_mhit0_rec100.txt
build/Example1locus_1pop_mhit0_rec100.out
```

In the case of the `example01` and `example10`, where a "prior" file is being used, you need to switch the directory first. For example:

```
pushd examples/example10
mpirun -np 4 ../../build/mlcoalsimXmpi  Example10loci.txt
../../build/Example10loci.out
popd
```

Other examples you might want to run:

- `build/mlcoalsimX examples/example00/Example1locus_1pop_mhit0_rec100.txt build/Example1locus_1pop_mhit0_rec100.out`

- `mpirun -np 4 build/mlcoalsimXmpi examples/example00/Example1locus_1pop_mhit0_rec100.txt build/Example1locus_1pop_mhit0_rec100.out`
- `build/mlcoalsimX_ZnS examples/example00/Example1locus_1pop_mhit0_rec100_S20_n100.txt build/Example1locus_1pop_mhit0_rec100_S20_n100.out`
- `mpirun -np 4 build/mlcoalsimXmpi_ZnS examples/example00/Example1locus_1pop_mhit0_rec100_S20_n100.txt build/Example1locus_1pop_mhit0_rec100_S20_n100.out`