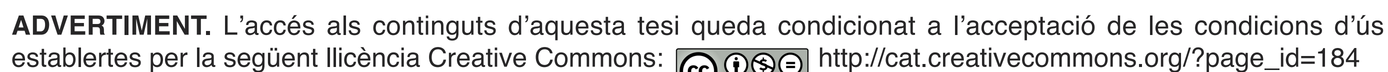
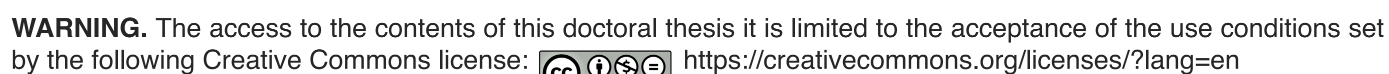
Universitat Autònoma de Barcelona



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

Departament d'Arquitectura de
Computadors i Sistemes Operatius

Execution strategies for memory-bound applications on NUMA Systems

Thesis submitted by **Josefina Lenis** for the degree of philosophae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Miquel Àngel Senar, developed at the Computer Architectures and Operating Systems department, PhD program in Computer Science (research line: High Performance Computing)

Barcelona, October 2018

Execution strategies for memory-bound applications on NUMA Systems

Thesis submitted by **Josefina Lenis** for the degree of philosophae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Miquel Àngel Senar, developed at the Computer Architectures and Operating Systems department, PhD program in Computer Science (research line: High Performance Computing).

Supervisors

Dr. Miquel Àngel Senar

Phd Student

Josefina Lenis

Ackowledgements

First of all I want thank my advisor Miquel. Thanks to his support and guidance I discover a strength that I did not know that I have. His actions and advice marked me and became an example of fine academic work.

During my stay in the USA I fell in love with a small city called Eugene in the state of Oregon. Where I still have very goods friends. The main responsible converting that time in an amazing experience was the Dr. Sameer Shende. To whom I will always be deeply thankful for his kindness and warm-hearted spirit. He invited me to work with his team, an incredible smart and capable group of people, where I faced several challenges and I've learned a lot. Thank you for the technical and moral support Sameer!

I want to thank each member of the Computer and Operating Systems Department. In particular the people that stood by me in numerous occasions: Eduardo Cesar, Anna Sikora -thank you for always encourage me to continue, your words and kind gestures help me more than I can express- , Remo Suppi, Lola Rexachs, Emilio Luque and Gemma Roque. My fellow colleagues during some of this time Albert Gutierrez, Aprigio Bezerra, Francisco Cruz, Javi Navarro, Francisco Borges, Javier Panadero, Ferran Badosa, Hugo Meyer, Cecilia Jaramillo, Joe Carrión, Laura Espinola and Jorge Villamayor.

A special mention for a close friend of mine in this PhD adventure: Àlex Chacón: I have no words to describe my gratitude to you. Your help was priceless. Thanks for your advices, your feedback, your extensive knowledge, for being there and for everything you have done for me. I truly admire you. I do not want to forget of my awesome friends Marcela Castro, Claudia Rosas y Pili Gomez. Thanks for listening and thanks for remind me what truly matter.

To my beloved husband Diego, my partner, my companion, my pillar and my rock. Thank you for supporting the long nights, for listening rehearsal my presentations, for help me handle the stress, for taking care of me when I needed the most, I love you.

To my family, my dad and my mom, my examples of sacrifice, hard-work and enormous

heart. To my unconditional siblings, nieces and nephews, the Flanders family: Juan P., María, Emma, David, Caro, Charly, Juan F., Agus, Gonzalo, Facu, Guille y Román. I love you and I miss you all.

Last but not least, I want to thanks all my friends that in one way or an other helped this work to come through: Nuni, Fede, Andrés, Bet, Petra, Pedro, Prass and Gero.

Abstract

Over the last several years, many sequence alignment tools have appeared and become popular thanks to the fast evolution of next generation sequencing (NGS) technologies. Obviously, researchers that use such tools are interested in getting maximum performance when they execute them in modern infrastructures. Today's NUMA (Non-Uniform Memory Access) architectures present major challenges in getting such applications to achieve good scalability as more processors/cores are used. The memory system in NUMA systems shows a high complexity and may be the main cause for the loss of an application's performance. The existence of several memory banks in NUMA systems implies a logical increase in latency associated with the accesses of a given processor to a remote bank. This phenomenon is usually attenuated by the application of strategies that tend to increase the locality of memory accesses. However, NUMA systems may also suffer from contention problems that can occur when concurrent accesses are concentrated on a reduced number of banks. Sequence alignment tools use large data structures to contain reference genomes to which all reads are aligned. Therefore, these tools are very sensitive to performance problems related to the memory system. The main goal of this study is to explore the trade-offs between data locality and data dispersion in NUMA systems. We introduced a series of methodical steps to characterize NUMA architectures and to help understand the potential of the resources. With this information we designed and experimented with several popular sequence alignment tools on two widely available NUMA systems to assess the performance of different memory allocation policies and data partitioning and replication strategies. We find that there is not one method that is best in all cases. However, we conclude that memory interleaving is the memory allocation policy that provides the best performance when for applications that used a large centralized data structured on a large number of processors and memory banks In the case of data partitioning and replication, the best results are usually obtained when the number of partitions used is greater, and in some cases, combined with an interleave policy.

Resumen

Durant els últims anys, moltes eines d'alineament de seqüències han aparegut i s'han popularitzat gràcies a la ràpida evolució de les tecnologies de Next Generation Sequencing (NGS). Evidentment, els investigadors que utilitzen aquestes eines estan interessats en obtenir el màxim rendiment quan les executen en infraestructures modernes. Actualment, les arquitectures NUMA (accés a la memòria no uniforme) presenten grans reptes en aconseguir que aquestes aplicacions tinguin una bona escalabilitat a mesura que s'utilitzen més processadors/nuclis. El sistema de memòria dels sistemes NUMA mostra una gran complexitat i pot ser la causa principal de la pèrdua del rendiment d'una aplicació. L'existència de diversos bancs de memòria en sistemes NUMA implica un augment lògic de la latència associada als accessos d'un processador donat a un banc remot. Aquest fenomen sol estar atenuat per l'aplicació d'estratègies que tendeixen a augmentar la localitat d'accés a la memòria. Tanmateix, els sistemes NUMA també poden patir problemes de contenció que es poden produir quan els accessos concurrents es concentren en un reduït nombre de bancs. Les eines d'alineació de seqüències utilitzen grans estructures de dades per contenir genomes de referència als quals totes les lectures estan alineades. Per tant, aquestes eines són molt sensibles als problemes de rendiment relacionats amb el sistema de memòria. L'objectiu principal d'aquest estudi és explorar les compensacions entre la localitat de dades i la dispersió de dades en els sistemes NUMA. Hem introduït una sèrie de passos metòdics per caracteritzar arquitectures NUMA i per ajudar a comprendre el potencial dels recursos. Amb aquesta informació, hem dissenyat i experimentat diverses eines d'alineació de seqüència populars en dos sistemes NUMA àmpliament disponibles per avaluar el rendiment de les diferents polítiques d'assignació de memòria i les estratègies de partició i replicació de dades. Trobem que no hi ha un mètode que sigui millor en tots els casos. Tanmateix, es conclou que la intercalació de memòria és la política d'assignació de memòria que proporciona el millor rendiment quan s'utilitza una gran quantitat de processadors i bancs de memòria. En el cas de

la partició i la replicació de dades, els millors resultats solen obtenir-se quan la quantitat de particions que s'utilitza és més gran, de vegades combinada amb una política interleave.

Resumen

En los últimos años, muchas herramientas de alineadores de secuencias han aparecido y se han hecho populares por la rápida evolución de las tecnologías de secuenciación de próxima generación (NGS). Obviamente, los investigadores que usan tales herramientas están interesados en obtener el máximo rendimiento cuando los ejecutan en infraestructuras modernas. Las arquitecturas NUMA (acceso no uniforme a memoria) de hoy en día presentan grandes desafíos para lograr que dichas aplicaciones logren una buena escalabilidad a medida que se utilizan más procesadores/núcleos. El sistema de memoria en los sistemas NUMA muestra una alta complejidad y puede ser la causa principal de la pérdida del rendimiento de una aplicación. La existencia de varios bancos de memoria en sistemas NUMA implica un aumento lógico en la latencia asociada con los accesos de un procesador dado a un banco remoto. Este fenómeno generalmente se atenúa mediante la aplicación de estrategias que tienden a aumentar la localidad de los accesos a la memoria. Sin embargo, los sistemas NUMA también pueden sufrir problemas de contención que pueden ocurrir cuando los accesos concurrentes se concentran en un número reducido de bancos. Las herramientas de alineadores de secuencia usan estructuras de datos grandes para contener genomas de referencia a los que se alinean todas las lecturas. Por lo tanto, estas herramientas son muy sensibles a los problemas de rendimiento relacionados con el sistema de memoria. El objetivo principal de este estudio es explorar las ventajas y desventajas entre la ubicación de datos y la dispersión de datos en los sistemas NUMA. Hemos introducido una serie de pasos metódicos para caracterizar las arquitecturas NUMA y ayudar a comprender el potencial de los recursos. Con esta información, diseñamos y experimentamos con varias herramientas de alineación de secuencias populares, en dos sistemas NUMA ampliamente disponibles para evaluar el rendimiento de diferentes políticas de asignación de memoria y estrategias de replicación y partición de datos. Encontramos que no hay un método que sea el mejor en todos los casos. Sin embargo, concluimos que aplicar interleave a la memoria es la política de alocación

de memoria que proporciona el mejor rendimiento cuando se utiliza una gran cantidad de procesadores y bancos de memoria. En el caso de la partición y replicación de datos, los mejores resultados se obtienen generalmente cuando el número de particiones utilizadas es mayor, a veces combinado con una política de interleave.

Contents

1 Introduction **1**

1.1 Introduction . . . 2
1.2 Contribution . . . 3
1.3 Related work . . . 4
1.4 Thesis Structure . . . 6

2 HPC Systems **8**

2.1 Introduction . . . 9
2.2 Brief history of parallel architectures . . . 9
2.3 Parallel Architectures . . . 11
2.3.1 Shared-memory system . . . 11
2.3.2 Distributed-memory systems . . . 13
2.3.3 Hybrid systems . . . 13
2.4 Study Case: NUMA Systems . . . 14
2.4.1 Our Infrastructure . . . 15
2.5 Measuring performance . . . 16
2.5.1 Definitions . . . 16
2.5.2 Performance tools . . . 18

3 Analysis Methodology **22**

3.1 Introduction . . . 23
3.2 Calculating Theoretical Bandwidth . . . 24
3.3 Measuring Latency . . . 25
3.4 Measuring Bandwidth . . . 26
3.5 Measuring Contention . . . 31
3.6 Creating a guideline . . . 33

4 Sequencing aligners **36**

4.1 Introduction 37
4.2 Motivation 37
4.3 NGS Workflows 38
4.3.1 Sequencing 38
4.3.2 Alignment 40
4.3.3 Variant calling 41
4.3.4 Annotation 42
4.4 Sequence Aligners 43
4.4.1 Burrows Wheeler Aligner 44
4.4.2 Bowtie2 45
4.4.3 Genome Multitool 3 45
4.4.4 Scalable Nucleotide Alignment Program 45

5 Experimentation 48
5.1 Performance of sequence aligners on NUMA systems 49
5.2 Allocation Strategies and Data Partitioning 51
5.2.1 Analysis of memory allocation 51
5.2.2 Data partitioning and replications strategies 52
5.3 Experimental Results 54
5.3.1 Analysis of memory allocation policies 55
5.3.2 Data partitioning and replication strategies 59
5.3.3 Summary results 63
5.4 Annexed results 66

6 Conclusions and Future lines 71
6.1 Conclusions 72
6.2 Future Lines 73

Bibliography 76

List of Figures

2.1 More’s Law vs Number of cores Source [1] 10
2.2 A UMA architecture provides each CPU core (C1-C4) the same memory access latency and bandwidth. Source [2] 12
2.3 A NUMA architecture connects different NUMA nodes (Nodes 1-4) - typically multicore CPUs (C1-C4) - via interconnect links (Link), to enable a single logically shared global memory. . Source [2] 12
2.4 In a distributed-memory model, the memory is physically and logically distributed among the individual processing units (P1-P3), e.g., CPUs. Accessing data from remote memory locations requires to initiate a data transfer protocol, such as point-to-point communication provided by the MPI. Figure source: [2] 13
2.5 In hierarchical systems the distributed- and shared-memory model are combined. In the depicted case three UMA-based shared-memory nodes are connected via a network connection, however, NUMA systems can be used in a similar manner. Figure source: [2] 14
2.6 Schematic diagram of the AMD Opteron 6376 architecture (Abu-Dhabi) . . 15
2.7 Schematic diagram of the Intel Xeon E5 4620 architecture (Sandy Bridge) . 15

3.1 Latency profile of AMD Opteron 6376 using lmbench 26
3.2 All possible bandwidth values for one-processor execution on AMD system . 27
3.3 All possible bandwidth values for one-processor execution on Intel system . . 27
3.4 BW measures for all systems using 16 threads 29
3.5 BW measures for all systems using 32 threads 30
3.6 Concurrency stress test on AMD system 32
3.7 Concurrency stress test on Intel system 33
3.8 Methodology to obtain a guideline of NUMA execution strategies 35

4.1 Standard variant calling workflow 38
4.2 Single-nucleotide polymorphism 42

4.3 Types of SNP . . . 42

5.1 Scalability study for BWA-ALN on AMD architecture . . . 50
5.2 Experimentation approaches . . . 51
5.3 Hybrid experimentation . . . 54
5.4 Different memory allocation policies. The lower the better. Arch: AMD. Dataset: GCAT Synthetic Input . . . 56
5.5 Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: GCAT Synthetic Input . . . 57
5.6 Different memory allocation policies. The lower the better. Arch: AMD. Dataset: NA12878 Real Input . . . 58
5.7 Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: A12878 Real Input . . . 59
5.8 Different memory allocation policies. The lower the better. Arch: AMD. Dataset: GCAT Synthetic Input . . . 62
5.9 Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: GCAT Synthetic Input . . . 63
5.10 Different memory allocation policies. The lower the better. Arch: AMD. Dataset: NA12878 Real Input . . . 64
5.11 Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: A12878 Real Input . . . 65
5.12 Execution times: Summary results for all aligners. The lower the better . . . 66

List of Tables

3.1 Processor and memory information of the architectures used 24
3.2 STREAM's vector kernels 27

5.1 Detailed information about the aligners. 54
5.2 Instances created for each aligner 61
A3 Execution times for BOWTIE2 using different memory allocation policies . . 67
A4 Execution times for BOWTIE2 using data partitioning 68
B5 Execution times for BWA-MEM using different memory allocation policies 68
B6 Execution times for BWA-MEM using data partitioning 69
C7 Execution times for GEM using different memory allocation policies 69
C8 Execution times for GEM using data partitioning 69
D9 Execution times for SNAP using different memory allocation policies . . . 70
D10 Execution times for SNAP using data partitioning 70

1
Introduction

"Self-control means wanting to be effective at some random point in the infinite radiations of my spiritual existence."

- Franz Kafka

1.1 Introduction

NUMA (Non-Uniform Memory Access) systems have become increasingly common with the passage of time. Nowadays practically any HPC facility, research center or university has these systems in its infrastructure. Despite its complex hardware design, programs and applications do not have to be adapted to run on them (unlike other hybrid or semi-hybrid systems such as accelerators or GPUs). This feature is possibly one of its great advantages but at the same time its Achilles heel. Due that when applications are not aware of the hardware where they run on, they end up making sub-optimal use of resources. Writing parallel programs that exhibit good scalability is far from easy. Studying the physical characteristics of a NUMA architecture is key to understanding the impact that these can have on the performance of the applications that are run on. However, if programmers wrote extremely optimal and architecture-dependant code, they would narrow the machines where such code could be executed on, so it is not a feasible solution either. Our initial hypothesis is that somewhere between these two points there is the proper balance: where the advantages surpass the drawbacks, by applying execution strategies to applications to optimize their execution time without loosing flexibility.

In this research we propose a benchmark suite to characterize NUMA architectures, through a set of well-known benchmarks. This information helps to see the limitations of our system for applications that intensively use the memory system, either in idealistic scenarios with purely sequential access patterns or unfavorable scenarios where accesses are completely random. The acquired knowledge is translated into a guideline for final users of NUMA system, in order to develop NUMA execution strategies for memory bound applications. One main aspect of our proposal is that the user can take benefit of the architecture while improving the performance of the application used without modifying a single line of code.

To validate the NUMA execution strategies based on our methodology, we designed an experimentation using real memory bound applications: genomic mappers. Most mappers take advantage of parallelization techniques in order to reduce the computational demands involved in the alignment of millions of reads onto a reference sequence. Numerous software tools have been developed in recent years and many studies have evaluated their performances through several comparison criteria. In general, comparison studies have focused on mapper sensitivity and mapper accuracy, while computational and time requirements have received comparatively less attention.

Our proposed NUMA execution strategies mitigate two major drawbacks of these systems: the latency of remote data accesses and the concurrence of multiple threads contending for

a single shared resource. We executed the mappers using different techniques of memory allocation like interleave but we also introduce a proposal based multiple instances to reduce the intense usage of interconnection bus. As a result we obtained improvements in the mappers performance of up to 5x. We also provide an easy to follow guideline for the final user to understand the architecture been used and make the most of it.

1.2 Contribution

As a result of this study we published three papers:

1. Josefina Lenis and Miquel Àngel Senar ”On the Performance of BWA on NUMA Architectures.” TrustCom/BigDataSE/ISPA (3) 2015: 236-241 (PBIO) [3]

2. Josefina Lenis and Miquel Àngel Senar ”Optimized Execution Strategies for Sequence Aligners on NUMA Architectures.” Euro-Par Workshops 2016: 492-503 (PBIO) [4]

3. Josefina Lenis, Miquel Àngel Senar ”A performance comparison of data and memory allocation strategies for sequence aligners on NUMA architectures.” Cluster Computing 20(3): 1909-1924 (2017) [5]

In our first work, we analyzed the performance of BWA-ALN, (Burrows-Wheeler Aligner) [6], where we detected scalability problems exhibited by BWA-ALN, and we proposed simple system-level techniques to alleviate them. We obtained results up to 4-fold speed higher than the original BWA-ALN multithread implementation.

In the second paper, we extended the study to other popular aligners from the literature. We analyzed performance problems of four aligners that constitute representative examples of the two most commonly used algorithmic strategies: hash tables and Burrow Wheeler Transform (BWT). The aligners under study were: BWA-MEM [7] (a newer version of BWA-ALN especially suited to dealing with longer reads), BOWTIE2 [8] (an ultrafast and memory-efficient tool for aligning sequencing reads to long reference sequences), GEM (GEnome Multi-tool) [9] and SNAP (Scalable Nucleotide Alignment Program) [10]. These aligners are widely used by the scientific community and real production centers, and frequently updated by developers. Although all the aligners under study take advantage of multithreading execution, they exhibit significant scalability limitations on NUMA systems. Data sharing between independent threads and irregular memory access patterns constitutes performance-limiting factors that affect the studied aligners. We have applied various memory allocation

policies as well as several data distribution strategies to these aligners and we have obtained promising results in all cases, reducing memory-bound drawbacks and increasing scalability.

In the last paper, we also extend our previous results by expanding our comparison study to two different NUMA systems, one based on Intel Xeon and the other one based on AMD Opteron, and by introducing a hybrid execution strategy that combines both data partitioning and memory allocation policies.

1.3 Related work

Since the emergence of the NUMA architectures in the 80s, there has been studies of the asymmetric accesses in these systems. Since then, NUMA systems have changed a lot compared to the old ones. And while many of the fundamental concepts that have been studied for decades remain, there are many others that need to be revised in order not to fall into fallacies when it comes to predicting the performance of applications in these systems.

Both researchers and programmers a like, have helped to broaden the understanding of these systems and improve their use. Mainly through performance models, libraries, data mapping tools and threads.

Between the most outstanding studies three great categories can be described:

Threads and data mapping tools

Challenges in memory access on NUMA systems have been addressed by some approaches that tried to optimize locality at the OS level. The AutoNUMA patches for Linux [11] implement locality-driven optimization along two main heuristics. First, the threads migrate toward nodes holding the majority of the pages accessed by these threads. Second, the pages are periodically unmapped from a process address space and, upon the next page fault, migrated to the requesting node.

Carrefour [12] [13] is another recent tool that consists of a memory-placement algorithm for NUMA systems that focuses on traffic management. As in our approach, Carrefour focuses on memory congestion as the primary source of performance loss in current NUMA systems. It places memory so as to minimize congestion on interconnecting links for memory controllers. By using global information and memory-usage statistics, Carrefour applies three main techniques: memory collocation (to move memory to a different node so that accesses are likely local), replication (copying memory to several nodes so that threads from each node

can access it locally) and interleaving (moving memory so that it is distributed evenly among all nodes).

AsymSched [14] is a dynamic thread and memory placement algorithm for Linux, that takes into account the bandwidth asymmetry in NUMA systems. AsymSched is based on 3 techniques: thread migration, full memory migration and dynamic memory migration.

In the three mentioned cases (AutoNUMA, Carrefour and AsymSched), memory management mechanisms are implemented in the Linux kernel and require a patch to be applied to the virtual memory layer. Our work, however, focuses on the evaluation of techniques that can be applied at the application level and therefore don't require root permissions to be applied to any NUMA system.

Application characterization tools

An interesting tool is Tabarnac [15], that not only identifies under-performance memory access patterns but also can provide with a solution of how to improve the source code of the program. Tabarnac is only used with benchmarks where a clear access pattern can be identified. However when complex combination of access pattern happen simultaneously, can't offer a solid solution.

In [16] Majo et. al provides a series of guide-lines software-developer-oriented to achieve an efficient use of NUMA systems. In this paper is remarked how the access pattern of a program have a huge impact in the overall performance.

Hardware characterization tools

In [17] the authors present a comprehensive study of 2 NUMA processors: Intel (Sandy Bridge-EP) and AMD (Bulldozer). They proposed a set of benchmarks to perform an in-depth analysis of current ccNUMA multiprocessor systems with processors. One of the most important contribution of this work is related to the specification of the cache-coherent protocols used by the different manufacturers.

NUMA Performance Models

Similar study to our work, Braithwaite et al. [18], who propose an "Empirical Memory-Access Cost Model in Multicore NUMA ARchitecture". Using STREAM benchmark [19][20] and LMBENCH [21], the author develops a benchmarking methodology, who through simple task-scheduling experiments to improves the performance of applications. Our study extends

this notion and also considers contention problems, not only memory-access cost due latency or memory bandwidth. We also consider the importance of executing programs in instances as a possible outcome prediction of our model.

The performance model Roofline [22] is designed to assist in software development and optimization by providing detailed and accurate information about machine characteristics. The Roofline model is a visually performance model used to relate computational performance with the memory capacities of architectures. The Roofline model helps to find possible bottlenecks and to identify the performance boundaries for a given processor, however does not provide any solution on how to solve such issues. An extended version of the Roofline [23] to NUMA systems was presented by Lorenzo et al. In this work, the model was extended to show the dynamic evolution of the execution of a given code. In it, is successfully shown the impact of thread migration. Although this work is a useful tool to understand and characterize the behaviour of the execution of parallel codes, only is tested with computational-bound applications.

Mappers enhancer tools

Genome alignment problems have been considered by Misale et al. [24]. The authors implement a framework to work under BOWTIE2 and BWA to improve the local affinity of the original algorithm. Herzeel et al. [25] replaces the pthread-based parallel loop in BWA with a Cilk *for* loop. Rewriting the parallel section using Cilk removes the load imbalance, resulting in a factor 2x performance improvement over the original BWA.

In both cases - Misale et al. and Herzeel et al. - the source code of the applications -aligners- are modified, which might be a costly action and dependent on the application version. Abuin et. al. [26] presented a big data approach to solving BWA scalability problems. They introduce a tool named BigBWA that enables them to run BWA on several machines although it does not provide a clear strategy for dividing the data or setting the number of instances. In contrast, our approach can be applied to different aligners with minimal effort and, although not tested yet, it can be easily applied to distributed memory systems. Our work is complementary to all the works mentioned above.

1.4 Thesis Structure

The thesis is structured as follows: In chapter 2 we explain briefly the history of parallel systems then we describe the basic concepts of NUMA architectures and provide concrete

details of the two systems used in our experiments. In chapter 3 we present our methodology to characterize the NUMA systems and obtain *NUMA Performace Aspects*. Chapter 4 introduces the problem of sequence alignment and a behavioral characterization of aligners used in this study. In chapter 5 we describe in detail the NUMA execution strategies proposed to validate our methodology, we also explained all scenarios used to evaluate the performance improvement of aligners under study and analyzed the results obtained in our experiments . Finally, in chapter 6 there is the conclusions and the future lines of work.

2

HPC Systems

”In the fight between you and the world, second the world. ”

- Franz Kafka

2.1 Introduction

This chapter briefly reviews some fundamental concepts related to high performance computing. In particular, we present an introduction to the basic elements that characterize the architectures of current computers. The description starts from the analysis of the dominant architecture in current microprocessors and finishes by briefly reviewing the computer architectures that are commonly used to solve computationally complex problems. This sets the base to discuss NUMA architectures advantages and drawbacks in depth. Finally, a brief review of all the HPC performance tools used and evaluated during this study.

2.2 Brief history of parallel architectures

In 1965 Gordon Moore predicted that the number of transistors in an integrated circuit followed a progression in which that value doubled every 18 months. This prediction, known as Moore's Law, has been fulfilled since then and continues in force. Translated in terms of performance, this prediction has meant a constant increase in the computing capacity of successive generations of microprocessors thanks to the increase in the frequency of its clock and the integration of architectural solutions such as the parallelism at the instruction level that decreased the latency of the memory system. In this way, sequential programs could be executed more quickly and this gain could be achieved transparently for the programmer. The progression in the improvement of the performance of the computers was altered at the beginning of the last decade by the physical limitations that derived from the problems of heat dissipation and consumption exhibited by the microprocessors. As of 2004, these limitations stopped the increase in the frequency of the processors and generated a technological solution that focused on the introduction of processors that instead of having a single processing unit (control unit plus logical arithmetic unit) by chip, they had several of them grouped in an entity called core.

In the 2.1 the evolution of the integration levels of the processors, their frequencies and the number of cores is shown [1]. Evolution in time of the number of transistors per chip (blue), the frequency (red) and the number of cores per processor (purple).

The architecture of a multicore processor is based on the existence of two or more execution cores within a single processor. The operating system perceives each of its execution core as a discrete logic processor with all its associated execution resources. Multicore processors perform more work per cycle, are able to operate at a lower frequency, and present

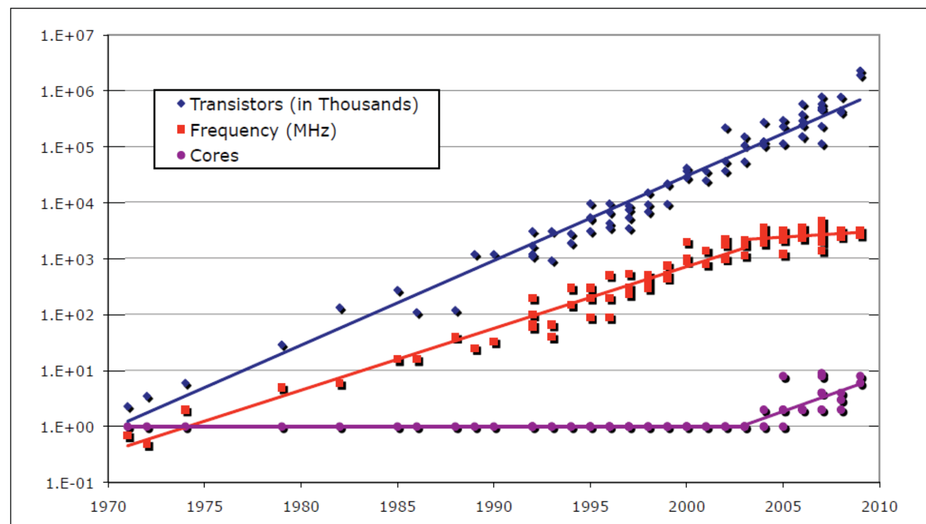


Figure 2.1: More's Law vs Number of cores Source [1]

improvements in the performance of calculation activities and bandwidth intensive.

The emergence of multicore processors has meant, however, a paradigm shift from the point of view of improving the execution of applications. This improvement is no longer achieved automatically by the effects of increases in the clock frequency, but must be achieved by conveniently exploiting the inherent parallelism offered by the new processors in the form of multiple cores integrated in single device.

However, unlike single-core processors, much of the responsibility for better performance now falls on the shoulders of programmers. The era where programmers relied on hardware designers to make their programs faster has ended [27]. The performance improvements in the new generations of computers will depend on the changes that are introduced in the applications and in the system tools so that the computing capacity provided by the multicore systems is exploited.

Currently Moore's law is fulfilled at the level of cores in addition to the level of transistors. The increase in the average of cores is shown, since the appearance of the multicore architectures in 2004, in Supercomputers (infrastructures composed of thousands of multicore processors). The increase in cores has doubled every two years since the emergence of multicore processors in 2004 [28]

The limits that separate high-performance computing from ordinary computing are rather arbitrary, since HPC refers merely to calculations that are made on computers more powerful than the standards [1]. With 500 u$s today you can buy a laptop that have more computing capacity, more main memory and a hard drive with more space than a computer of 1000000

u$s in the year 1985 [27].

2.3 Parallel Architectures

The parallel architectures are defined under the MIMD (Multiple Instruction Multiple Data) paradigm, where different instruction flows are applied to several data streams. In this section we will review the 3 broad categories that parallel architectures are normally classified into:

- Shared-memory systems
- Distributed-memory systems
- Hierarchical (hybrid) systems

2.3.1 Shared-memory system

In shared memory systems, all CPUs (one or more) share the one physical memory address space. Within this category there are 2 important set of shared-memory systems that have a different performance and different complexity:

- Uniform Memory Access (UMA): where latency and bandwidth are the same for all processors and all memory locations.
- Non-Uniform Memory Access (NUMA) in this case the memory is physically distributed through several processors but logically shared, so the memory addresses are global. Memory access times vary depending on the processor and the memory location that is intended to access.

For this paradigm the most popular programming libraries include OpenMP[29], Cilk[30] and Pthreads[31].

UMA

The most simple layout of parallel system is the UMA. Processors access memory through the shared bus as seen in Figure 2.2. As stated before, all CPUs have the same latency and bandwidth when accessing memory. This is why are known as Symmetric multiprocessor (SMP) where systems use this centralized memory approach, where each processor is connected to a shared bus. This shared bus handles all accesses to main memory and I/O. Communication

between CPUs is implicit and transparent. All these characteristics makes it a very easy to program for. Converting a serial code into a parallel code on UMA architecture requires a minimal effort. However, this symmetric pattern does not scale properly, there is a limited number of CPUs that can be added to a UMA architecture without turning the shared bus into a bottleneck. In order to maintain scalability of memory bandwidth with CPU number, non-blocking crossbar switches can be built that establishes point-to-point connections between sockets and memory module but at the cost of giving up the UMA principle [32].

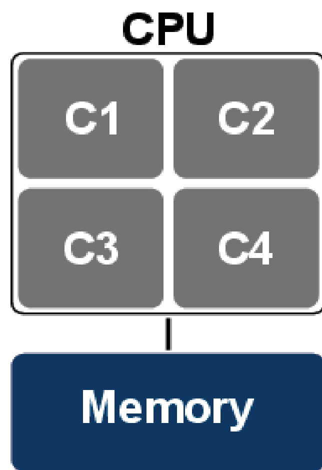


Figure 2.2: A UMA architecture provides each CPU core (C1-C4) the same memory access latency and bandwidth. Source [2]

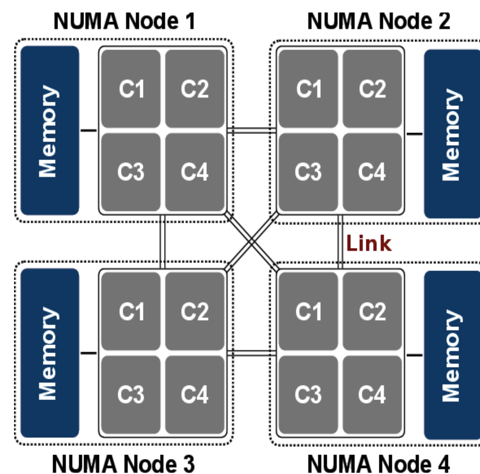


Figure 2.3: A NUMA architecture connects different NUMA nodes (Nodes 1-4) - typically multicore CPUs (C1-C4) - via interconnect links (Link), to enable a single logically shared global memory. . Source [2]

NUMA

In NUMA systems, the main memory is physically distributed across the processors but, logically, this set of main memories appears as only one large memory, so the accesses to different parts are done using global memory addresses [1]. A processor and its respective memory are called a NUMA node. A program running on a particular processor can also access data stored in memory banks associated with other processors in a coherent way but at the cost of increased latency compared to accessing its own local memory bank. In general, parallel applications that may run using multiple processors are not usually designed taking the NUMA architecture into account, mainly because is still a shared-memory system and processors do not explicitly communicate with each other so communication protocols are hidden within the system. However in NUMA architectures that can have a high cost in

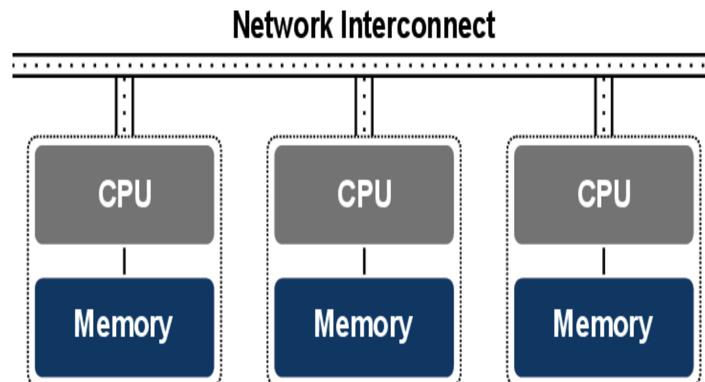


Figure 2.4: In a distributed-memory model, the memory is physically and logically distributed among the individual processing units (P1-P3), e.g., CPUs. Accessing data from remote memory locations requires to initiate a data transfer protocol, such as point-to-point communication provided by the MPI. Figure source: [2]

performance if the data is not allocated in an optimal fashion as too many remote access may occur.

2.3.2 Distributed-memory systems

In parallel distributed-memory computers, processing units do not share memory but each has its own memory address space as shown in Figure 2.4. This model originates from a time where a CPU contained a single processing core. Such a setup is typically not found anymore in today's cluster systems due to the advent of multicore CPUs. However, the model still serves well as an introduction to distributed computing, as the concepts are still applicable[2]. When a parallel program is executed, the compute nodes exchange information between them by passing messages that are transferred over interconnection networks (Gigabit Ethernet, Infiniband, etc). The dominant programming standard in this type of systems is MPI [33], although there are other alternatives such as co-Array Fortran[34], GASnet[35], Charm++ [36] among others. In distributed-memory system there are no longer race conditions to access main memory however the communication between process becomes explicit and need to be handled fully by the programmer. This increases significantly the programming complexity.

2.3.3 Hybrid systems

Nowadays, no system is purely shared-memory or strict distributed-memory, most configurations are combination of both. Commonly composed by a layout of NUMA architectures

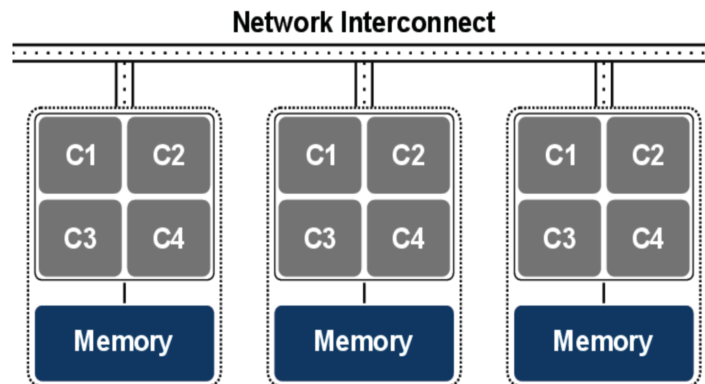


Figure 2.5: In hierarchical systems the distributed- and shared-memory model are combined. In the depicted case three UMA-based shared-memory nodes are connected via a network connection, however, NUMA systems can be used in a similar manner. Figure source: [2]

connected over fast network interface, as shown in Figure 2.5. The programming approach is also hybrid: using MPI for the inter-node communication and OpenMP for intra-node executions. The concept behind hybrid systems is broader than just multicore computers connected via network, it is also used to name any system layout that mixes available programming paradigms on different hardware layers. An example could be a cluster built from nodes that contains, besides the "usual" multicore processors, additional accelerator hardware, ranging from application-specific add-on cards to GPUs (graphics processing units), FPGAs (field-programmable gate arrays), ASICs (application specific integrated circuits), co-processors, etc.[32]

2.4 Study Case: NUMA Systems

NUMA arrived as a solution to bottlenecks produced by the intensive access to the single memory bus present on Symetric Multi-Processor (SMP) systems. To profit from the scalability provided by NUMA systems, applications need to be aware of the hardware they are running on and the memory allocation policies applied by the operating system. Memory pattern accesses that imply memory transfers from non-local banks will negatively impact the overall execution time due to the distance penalty of remote memory banks. A second issue that can also increase execution time is contention; applications employing large numbers of threads and storing all data in a single bank, generate a race between threads, congesting the connection links and downgrading access times -local and remote- to memory.

A processor and its respective memory bank is called NUMA domain or NUMA node,and

we will use these terms indistinctly along this thesis. Other clarification that needs to be done respect naming it is related to which part of the NUMA node we are referring to. For example if we say ”threads are bound to NUMA node 0” we mean the processor but if the phrase changes to ”data is placed on NUMA node 0” we mean the memory bank.

2.4.1 Our Infrastructure

As an example of NUMA systems, we can see the following two Figures (Figure 2.6 and Figure 2.7) that represent the two architectures we employed in this study, one manufactured by AMD and the other by Intel.

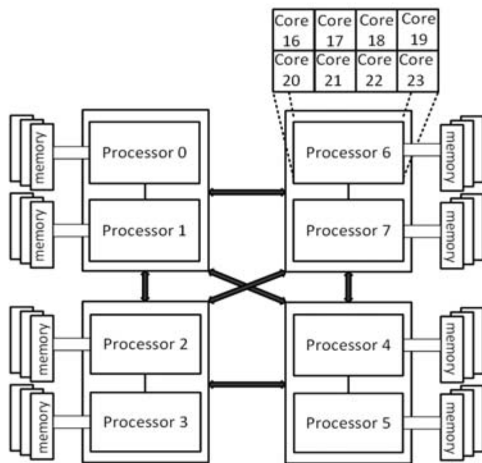


Figure 2.6: Schematic diagram of the AMD Opteron 6376 architecture (Abu-Dhabi)

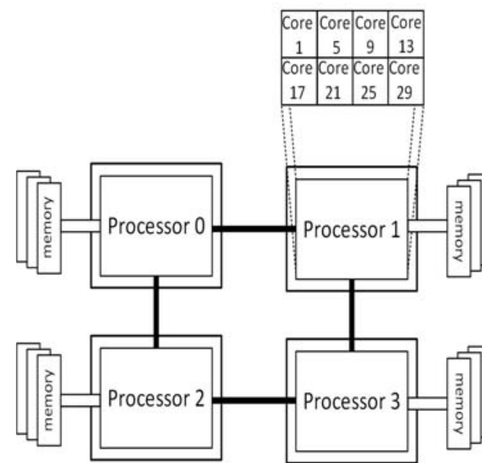


Figure 2.7: Schematic diagram of the Intel Xeon E5 4620 architecture (Sandy Bridge)

The first system, shown in Figure 2.6, is a four-socket AMD Opteron Processor 6376, with each socket containing 2 dies packaged onto a common substrate referred to as a Multi-Chip Module (MCM). Each die (processor) consists of 8 physical cores that share a 6 MB Last Level Cache (LLC) and a memory bank. Only one thread can be assigned to one core and, therefore, up to 64 threads can be executed simultaneously. The system has 128GB of memory, divided into 8 modules of 16GB DDR3 1600 MHz each. The second architecture (Figure 2.7) is an Intel Xeon CPU E5 4620 -also four-socket. Each socket contains an 8 core-processor and a 16 MB LLC. The total number of cores is 32. 64 threads can be executed simultaneously using HyperThread technology. This system also has a memory of 128 GB, but it is divided into 4 modules of 32GB DDR3. 1600MHz each.

Nodes are connected by links - HyperTransport (AMD) and QuickPath Interconnect (Intel). Memory bandwidth for both cases depend on the clock, the width of the interconnection links

(number of bits that can be sent in parallel in a single transfer), the data rate (single or double), and the actual bandwidth between memory controller and the DDR3 memory modules [18].

Allocation Policy

Linux operating system uses Node Local allocation as the default allocation policy when the system is up and running. Node Local allocation means that when a program is started on a CPU, the data requested by that program will be allocated to a memory bank corresponding to its local CPU.

Specifying memory policies for a process does not cause any memory allocation [37]. Allocation policy takes effect only when a page is first requested by a process. This is known as the first-touch policy, which refers to the fact that a page is allocated based on the effective allocation policy when a process first uses a page in some fashion. Despite the default Linux policy, a programmer can set an allocation policy for its program using a component of NUMA API [38] called libnuma. This user space shared library can be linked to applications and provides explicit control of allocation policies to user programs. The NUMA execution environment for a process can also be set up by using the *numactl* tool [38]. *Numactl* can be used to control process mapping to cpuset and restrict memory allocation to specific nodes without altering the program's source code.

A popular memory allocation policy is interleave where memory pages are allocated between selected nodes in a round-robin fashion. The idea behind it is that all used node all accessed equally as opposed to all the traffic being concentrated on one node.

2.5 Measuring performance

In this section, in first place we will enunciate the definitions of the metrics used during this work and later we will describe all the performance monitoring tools used and/or test for this work.

2.5.1 Definitions

When we think about computer performance we think of the effectiveness that a computer system complete a task. Depending on the context, 'effectiveness' may involve one or more of the following concepts:

- Short time for a given computational work

- High throughput (rate of processing work)
- Energetic efficiency of computing resource(s)
- High availability of the computing system or application
- Fast (or highly compact) data compression and decompression
- Short data transmission time

In HPC time is the most important criteria to define a system performance. The computer that performs the same amount of work in the shortest time is the fastest.[27] The execution time of a program is measured in seconds. In order to improve performance for a given application, it is necessary to monitor, analyze and tune the critical elements involved in the execution.

Execution Time

It is the measure that shows how long it takes to execute a process. It is defined as the interval between the start of the execution and the end of it. For parallel executions, the end of the execution is given when the last thread or parallel process that composes the application ends.

SpeedUp

It is a metric that characterizes the gain of running a program in parallel. Let T(n,1) be the run-time of the fastest known sequential algorithm and let T(n,p) be the run-time of the parallel algorithm executed on p processors, where n is the size of the input.

$$Speedup(n) = \frac{T(n,1)}{T(n,p)} \tag{2.1}$$

Parallel efficiency

It is obtained from the division between the Speedup and the number n of computing elements with which the application is executed (see equation 2.2). The value of the efficiency can vary between 0 and 1. Being 1 an ideal value that represents that the application is completely parallel.

$$Parallelefficiency = \frac{Speedup(n)}{n} \tag{2.2}$$

2.5.2 Performance tools

In this section we will describe the tools used to measured the behavior of benchmarks and the applications tested as validation that later we will be discussed in chapters 3 2. Two important features present in performance tools are: profiling and tracing. As stated Sameer Shende in [39].

"To understand the behavior of the parallel program, we must first make its behavior observable. To do this, we regard the execution of a program as a sequence of actions, each representing some significant activity such as the entry into a routine, the execution of a line of code, a message communication or a barrier synchronization. Performance analysis based on profiling and tracing involves three phases:

- instrumentation or modification of the program to generate performance data,
- measurement of interesting aspects of execution which generates the performance data and
- analysis of the performance data.

"

In the following subsections we will provide a brief description of the hardware counters used and the tools we rely on to do our research.

Hardware Counters

Hardware counter are extra logic added to the CPU that track low-level operations or events that happen within the processor [40]. Traditional hardware performance counters measure only values on a single core. A chip package has many resources which are package-wide and thus need a separate performance reporting mechanism. The shared socket-wide values are called Offcore events are per-core values on their way to the uncore [41].

For all the experiments performed in this work we measured the basic hardware given by perf tool [42]. These are:

- **context-switches:** when a task needs to be change for while ensuring that the tasks do not conflict. A high number of contest switches could impact negatively the performance of an application.
- **CPU-migrations:** when a task is moved from one computing environment to another.

- **page-faults:** when a running program accesses a memory page that is not currently mapped by the memory management unit
- **cycles :** usually, the time required for the execution of one simple processor operation such as an addition; this time is normally the reciprocal of the clock rate.
- **stalled-cycles-frontend:** measures all the cycles that are a waste because the CPU has to wait for resources (usually memory) or to finish long latency instructions.
- **stalled-cycles-backend :** measures all the cycles that are a waste because Front-End does not feed the Back End with micro-operations. This can mean that you have misses in the Instruction cache, or complex instructions that are not already decoded in the micro-op cache. Just-in-time compiled code usually expresses this behavior.
- **instructions :** the number of instructions executed
- **branches:** measures how many time the branch predictor tried to guess the correct branch.
- **branch-misses:** measures how many time the branch predictor failed to guess the correct branch.
- **LLC-loads-misses:** when the processor needs to fetch data that does not exist in the last level cache, so it brings it from memory.
- **cache-misses:** this event represents the number of memory access that could not be served by any of the cache.
- **seconds time elapsed:** entirely execution time.

PAPI

PAPI[43] provides two interfaces to the underlying counter hardware; a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. The low level PAPI interface deals with hardware events in groups called EventSets. EventSets reflect how the counters are most frequently used, such as taking simultaneous measurements of different hardware events and relating them to one another.

TAU

TAU [44] is a performance evaluation tool that supports parallel profiling and tracing. TAU has become the standard within the HPC world, not only because of the quality of its results but also, for the low overhead, plus it is regularly updated. When profiling a program, shows you how much (total) time was spent in each routine and tracing shows you when the events take place in each process along a timeline. TAU can Profiling and tracing can measure time as well as hardware performance counters from your CPU. TAU can extract

- It supports C++, C, UPC, Fortran, Python, and Java
- TAU runs on all HPC platforms and it is free (BSD style license)
- TAU has instrumentation, measurement and analysis tools
- To use TAU, you need to set a couple of environment variables and substitute the name of the compiler with a TAU shell scrip

As a result of this research, TAU added a NUMA feature to measure remote/local main memory access ratio for a given parallel application.

Likwid

Likwid (Like I Knew What I'm Doing) [42] is a tool suite to measure the different aspects that affect the performance of an application. It works on Intel and AMD processors on Linux systems. It offers several features, like measure performance counter, obtain architecture topology, core-biding etc. Likwid is oriented to performance programmers and offers a less complex interface. It is simple to use toolsuite of command line applications. While the focus of Likwid is on x86 processors some of the tools are portable and not limited to any specific architecture.

3
Analysis Methodology

”Test yourself on mankind. It is something that makes the doubter doubt, the believer believe. ”

- Franz Kafka

3.1 Introduction

The properties of the main memory system of an HPC architecture can have a drastic impact on the performance and efficiency of a program's execution. In order to provide guidance to a user or programmer we have created an empirical methodology. To characterize a NUMA architecture we rely on four main features, that we call *NUMA performance aspects*:

1. Topology and the native features of the architecture
2. Latency
3. Bandwidth
4. Contention degradation

The first indicator is information related to the hardware itself and it is provided by the manufacturer: the number of processors, cores and memory banks. From this information we calculate the theoretical maximum memory bandwidth and determine the restrictions due to the interconnection layout. The following 3 characteristics are latency (time elapsed since data is requested until it is retrieved by a CPU), bandwidth (the amount of data delivered per time over a physical link) and contention degradation (the factor on how a multithread race condition affects the performance of a parallel application).

Taking into account the concepts described above, we propose a methodology consisting of a series of steps to obtain the key information that allow us to characterize the architecture where the parallel programs will be executed. The result of carrying out these steps is to obtain what we will now define as *NUMA performance aspects*.

This information is valuable, mainly at the user level, since certain parameters can be tuned when executing the programs -such as the data allocation policy or the creation of independent instances- to obtain a better performance from the hierarchy of memory. But it could also be potentially used by a programmer because through the libnuma API tuning parameters can be applied at the code level. Without getting to the point of programming an algorithm that is architecture dependent, but considering *NUMA performance aspects* to better benefit from the hardware underlying.

Through greater detail we can know clearly the difference between what the manufacturer claims and the reality. Technical specifications are not always met. Being aware of the true potential of our system, help us know where we are standing before executing an parallel application. In this chapter we will describe how to obtain each of the indicators and understand how they characterize the architectures.

3.2 Calculating Theoretical Bandwidth

Table 3.1 was built with the respective manufacturer's specification [45] [46]

Table 3.1: Processor and memory information of the architectures used

	Intel E5-4620	AMD OPTERON 6376
# of Cores	8	16
# of Threads	16	16
Processor Base Frequency	2.20 GHz	2.30 GHz
Cache	16 MB SmartCache	2 x 8 MB (16 MB)
Bus Speed	7.2 GT/s QPI	5.2 GT/s HT
# of Links (QPI or HT)	2	4
Memory Types	DDR3 800/1066/1333	DDR3 1600
Max # of Memory Channels	4	4

As a first step, we need to calculate the maximum theoretical bandwidth to know what is the upper limit of our system. It is worth noting that these values are rarely reached. To calculate this value it is considered that the scenario is the best possible: that is to say that the program is compiled in an optimal way, that the prefetchers are able to anticipate most data request, the access pattern can be predicted, etc.

$$Max\ BW = \frac{Memory\ Frequency\ x\ \#\ of\ Channels\ x\ Word\ Size}{Byte} \tag{3.1}$$

Using the information in table 3.1 and formula 3.1 we obtain the following numbers for both architectures:

- Intel: 1333 x 4 x 64/8 = 42.6 GB/s
- AMD: 1600 x 4 x 64/8 = 51.2 GB/s

These reflect the value of the local memory access. Depending on the amount of sockets of the system, we could theoretically increase this value. In our case both architectures used have 4 sockets. So potentially a maximum memory bandwidth (Max BW) of 170.4 GB/s (Intel) and 204.8 GB/s (AMD) if 4 sockets are accessing data allocated in their respective memory banks independently. This is a unrealistic scenario, due it is highly unlikely to avoid out-of-socket-accesses that are restricted to the interconnection bus speed. It is possible to

calculate interconnection bus bandwidth from table 3.1 and considering that one hop accesses are performed using 16-bit links. Values are:

- Intel: 7.2 x 2 x 16/8 = 28.8 GB/s
- AMD: 5.2 x 2 x 16/8 = 20.6 GB/s

Another important fact to consider when calculating the theoretical values of memory bandwidths is that they are based on the assumption that all channels are being occupied. For example, in the case of AMD, there are 4 channels. To aspire to achieve the theoretical bandwidth there should be a physical memory module in each DIMM slot of the channel. A user can retrieve the information of how many channels are being used, using linux command *dmidecode –type memory*.

Calculate the bandwidth is a good practice in order to know the optimal capacity of the system to use. Although it should be a more widespread practice among HPC system users, it is understandable that it entails some complexity.

3.3 Measuring Latency

We used the lmbench [21] to measure latency, specifically the lat_mem_rd module. This test consists of measure memory read latency changing the memory sizes and strides. The entire memory hierarchy is measured, including onboard cache latency, LLC cache latency and main memory latency. The module lat_mem_rd uses a double nested loop. The outer loop is the stride size. The inner loop is the array size. For each array size, the benchmark creates a ring of pointers that point backward one stride. And then it goes through each element of the array in turn by doing

$$p = (char**)*p \tag{3.2}$$

The size of the array starts in 512 bytes up to final size provided as parameter. We used as final size 2 GB, large enough to test the main memory. For the small sizes, the cache will have an effect, and the loads will be much faster as the array increases its size and stride the latency will reflect that.

Latency

In Figure 3.1 we can see that the processor has a 16kB L1d, 2MB L2 and 8MB L3. We do not see sharp edges in the transition from one level to the other because the caches are used by

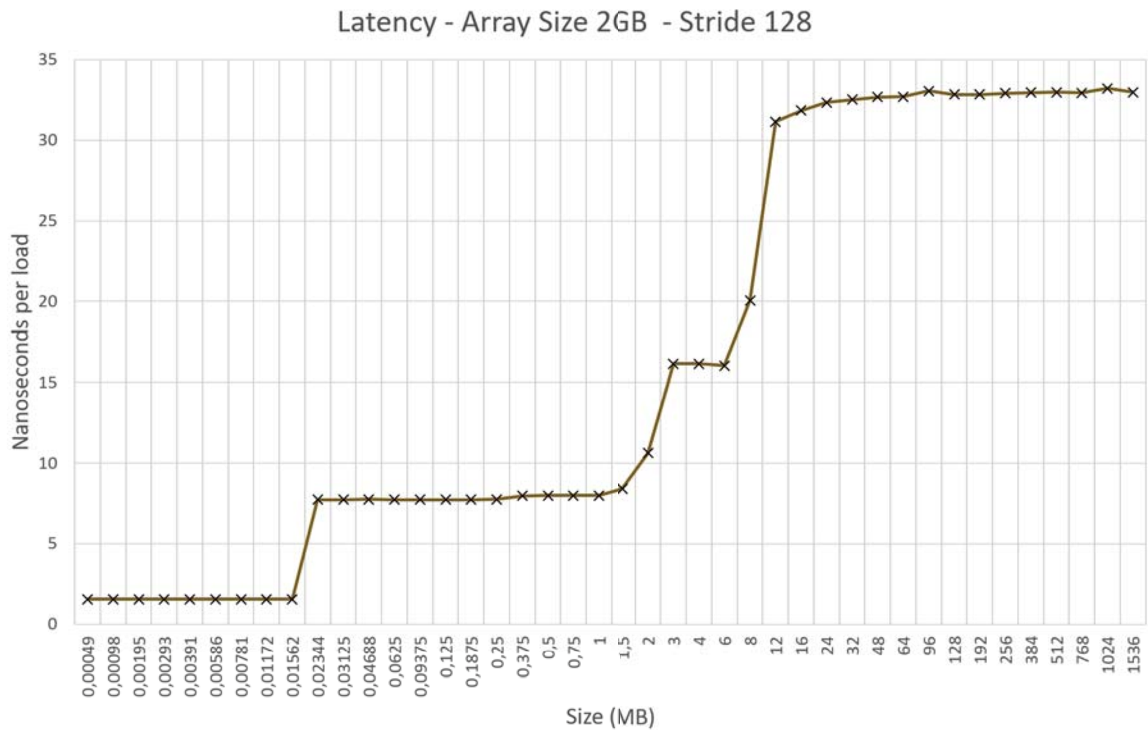


Figure 3.1: Latency profile of AMD Opteron 6376 using lmbench

other parts of the system as well and so the cache is not exclusively available for the program data. Specifically the L2, L3 cache is a unified cache and also used for the instructions. [47]

3.4 Measuring Bandwidth

To measure the memory bandwidth we use the well-known STREAM benchmark [19][20]. It measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for four simple vector kernels:

The first test (COPY) is the simplest - and the fastest - of all since it does not involve any arithmetic operation. It simply brings 2 from memory a(i) and b(i) and performs an update operation.

The second kernel (SCALE) is a modification of the first one, where a simple arithmetic multiplication operation is added. It consists of obtaining two values of the memory, a(i) and b(i), but multiply b(i) by a scalar before writing it in a (i). It is a simple scalar operation but more complex operations are created from it. The performance of this test can be used as an indicator of the performance of more complex operations in comparison.

name	kernel
COPY	a(i) = b(i)
SCALE	a(i) = q*b(i)
SUM	a(i) = b(i) + c(i)
TRIAD	a(i) = b(i) + q*c(i)

Table 3.2: STREAM's vector kernels

The third kernel (SUM), adds a third operand. For larger arrays, due it fetches 3 values, this can potentially fill a processor pipeline, so the memory bandwidth can be tested by filling the processor pipeline or the performance when the pipeline is full. This benchmark tries to approximate the actual behavior of a real application.

The fourth one (TRIAD), allows chained, overlapped or fused, multiple-add operations. It builds on the SUM benchmark by adding an arithmetic operation to one of the fetched array values. Given that fused multiple-add operations (FMA) are important operations in many basic computations, such as dot products, matrix multiplication and polynomial evaluations. This benchmark can be directly associated with application performance. The FMA operation has its own instruction set now and is usually done in hardware. Consequently, feeding such hardware operations with data can be extremely important – hence, the usefulness of the TRIAD memory bandwidth benchmark [48].

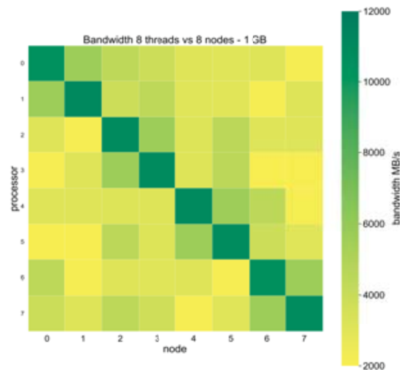


Figure 3.2: All possible bandwidth values for one-processor execution on AMD system

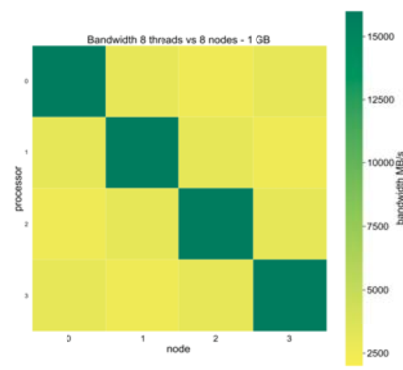


Figure 3.3: All possible bandwidth values for one-processor execution on Intel system

NUMA system has a performance sensitivity to the placement of threads and data. Changing these parameters can give very different outcomes, however exploring all possibilities

can be unfeasible. The AMD system has 8 NUMA domains this means that if an 8-threads-program is executed (using one processor to bind all threads and one memory bank to host all data) we have 64 possibilities of parameters of execution. We calculate it considering all the variations allowing repetition and taking into account the order ($V'(8) = 8^2 = 64$). But as the number of threads involved increases, also it does the possible combinations. In a 16 threads scenario we would need 2 processors, in this case we do not allow repetitions for the processors, however the combinations increase to 224,

$$V_k(n) = \frac{n!}{k!\,(n-k)!\,)} = \frac{8!}{2!\,(8-2)!\,)} = 28 \;\; \forall \;\; NUMA\ domains\ \varepsilon\ (0,7)$$

and employing the same formula we can see that for 32 threads the number reaches 560 combinations. Running a syntactic program, such as STREAM, gives us the opportunity to test most combinations and profiling the memory behavior without the need of executing higher time consuming applications.

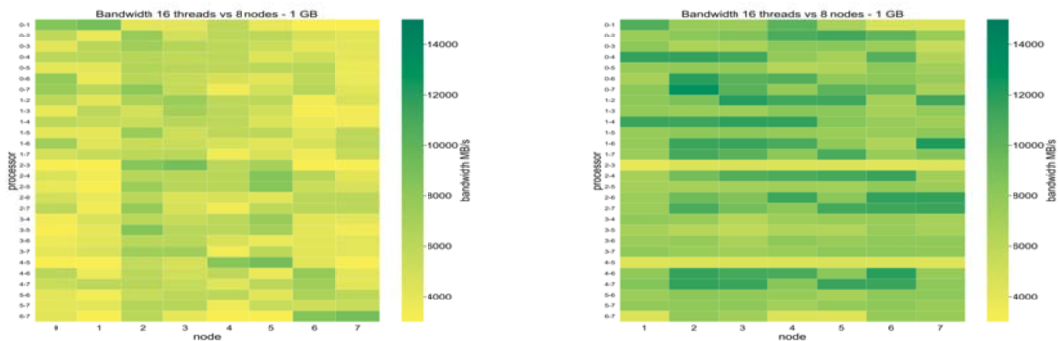
Algorithm 1 Execution of STREAM with 16 threads using numactl API (AMD)

```
Data: NUMA_domains = 0, 1, 2, 3, 4, 5, 6, 7
for processor1 in NUMA_domains do
    for processor0 in NUMA_domains do
        if $processor0 < $processor1 then
            for memory_bank in (1, 2, 3, 4, 5, 6, 7) do
                numactl —cpunodebind=$processor0, $processor1 —interleave=0, $node
                 ./stream
            end
        end
    end
end
```

The profiling process consists of testing all possible combinations for the 8, 16 and 32 threads. In figures below we can appreciate in a very clear way how the maximum bandwidth is reached when the data is fetched from the local memory banks, pointed out by the dark green diagonal. In Figure 3.2 (AMD) we have 8 NUMA domains, as a result we have an 8x8 matrix. Light green areas surround the diagonal, representing that accesses to processors on the same chip have an acceptable memory bandwidth. The matrix reflecting AMD system is not symmetric, as explained in chapter 2, the interconnection bus have different bit wide links. In Figure 3.2 (Intel) we have a 4x4 matrix and in this case we can see a more remarked

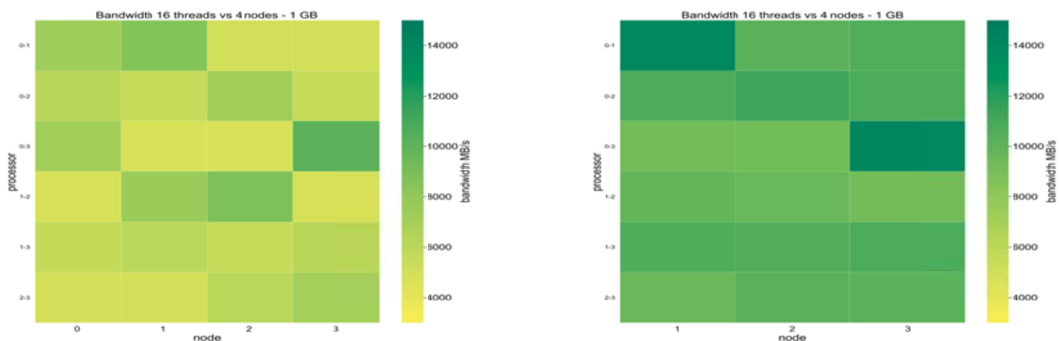
difference between local allocation and remote nodes. In AMD, memory bandwidth drops to half when data is changed from local to on-chip neighbour (1 HOP), from 11000 MB/s to 5700 MB/s and reduced 5.25x when the farthest distance-combination is employed. In Intel, on the other hand, the memory distance-penalty is even higher. When the data is local, the BW is 16700 MB/s drops 5.5x when it is 1-Hop distance and 6.4x when is 2-HOP distance.

For the following measures of BW using 16 and 32 threads we also considered the allocation policy. Due that these scenarios we could no longer guarantee that the accesses were 100% local. As remote accesses are unavoidable at this point, we wondered if it was possible to balance the memory accesses by allocating the data onto two memory banks. The data were interleaved between bank 0 and the rest of banks. Script 1 shows the steps followed to benchmarking the combinations.



(a) BW values for 16 threads on AMD system with membind policy

(b) BW values for 16 threads on AMD system with interleave policy

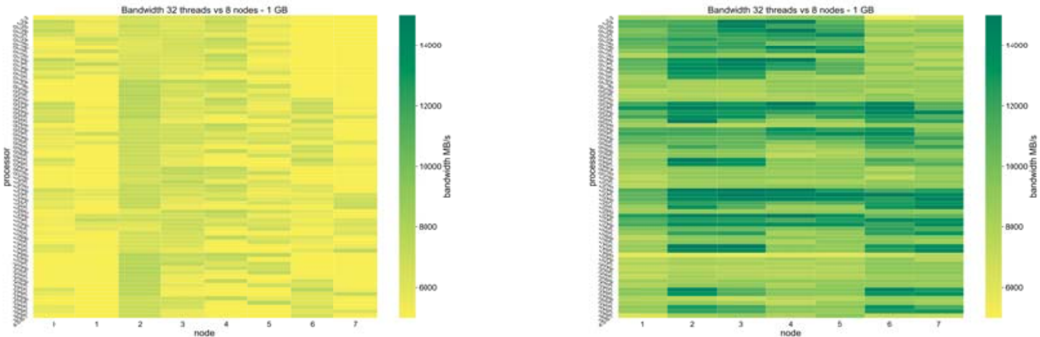


(c) BW values for 16 threads on Intel system with membind policy

(d) BW values for 16 threads on Intel system with interleave policy

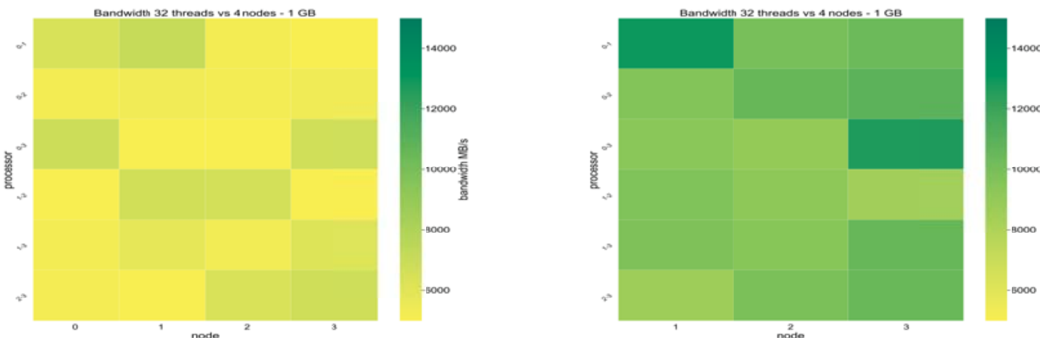
Figure 3.4: BW measures for all systems using 16 threads

It is clear that many of the combinations explored are sub-optimal and it does not make



(a) BW values for 32 threads on AMD system with membind policy

(b) BW values for 32 threads on AMD system with interleave policy



(c) BW values for 32 threads on Intel system with membind policy

(d) BW values for 32 threads on Intel system with interleave policy

Figure 3.5: BW measures for all systems using 32 threads

sense to take them into account when executing a program but the objective of this stage of the methodology is to show the sensitivity of the memory system to the allocation policies. In Figure 3.4a again can be observed the diagonal marked by local accesses. Now it is represented by two cells, side by side. When the number of NUMA node matches the memory bank the memory throughput is maximum, however this does not happen for Intel (Figure 3.4c), because no matter which set of combination do we test in either case a remote access is happening, if the remote access is 1-HOP distance then the cells are greener but even then the difference of bandwidth is not significant.

When interleave policy is used, it is clear the increment of the global memory throughput of the system, it is easy to observe as the graphics get more green on both cases (Figures 3.4d and 3.4b). Curiously the last presents two straight yellow lines at when using nodes 2-3 and 4-5, this is related to the fact mentioned above, the links that link the nodes are

not all bidirectional, the access relationship between a processor and a memory bank is not bijective. In other words, accesses from the processor x to the bank y is not always equivalent to accessing the processor y bank x.

In the last scenario, it can be seen that as the number of threads increases, the bandwidth becomes more restricted by inter-node accesses. But as in the previous case, the global bandwidth of the system increases when the data is distributed using interleave instead of concentrating all the accesses to a single bank.

3.5 Measuring Contention

To measure contention we used a benchmark developed by James Brock and modified by Andreas Klöckner. We also made some changes to allow certain parameters at the time of execution, which we will detail below. This benchmark, unlike STREAM, consists in generating random accesses to an array on memory, trying to trick the hardware prefetch and obtain a latency close to the true one. The random accesses are produced by multiplying the iterator by a prime number which is comfortably larger that the cache line. The following formula ensures that each byte is incremented exactly once.

$$*(((char*)x)+((j*1009)\%N))+=1 \tag{3.3}$$

This benchmark automatically creates the maximum number of threads possible for a given system. Each thread is pinned to its matching core. It allocates the array on node 0 and then each thread traverses the array using the formula 3.3, in three different modes: sequential (one thread at the time), two simultaneously, and all at once. As each thread ends up accessing the array, the bandwidth is calculated from all possible positions towards the memory bank 0 with equation 3.4.

$$bandwidth^* = \frac{array_size * ntrips * cache_line_size}{time} \tag{3.4}$$

We have introduced small variations to this benchmark that allowed us to pass through as parameter the number of memory bank where the array is allocated, in addition to adapting the pin process of the threads to the cores for both architectures used (AMD and INTEL) since the numbering of the cores change for each manufacturer.

With these variations we can calculate the bandwidth from any core to any memory bank, and we can also see how the final bandwidth is affected the intensive use of interconnection

buses as the number of threads increases.

These test show how they affect the fact that many threads simultaneously try to access the same memory bank. For this testbed we used the cases of 8, 16, 32, 48 and 64 threads. The behavior concurs with what was explained in the previous section, for AMD Figure 3.6 with 8 threads (blue line) you can see the maximum bandwidth is achieved when data is completely local, this only can achieved with 8 threads. In the case of 16 threads (orange line) we will have remote accesses, so the BW even in the best situation is 40 % worse per thread. This behavior can be extrapolated by increasing the number of threads obtaining each time a minor bandwidth memory per thread. For 64 threads the behavior is practically constant, no matter where the data is stored, it will always have the maximum access time (unless data are spread system-wide). For Intel, Figure 3.7 the behavior is quite similar, with the particularity that the step change between the usage of only local node and combination of local and remote is more clearly exposed for 8 threads (blue line).

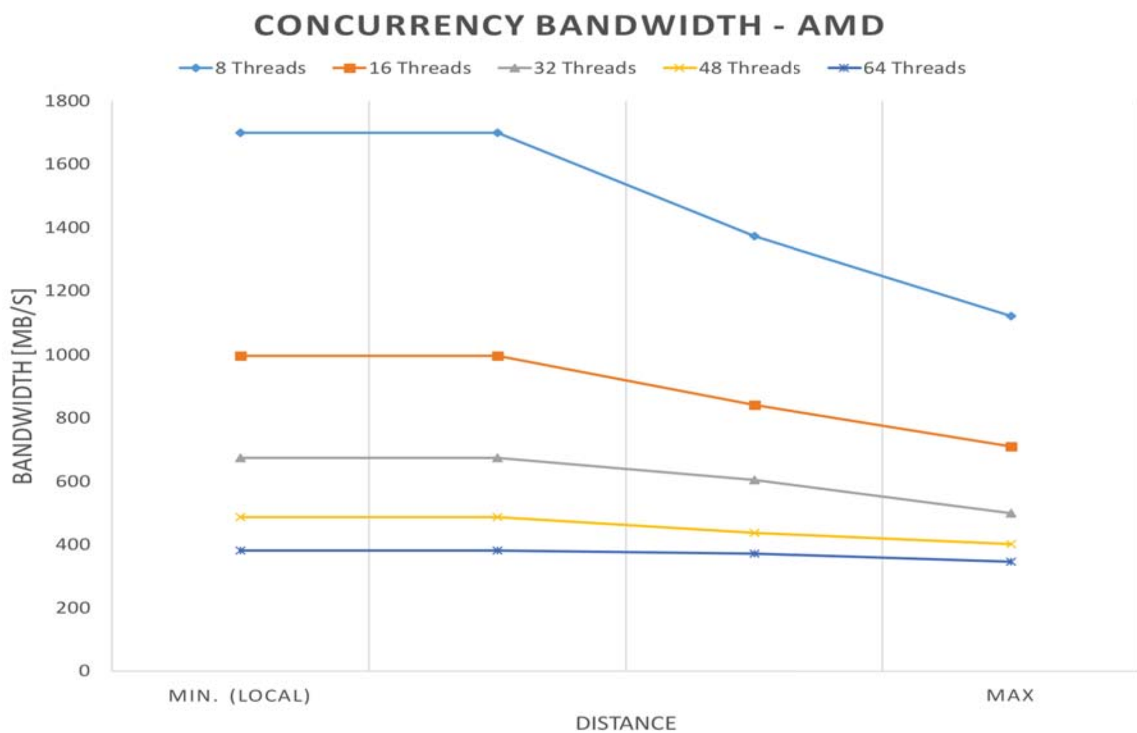


Figure 3.6: Concurrency stress test on AMD system

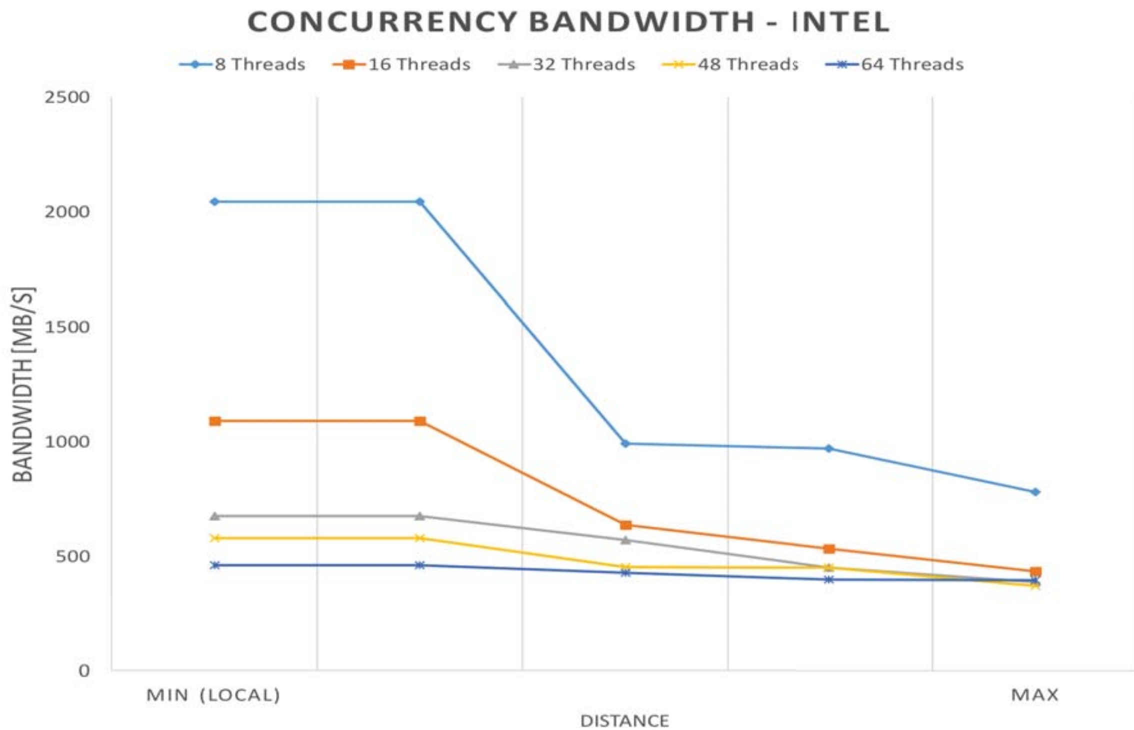


Figure 3.7: Concurrency stress test on Intel system

3.6 Creating a guideline

In this chapter we presented a series of steps to characterize two common commercial NUMA architectures. And could be extrapolated to other NUMA systems. It was carried out using well-known benchmarks such as STREAM and lmbench, and also the manufacturer's technical specification. The steps are easy to implement and quickly an end user could get the following *NUMA performance aspects* from the system:

- First, the theoretical ceiling. Know what is the maximum value of memory bandwidth that we can obtained. At this point we also highlight which are the relevant data in the technical specification that allow us to calculate said bandwidth.
- The next step is to know the latency through lmbench. The memory access latency is a complementary information that outlines the memory hierarchy and quantifies the cost of not using the cache system properly.
- The third step is to know the parallel bandwidth. This is where the user can really see what a NUMA architecture implies. Using STREAM, with the presented execution

algorithm, we make a sweep of all possible combinations of data placement and threads, getting a variety of BW. As the number of threads increases it does also the number of possible place, creating subsets that are optimal for execution and need to be taken into account when running a parallel program. And also that the implementation of interleave policy can have a strong impact in it.

- In this last stage, the effect of contention is analyzed. It is perceived a trade-off game between the number of threads and total system BW. How the performance is degraded by each single thread but increases throughput at the system level to a certain limit.

All these steps were manually implemented through scripts that can be consulted in our github repository:

https://github.com/jlenis/NUMA_characterization_-_exec_strategies

In order for a program to run on NUMA architecture, there is no need to make any adjustments to the code, unlike when accelerators or GPUs are used. However, the user must be aware that she/he is dealing with a hybrid system. We do not advocate for notion that a programmer should write code for a given architecture in mind, however all the complexity that the programmer is saving in her/his code is transferred to the final user as execution time penalty. We propose a guideline at user level, acquiring the hardware insights from the *NUMA performance counter* plus his own knowledge of the applications he wants to run. We are now at the point (4) of our methodology Figure 3.8.

For the two NUMA systems we characterize, these are the insights:

- Both system are extreme sensitive to memory allocation policy .
- On paper AMD system has a better BW however Intel benchmarked better results.
- Both are sensitive to contention degradation when data is centralized.
- Intel has better local access BW but suffers from a harder drawback when data is accessed remotely.
- In AMD the distance or HOPS that dictated by the amount of links between nodes than the layout.

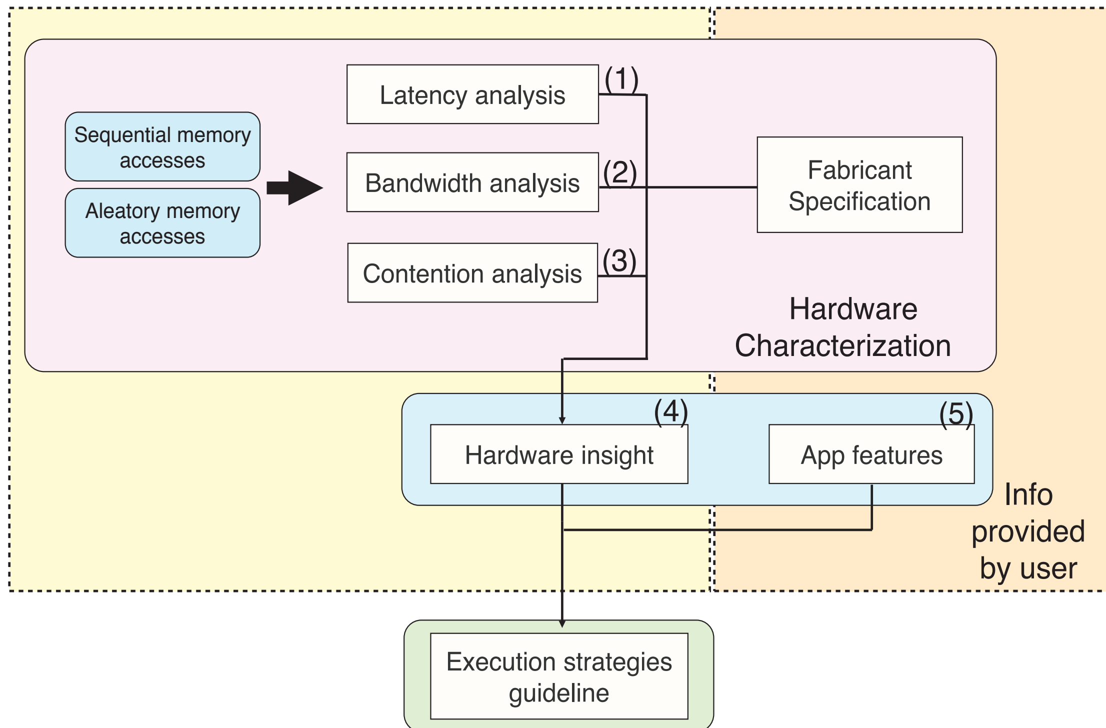


Figure 3.8: Methodology to obtain a guideline of NUMA execution strategies

We believe that there is a sweet spot between the two extremes points of: programming an application dependent of the architecture and not considering the hardware features at all, a series of parameters adjustments that can be done at user lever to reduce the drawback of NUMA effects. To validate this hypothesis we want to evaluate the comportment of real parallel memory-bound applications on the systems studied. We opted for sequencing aligners due the huge relevance of genomic workflows in the scientific community. Applications have several algorithmic phase, and even though that are labeled as mainly memory-bound they have also an important computational phase and a strong I/O interaction. Everything related to the sequencing aligners and its background will be properly explained in next chapter.

4
Sequencing aligners

”Don’t despair, not even over the fact that you don’t despair. Just when everything seems over with, new forces come marching up, and precisely that means that you are alive. ”

- Franz Kafka

4.1 Introduction

Deciphering the DNA through its variations has become a essential study that provides relevant results to all branches of biology and medicine. The advancement of technology has resulted in a reduction of costs in the so-called new generation sequencers, enabling to a large part of the scientific community have large volumes of data for research.

In this chapter we will explain conceptually basic principles of the human genome study. In a first instance, an global introduction to the subject and then detailing each of the stages involved in the process, to provide an extensive context to sequence alignment. Finally we will describe all the short-read mappers used as a test bed to validate the NUMA execution guideline.

4.2 Motivation

The information contained in the chromosomes is what determines the characteristics of human beings, from our eye color to the disorders that we may be prone to suffer. Among the many possibilities that arise from this field of study, perhaps one of the most the resonance of personalized medicine. The variations in the genome of individuals can be associated with certain diseases and its analysis enables the creation of treatments adapted to the specific characteristics of a patient's genome.

The genomic studies that have been carried out since the middle of the 70s were possible thanks to a set of biochemical techniques that obtained sequences of nucleotide bases from of a organic tissue. These techniques, known as sequencing, evolved rapidly to this day. Since approximately ten years new tools were introduced to the scientific community of sequencing, developed mainly by Illumnia, SOLiD Systems, 454 Roche and HeliScope. These **N**ew **G**eneration **S**equencers (NGS) have drastically improved the amount and quality of the output data, and significantly reducing the costs.

As a consequence, not only the laboratories with the latest technology have this equipment, but it is increasingly common that institutes, hospitals and universities acquire sequencers that generate massive data sets. Taking into account that many of these institutions have modest clusters composed of common PCs (Commodities PC) and that the sequencers mentioned above they have the capacity to produce data of the order of one billion of base pairs per day [6], the challenge is then in which the computational applications have sufficient capacity to deal with the analysis of this data in an efficient way.

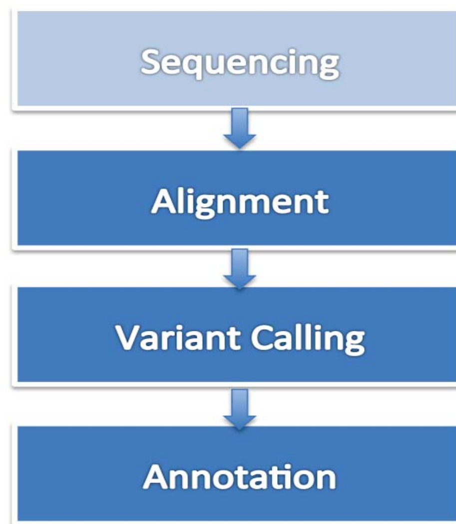


Figure 4.1: Standard variant calling workflow

4.3 NGS Workflows

Genome investigations are mainly based on the study of variations in DNA sequences. These variations are can identify, using the reference genome as a comparison human [49], or comparing sets of genomes of individuals to each other. According to the type of variation and how it is expressed in the human genome, it is possible to relate them to certain diseases or disorders. Be able to determine if these variations can be associated to hereditary disorders (Mendelian), to complex diseases or somatic mutations, among others, is the result of a process analysis of the data obtained with NGS.

The entire process or workflow is very complex as it depends on many programs and databases, and involves the handling of huge amounts of heterogeneous data. Thanks to the growing interest in projects NGS, it is understandable the appearance of numerous tools (private and opensource) that allow to perform specific tasks within the workflow [50].

The choice of tools to use depends largely on the biological results that are sought, so workflows are rarely the same. However, you can perform an abstraction and say that most of the workflows share the same conceptual stages.

4.3.1 Sequencing

The sequencing stage consists in the transformation of a tissue biological in DNA sequences, called readings (reads). The length of the reads are measured in the amount of nucleotide bases it has. These fragments are combined in files of fastQ format. Some of the techniques

used in this process are:

(a) Pyrosequencing.

(b) Sequencing by synthesis.

(c) Sequencing by ligation.

(d) Sequencing of a single molecule, real-time sequencing.

(e) Secuenciación Post-light.

The sequencing process follows a standard protocol for its realization, however certain parameters can be customized to obtain reads with certain characteristics. These characteristics affect to all stages of the workflow:

Reads length

During the sequencing it is possible to specify the number of readings. The more extensive readings are more reliable however they imply a higher cost of generation and analysis. Normally they range from dozens up to several hundred bases.

Single-end y Paired-end reads

In the single-end readings, the sequencer reads from one end of one molecule to the other, generating a sequence of even bases. In the paired-end readings, the sequencer starts at the end of a fragment and ends its reading in a specified number of bases and there begins a new reading from the opposite end in the direction opposite to the first. As the molecules have a length of two times greater than an average sequencer reading (Illumina in this case), the paired-end readings do not overlap. The data sequenced in paired-end readings come in two files. One that saves the sequences in one direction and the other stores in the order reverse.

Paired-end readings are better for identifying positions relative to numerous readings within the genome. allowing effective way to solve structural arrangements such as insertion, the elimination or investments of genes.

Coverage

The coverage measures the number of times a specific site in the genome It is sequenced during the sequencing process. It can be calculated as:

$$coverage = \text{N}^{\text{o}} \text{ total of generated bases/Size sequenced genome}$$

For example, 30x coverage, which in some cases can be considered high coverage, means that on average each base is reported 30 times This does not mean that it is made as an equal bread basket. Some bases can be sequenced 70 times and others 1 or 2, or even any arbitrary number of times. The greater the coverage, the more reliable the sequencing.

Genome vs. Exome

Human genome sequencing has reduced its costs with the introduction of new generation technology, however, remains expensive on a large scale. That's why, many researchers are inclined to use the exome sequencing. The exome is the coding part of the genome, It represents approximately 1.5 % of it. Being represented in smaller files, exomes allow work with coverage greater than the genomes. The exome sequencing was already used to identify molecular defects of a single gene, disorders heterogeneous and improve the accuracy of patient diagnosis.

4.3.2 Alignment

The alignment or alignment stage is the process in which determines the position corresponding to each reading of nucleotide bases. There are two main approaches to solve the alignment problem, which is based on using a reference genome (assembled by mapping) or dispense with it (assembled *de novo*). The de novo assembly consists of in determining the order of the fragments or readings, using overlapping of common regions to join them. The continuous sequence of readings it's called "contig". The contig length that an assembler can produce is limited by the number of repeated sequences, polymorphisms, missing data and errors [51]. In the assembly by mapping, each reading is compared to a reference genome, and there that your position is determined. It is clear that when they are investigated new species the use of denovo alignment is necessary, since there is no reference genome. But in the case of study of the human being, the decision on which method to choose becomes more complex In denovo, all the readings are compared to each other, as a result, a much more precise process is obtained [52] but computationally more demanding. That is harmed by readings too short, as this increases the possibility that more readings are overlaid in a non-univocal way. That's why for short readings from 35bp to 100bp, the use of assemblers predominates by mapping that are faster, but as a counterpart the assemblers that use a reference genome have polarized

results since readings that look more like the reference sequence are more likely to be mapped successfully compared to readings that contains valid differences. [53]

Using the assembly by mapping would imply that to discover the position correct one reading should be compared to three billion of possible positions within the human genome. The current programs they use indexing techniques to optimize their mapping algorithms. The two most commonly used approaches [54] by The aligner programs are:

Hash table based algorithms (paced-seed indexing):

In this method in each position in the reference is fragmented into four pieces of equal size called seeds and said seeds they are paired and saved in a query tables. As results you get 6 combinations of seed pairs for possible position. Each reading is also fragmented in the same way where the seed pairs are used as a key to search for positions that correspond in the reference algorithm.

Preffix/Suffix Tree based algorithms (Burrows-Wheeler Transformed):

Use the Burrows-Wheeler transform to store representations of the reference genome whose memory utilization is very efficient. Implementations of this method use about 2GB of RAM for represent the complete genome compared to the technique of spaced seeds that requires approximately 50GB. Readings are aligned character by character, from right to left using as a reference the transformed genome. Each new character aligned allows you to reduce the list of possible positions that may correspond to you to reading. This is a significantly more complex algorithm but it runs faster.

4.3.3 Variant calling

In a few words at this stage it is about determining the possible variants that an individual presents with respect to a specific reference, and see how they relate to the biological phenomenon that is being investigated

The set of variations that occur in a single nucleotide base they are known as SNV (Single Nucleotide Variation), within this category are the SNP single nucleotide polymorphisms (Single Nucleotide Polymorphism) and the insertions / deletions of a single INDELs base (InsertionDeletion). The second set of variations are structural variations SV (Structural Variation), which involve to large blocks (more than 100 bases) of DNA. In this category there are duplication, INDELS, investments, translocations, among others [55].

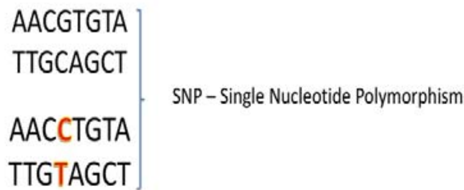


Figure 4.2: Single-nucleotide polymorphism

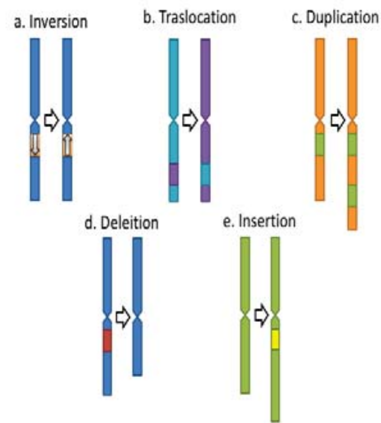


Figure 4.3: Types of SNP

Before the emergence of NGS technologies, genome studies human beings positioned the SNP variations as the main source of the phenotypic and genetic variations. However, in the past years has been shown, that there are large structural variations in the human genome [56]. The evidence indicates that the variations can be associated with diversity and susceptibility to diseases, despite the fact that most structural variations they are outside the coding region. The tools for variant, differ according to the types of variations what you want to search But there is also another classification that is born of the biological phenomenon that you want to analyze:

1) **Determination of germline variants,** which is central to find the causes of rare and characteristic diseases genetic It is obtained by comparing the tissue of the individual in question with the reference genome.

2) **Determination of somatic variants,** that are searched for diseases like cancer. But in this case a comparison is made between the results of the sequencing of healthy tissues with those of carcinogenic tissues of the same individual.

4.3.4 Annotation

After identifying the variations, it is necessary to be able to predict its functional impact. Expressed otherwise associating the variation to its corresponding biological information. For that the tools who make the annotations, they usually compare the variations found with public databases like dbSNP [57] that provides more than 20,000 validated human SNPs, there are other bases of data such as(HGMD Human Gen Mutation Database)[58] that contains more of 76,000 mutations of approximately approximately 2900 genes,or the COSMIC (Catalog

of Somatic Mutation in Cancer)[59] at the institute Sanger, stores ˜25000 human somatic mutations related to cancer [53].

4.4 Sequence Aligners

Sequence aligners -or aligners, for the sake of simplicity- can be classified into two main groups: based on hash tables or based on Suffix/Preffix trie. In ”A survey of sequence alignment algorithms for next-generation sequencing” Li et. al[60],define very precisely each:

- **Hash Table:** ”The idea of hash table indexing can be tracked back to BLAST. All hash table based algorithms essentially follow the same seed-and-extend paradigm. BLAST keeps the position of each k-mer (k = 11 by default) subsequence of the query in a hash table with the k-mer sequence being the key, and scans the database sequences for k-mer exact matches, called seeds, by looking up the hash table.”

  SNAP is an example of a hash table-based aligner, where given a read to align draws multiple substrings of length s from it and performs an exact look-up in the hash index to find locations in the database that contain the same substrings. It then computes the edit distance between the read and each of these candidate locations to find the best alignment.

- **Suffix Trie:** ”reduce the inexact matching problem to the exact matching problem and implicitly involve two steps: identifying exact matches and building inexact alignments supported by exact matches. To find exact matches, these algorithms rely on a certain representation of suffix/prefix trie”

  Examples of BWT-based aligners are BWA, BOWTIE2 and GEM. Hash tables are a straight forward algorithm and are very easy to implement, but memory consumption is high; BWT algorithms, on the other hand, are complex to implement but have low memory requirements and are significantly faster [54].

On the other hand, BWT is an efficient data indexing technique that maintains a relatively small memory footprint when searching through a given data block. BWT is used to transform the reference genome into an FM-index, and, as a consequence, the look-up performance of the algorithm improves for the cases where a single read matches multiple locations ”The advantage of using a trie is that alignment to multiple identical copies of a substring in the reference is only needed to be done once because these identical copies collapse on a single

path in the trie, whereas with a typical hash table index, an alignment must be performed for each copy."

The computational time required by an aligner to map a given set of sequences and the computer memory required are critical characteristics, even for aligners based on BWT. If an aligner is extremely fast but the computer hardware available for performing a given analysis does not have enough memory to run it, then the aligner is not very useful. Similarly, an aligner is not useful either if it has low memory requirements but it is very slow. Hence, ideally, an aligner should be able to balance speed and memory usage while reporting the desired mappings [61]. In [24], Misale et al. define three distinguishing features among the parallelization of sequence aligners:

1. There is a reference data structure indexed (in our study, the human genome reference). Typically this is read-only data.

2. There is a set of reads that can be mapped onto the reference independently.

3. The result consists in populating a shared data structure.

From a high level point of view, this is the behavior of all the aligners that we used in this study. Therefore, continuous access to the single shared data structure -index- by all threads can increase memory performance degradation. Additionally, read mapping exhibits poor locality characteristics: when a particular section of the reference index is brought to the local cache of a given core, subsequent reads usually require a completely different section of the reference index and, hence, cache reuse is low.

4.4.1 Burrows Wheeler Aligner

Burrows Wheeler Aligner (BWA) is one of the most used short-read mapper by the genomic community. BWA uses the BWT on a reference genome to create an index of it, and two auxiliary data structures. These data structures are used to calculate the range where a read sequence matches the reference genome, so they are accessed every time a read is analyzed. Once the index of the reference genome is allocated in memory, BWA takes each individual read and calculates its suffix array coordinates using BWT and the reference genome index.There is no dependency between reads, so several reads can be aligned in parallel. In fact, BWA allows multithreaded execution using pthreads. Each thread executes a sequential version of the core mapping algorithm, while main data structures are allocated

at the beginning of execution before the rest of threads are created [25]. For this work we centered on BWA original algorithm *aln* [6].

4.4.2 Bowtie2

Bowtie 2 is tool for aligning sequencing reads to long reference sequences. It is particularly good at aligning reads of about 50 up to 100s of characters to relatively long genomes. Bowtie 2 indexes the genome with an FM Index (based on the Burrows-Wheeler Transform or BWT) to keep its memory footprint small: for the human genome, its memory footprint is typically around 3.2 gigabytes of RAM. Bowtie 2 supports gapped, local, and paired-end alignment modes. Multiple processors can be used simultaneously to achieve greater alignment speed.It runs on the command line under Windows, Mac OS X and Linux.

4.4.3 Genome Multitool 3

Genome Multitool 3 (GEM3) is a high-performance mapping tool for aligning sequenced reads against large reference genomes (e.g. human genome). In particular, it is designed to obtain best results when mapping sequences up to 1K bases long. GEM3 indexes the reference genome using a custom FM-Index design, and performs an adaptive gapped search based on the characteristics of the input and the user settings. GEM3 allows performing complete searches that find all possible matches according to the user criteria. It enables the user to perform searches looking for the first-match/best-match, all-first-n matches, and all-matches. GEM3 supports single-end, and paired-end mapping modes. Also, it supports both global-alignment and local-alignment models for different error models It offers out-of-the-box multithreaded mode to exploit several cores processors and achieve greater speed, without affecting the relative order of the reads. GEM3 is distributed under GPLv3, and runs on command line under Mac OSX and linux.

4.4.4 Scalable Nucleotide Alignment Program

Scalable Nucleotide Alignment Program (SNAP) is a sequence aligner that is 3-20x faster and just as accurate as existing tools like BWA-mem or Bowtie2. It runs on commodity x86 processors, and supports a error model that lets it cheaply match reads with more differences from the reference than other tools. SNAP search algorithm is based on hash table. SNAP fast performance is related to the larger length of the reads that it processes which allow for fast

hash-based location of reads using larger ”seed” sequences. It also meant to be executed on machines withat least 10 GB of RAM to take profit of the increased server memories, which allow trading memory to save CPU time.

5
Experimentation

”You can hold yourself back from the sufferings of the world, that is something you are free to do and it accords with your nature, but perhaps this very holding back is the one suffering you could avoid. ”

- Franz Kafka

5.1 Performance of sequence aligners on NUMA systems

In Chapter 3 we presented a series of studies that can be used to characterize NUMA architectures. From there we analyzed the *NUMA performance aspects* in order to understand the functioning of the system. We analyze how this acquired knowledge can give us answers to possible problems that could be experienced by applications that are not NUMA-aware but that are executed on such systems. In this chapter we propose to study the behavior of sequence aligners, which are real applications and mainly memory-bound, and evaluate their level of NUMA-awareness. We will first start with BWA, which is one of the most adopted sequence aligner by the scientific community.

As explained in chapter 4, BWA uses the Burrows-Wheeler Transformation of the reference genome, and two auxiliary data structures:*c_tab* and *o_cctab*. These data structures are used to calculate the range where a read sequence matches the reference genome, so they are accessed every time a read is analyzed. There is no dependency between reads, so several reads can be aligned in parallel. In fact BWA allows multithread execution employing pthreads. Each threads executes a sequential version of BWA algorithm. In BWA, the tables *c_tab* and *o_cctab* are allocated at the beginning of the execution in a single node, before the alignment starts. This means that all pthreads which are not on the local socket will have slower access time to these tables[25]. BWA counts with three algorithm: bwa-backtrack, bwa-sw, bwa-mem. In this section we will focus on BWA's original algorithm bwa-bactrack. BWA has three components: *index, aln,* and *sampe/samse*. The first component is used to index the reference genome, employing Burrows Wheeler Transformation. This only needs to be done one time, due that the same reference genome can be used for all the executions of BWA. The *aln* command takes the reads and calculates their suffix array coordinates using BWT and the reference genome index. In the last sub-module *samse* -for single end reads - and *sampe* -paired end reads- the chromosome coordinates are created. BWA implements parallelisation by means of pthreads library but only on *ALN* component. In order to obtain the minimum execution time possible, one could think that the best strategy is to use as many threads as cores available. However the results obtained are not consistent with this hypothesis. Execution times for BWA-ALN multithread using up to 64 threads are displayed in figure 5.1

The figure 5.1 reflects the execution times as the number of threads increases on AMD system. The blue line represents the execution times of BWA using the default allocation policy. This is letting the operating system be responsible for placing the data. It can be seen that BWA presents a limited scalability beyond 32 cores. When running an application on

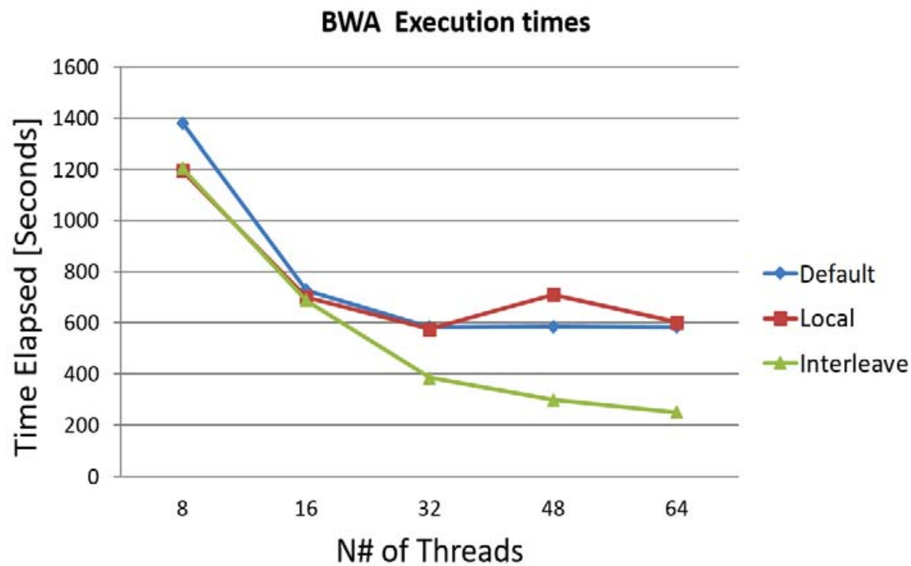


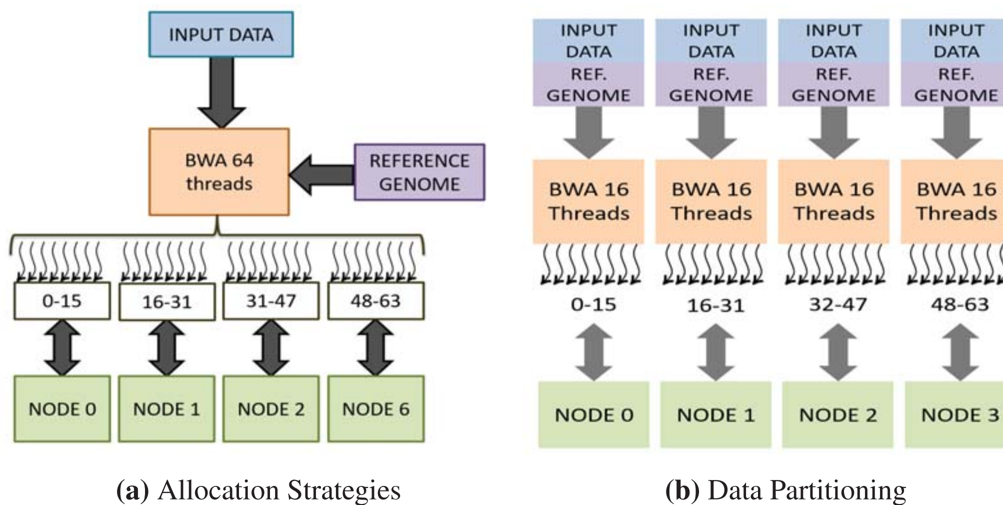
Figure 5.1: Scalability study for BWA-ALN on AMD architecture

a NUMA architecture, the most basic reasoning leads us to think that from 32 threads and higher, too many remote accesses are occurring and this counteracts the benefit of increasing parallelism. That is why it is even more surprising than when we forced the policy to allocate data so that it remains local to the main thread (red line), the performance is even worse. But if we base our understanding on the *NUMA performance aspects* of this architecture, we know that it is very prone to have contention problems both in applications with sequential and random memory access patterns. In addition, the fact of employing 32 threads -or higher- implies that remote access of the maximum distance can not be avoid. However, as discussed in the chapter 3 we can mitigate the effect if the data distribution is made using interleave policy, where access to all banks is balanced fairly. When we change the policy to interleave (green line), we can observe how the scalability increases now beyond the 32 threads, as predicted.

The behavior of BWA is adjusted to what was expected, taking into account the *NUMA performance aspects* of the system. This led us to design an experiment to characterize and evaluate the behavior of a set of sequence aligners, which we will explain in the next section.

5.2 Allocation Strategies and Data Partitioning

The experiments carried out in this study are the product of a series of systematic tests designed to evaluate the behavior of aligners in the two NUMA architectures. The experimentation can be divided into two main parts. In the first part, we tested several configurations of data allocation that enforced locality between threads and memory banks as well as configurations where shared data structures are spread evenly on different memory banks. In the second part, experiments were based on the idea of data partitioning and replication: multiple independent instances of the same application were executed simultaneously, so the main shared data were replicated on each memory bank and input data split. A conceptual scheme of both experiments are shown in figure 5.2. Details of these two schemes are presented below.



(a) Allocation Strategies **(b)** Data Partitioning

Figure 5.2: Experimentation approaches

5.2.1 Analysis of memory allocation

First, we have analyzed how sensitive a particular aligner is to different memory allocations. In order to achieve this, we carried out 3 experiments: the first one is a traditional scalability study in which aligners run with default system settings. We focused on 5 particular cases: using 8, 16, 32, 48 and 64 threads, because in the AMD system, each processor has 8 cores and 1 memory bank associated; so 8, 16, 32, 48 and 64 threads implies a minimum usage of 1, 2, 4, 6 and 8 NUMA nodes, respectively. To compare the output, we performed the same cases in the Intel System. For the other two cases, we used the Linux Tool *numactl* to set a memory policy allocation. With the parameter *–localalloc*, the data is allocated on the

current node where threads are running the program. The idea behind this is to maximize local data affinity, keeping data onto the closest memory to the running processor. Finally, in the third case the *–interleave* parameter is used so that memory is allocated in a round robin fashion between selected nodes. All the aligners that we used need two input data files: one that contains all the reads that need to be mapped and a second one that contains the reference genome index.

The objective of these experiments is, firstly, to gain insight into the level of scalability of the aligner. Additionally, re-running the aligner using different parameters of *numactl* provides us with information about the behavior of the application and its data allocation sensitivity by using two extreme cases: when the locality and concurrency increase (*localalloc*) and vice-versa (*interleave*).

5.2.2 Data partitioning and replications strategies

With the second part of our experimentation, we aim to reduce to a minimum the contention produced when multiple threads access the index. To achieve this, we used data replication and data partitioning techniques. We ran simultaneous independent instances of a given aligner, each instance with a copy of the index. For example, if 4 simultaneous independent instances are created, each one will process a 4th part of the original input data (see figure 5.2b) and use an entire copy of the index. It is important to remark that creating instances increases the initial memory requirements, due to the fact now multiple copies of the index are required instead of just one. Ideally, each memory bank would hold a copy of the reference index and the threads local to that bank would not need to access any data located remotely. For this case, we could think of each NUMA node as a symmetric multi-processor unit, capable of running an independent instance of an aligner. However, only BOWTIE2 and BWA generate an index small enough to fit on a memory bank of the systems we are using. The case of GEM and SNAP is different, where the index needs to be stored in more than one memory bank. To decide how many instances we would run, we took two critical factors into account:

1. the layout of the memory system.
2. the size of the index needed by the aligners.

Regarding the memory system, it is worth noting that AMD architecture presents more restricting features than INTEL, in the sense that the memory banks are smaller. The AMD

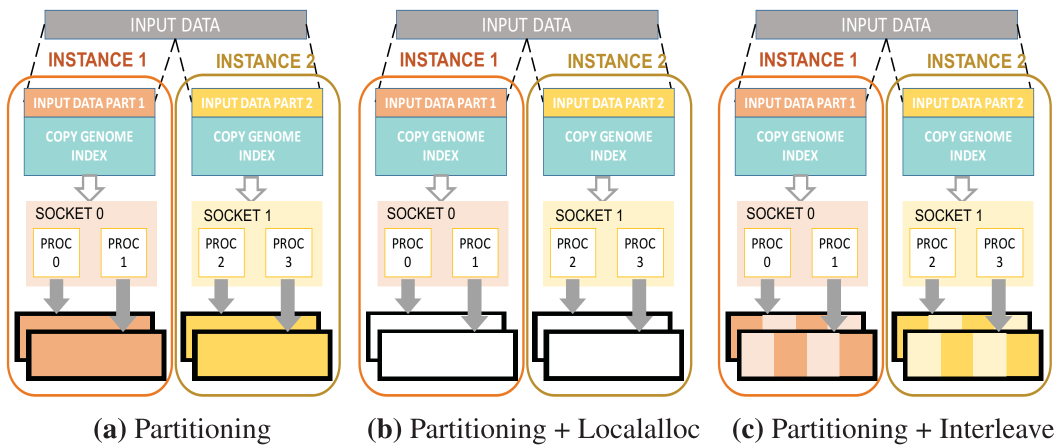
system also has a more complex layout due to the fact that the number of NUMA nodes is larger. Knowing this, we designed the experimentation having AMD's constraints in mind, but considering that it would work on the Intel system too. The AMD system allowed us a maximum of 8 instances of 8 threads each, for an aligner with a small index. For these aligners, we also created other combinations with 4 instances of 16 threads each and 2 instances of 32 threads, always using the maximum number of threads possible. In Table 5.1, we can see the sizes of the indexes used. For SNAP, it was not possible to create more instances than 2 due to the fact that the index does not fit on one memory bank of the AMD system and barely fits on two.

We introduce a novel hybrid execution technique combining the partitioning techniques with the memory allocation policies explained in Section 5.2.1: localalloc and interleave. Figure 5.3 shows a graphical representation of the three hybrid scenarios that we have tested when partitioning techniques were combined with memory allocation policies. In this example, 2 instances (partitions) are created. Instance 1 is running on Socket 0 (Processor 0 and Processor 1) and Instance 2 on Socket 1 (processor 2 and processor 3). Each instance processes one half of the total input and uses a copy of the index. When combined with memory allocation policies, data is allocated in three different ways, which result in the three resulting scenarios:

- **Partitioning (Original)** Figure 5.3a: Memory banks are reserved ahead of the execution using the command *membind*. NUMA nodes local to Processor 0 and Processor 1 are explicitly selected for Instance 1 and for Instance 2 the NUMA nodes local to Processor 2 and Processor 3. This does not necessarily mean that both memory banks would be used; they are allocated and will be used if needed.

- **Partitioning + Localalloc** Figure 5.3b: The difference between this technique and ”Partitioning” is that the reservation of NUMA nodes is performed implicitly, using the command *localalloc*.

- **Partitioning + Interleave** Figure 5.3c: As in ”Partitioning”, when an interleave policy is used, the NUMA nodes are explicitly reserved, but the allocation that takes place is done in a round-robin fashion, guaranteeing that both memories are being used and that the index is equally distributed.



(a) Partitioning **(b)** Partitioning + Localalloc **(c)** Partitioning + Interleave

Figure 5.3: Hybrid experimentation

Table 5.1: Detailed information about the aligners.

Aligner	Version	Index (GB)
BOWTIE2	2.2.9	3.9
BWA-MEM	0.7.12	5.1
GEM	3.0	15.0
SNAP	1.0.18	29.0

5.3 Experimental Results

In this section, we show the main results obtained during the experimentation. For all the experiments, we used the reference human genome GRCh37 [62], maintained by The Genome Reference Consortium, and two data sets were used as input data:

- *Synthetic benchmark [63]:*
  Single end, base length = 100, number of reads= 11M Size = 3.1GB

- *Segment extracted from NA12878 [64]:*
  Single end, base length = 100, number of reads= 22M Size = 5.4GB

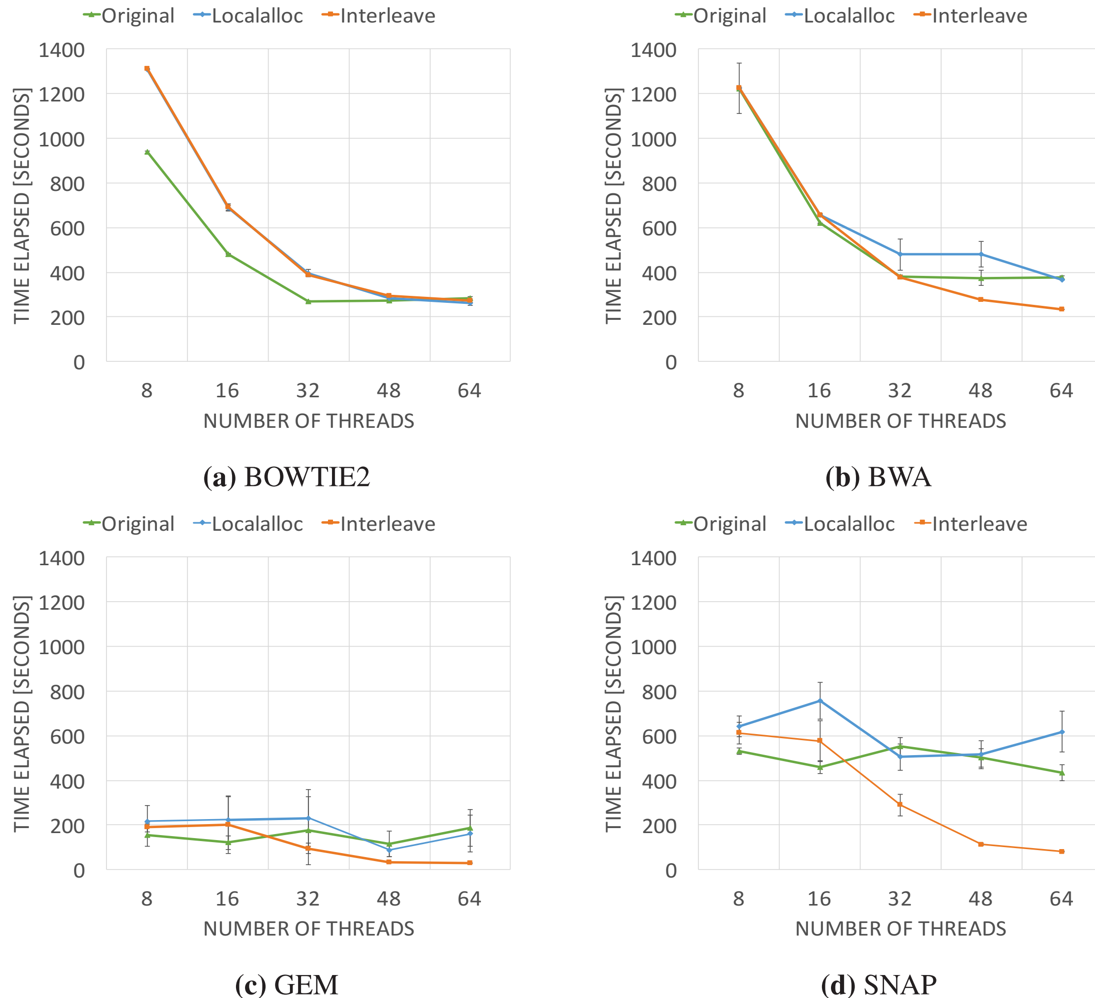
The aligners were compiled using GCC 4.9.1 and we used the latest version available of each, as shown in the second column of Table 5.1. Results were obtained as an average of ten executions. Figures 5.4, 5.5, 5.6 and 5.7 of the following subsection show both average execution times and the corresponding standard deviation for each test. Detailed numerical

values of mean execution times and corresponding relative errors are available in Tables A3, B5, C7 and D9, annexed at the end of this chapter.

5.3.1 Analysis of memory allocation policies

In Figures 5.4, 5.5, 5.6 and 5.7, the execution times for all four aligners can be seen, each one using the datasets mentioned above: GCAT Synthetic Input and NA12878 Real Input. Execution times are also evaluated for both systems described in Chapter 2 (AMD-based cluster and Intel-based cluster). For each aligner, each figure shows how it scales when different memory allocation policies are used (namely, *original*, *localalloc* and *interleave*). This first set of experiments shows the behavior of the aligners under three scenarios. The first one (*original*) corresponds to the execution of a given aligner with its default parameters without any particular allocation policy or NUMA control and lets the operating system handle the allocation. On Linux systems, this will normally involve spreading the threads throughout the system and using the first-touch data allocation policy, which means that, when a program is started on a CPU, data requested by that program will be stored on a memory bank corresponding to its local CPU [37]. The allocation policy takes effect only when a page is first requested by a process. The second case (*localalloc*) corresponds to the scenario where the functions of the *numactl* utility are used to reduce remote access, restricting the allocation to specific nodes. The third case (*interleave*) evaluates the performance of the application when its memory pages are distributed in the nodes following a round-robin scheme. When aligners are executed with no explicit memory allocation policy (shown by the green line in Figures 5.4, 5.5, 5.6 and 5.7), scalability decreases significantly beyond 32 threads in all four aligners. When these aligners run on more than 32 cores, at least one NUMA node at two-hops distance is used. Therefore, all the speedup gained due to multithreading is mitigated by the latency of remote accesses and traffic saturation of interconnection links. BOWTIE2 and BWA show a more regular and similar behavior. They reduce their execution time gradually to the point of using 32 cores. From there, their times are increased (slightly in the case of BWA and more significantly in the case of BOWTIE2). GEM and SNAP show a more irregular behavior since their execution times are not always reduced when the number of cores increases. It is worth mentioning, actually, that all aligners' execution time when using the complete system is worse than using a smaller number of cores. From the two memory allocation policies, interleave is clearly the one that performs best. For all the aligners, the interleave policy reduces execution time as more processors are used (with the only exception being BOWTIE2 in the case of the real dataset). GEM and SNAP, actually,

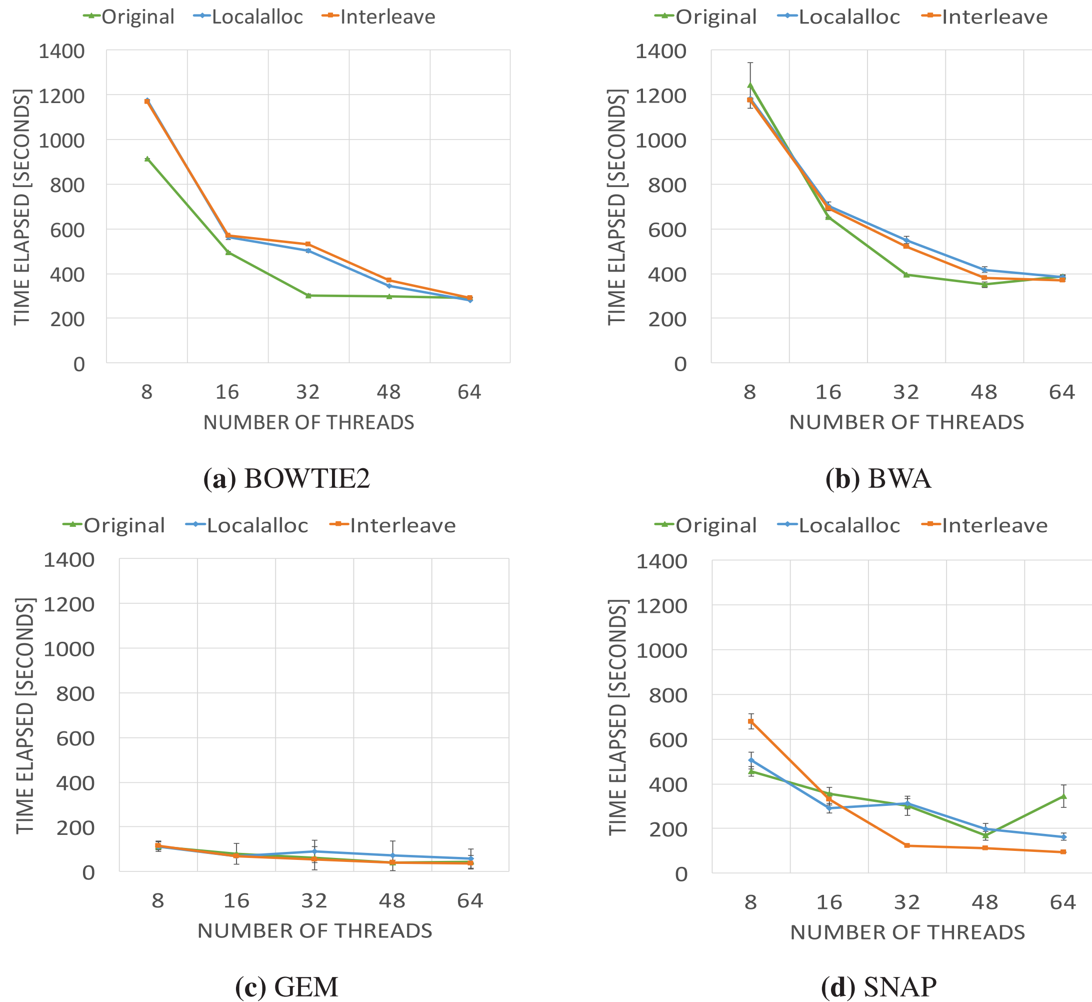
obtain their best execution times when interleave allocation is used. It is also worth noting that more stable and steady results are obtained with interleave. The variability between executions is reduced drastically, mainly because interleave spreads data across the system, and the aligners experience fair and balanced accesses.



(a) BOWTIE2 **(b)** BWA **(c)** GEM **(d)** SNAP

Figure 5.4: Different memory allocation policies. The lower the better. Arch: AMD. Dataset: GCAT Synthetic Input

As explained in Section 4.4, aligners share a common data structure -an index- among all threads. This structure is loaded in memory by the master thread (by default, Linux will place this data on its local memory bank). Therefore, as the number of threads increases, the memory bank that allocates the index becomes a bottleneck. Allocating data in an interleave way does not reduce remote accesses but guarantees a fair share of them between all memory banks and, therefore, prevents access contention, a phenomenon especially prone to happening



(a) BOWTIE2

(b) BWA

(c) GEM

(d) SNAP

Figure 5.5: Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: GCAT Synthetic Input

in these architectures due to reduced memory bandwidth between NUMA nodes [17]. This reason explains why using an allocation policy that reinforces locality between processors and memory banks does not provide good results. Execution times are made worse by both increased latencies in memory accesses and increased congestion of banks containing the index. Execution times for a given aligner change depending on the system that is used. This makes sense, as mentioned in Chapter 2: systems are equivalent but not identical. It is also worth noting that the behavior of a given aligner changes depending on the dataset. Aligners do not have a unique and uniform memory access pattern. There are queries that are easier to map than others, and the mappeability among the regions of the human genome is not uniform [65]. Not all aligners process the queries in the same way, and the work done is irregular. The

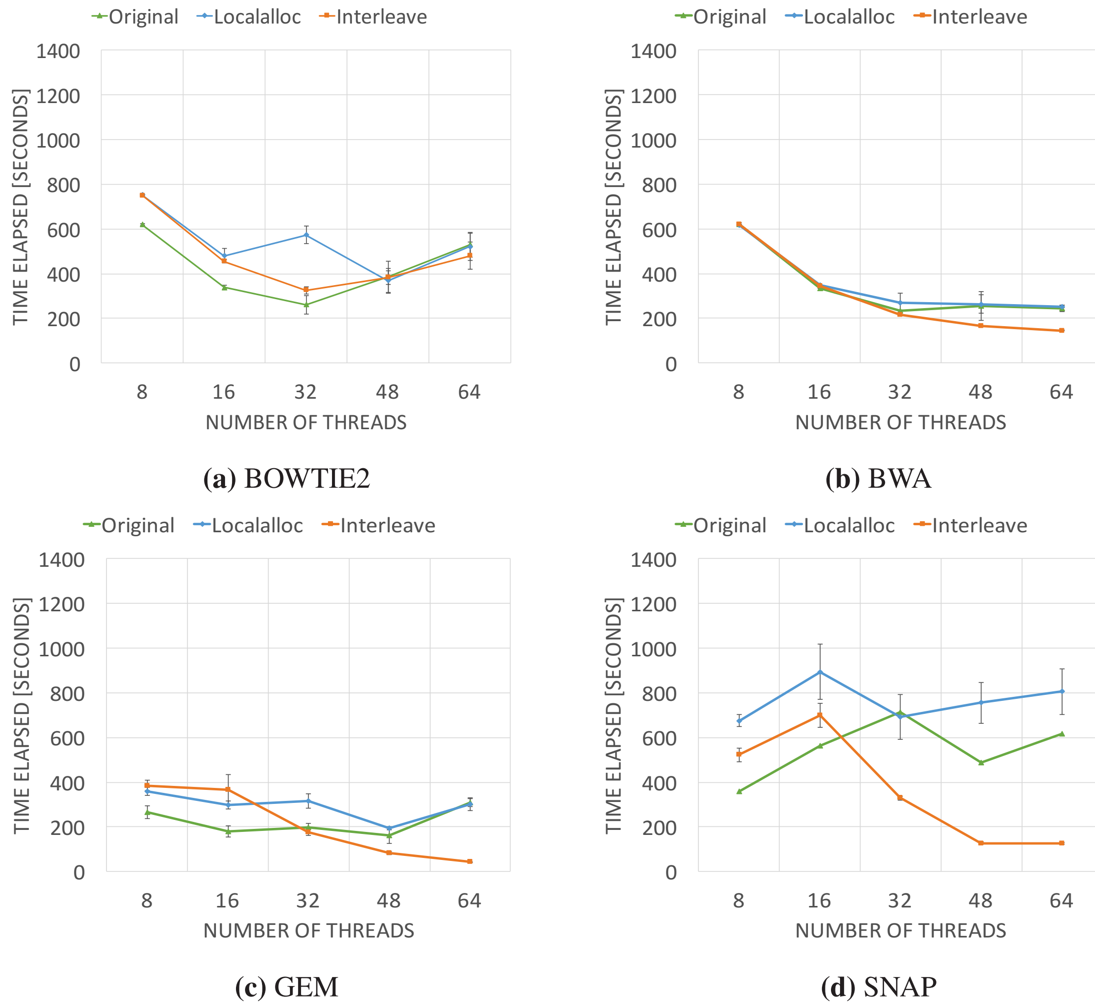


Figure 5.6: Different memory allocation policies. The lower the better. Arch: AMD. Dataset: NA12878 Real Input

nature of the input data can significantly affect the behavior of an aligner. However, a memory placement strategy that distributes data evenly among all nodes seems to be the strategy that most consistently provides the best results.

Accessing remote data is not the only problem on NUMA architectures. The congestion generated by multipl to a shared common data structure increases congestion. Executing the aligners with interleave policy does not reduce the amount of remote accesses but diminishes the drawbacks of congestion in a sensible way. These outcome matches with the ones predicted by the *NUMA performance aspects* exposed in chapter 3. When benchmarking both systems with STREAM we observed and quantify how an interleave fashion of placing the data improved the aggregated bandwidth of the entire system. How ever testing this theory,

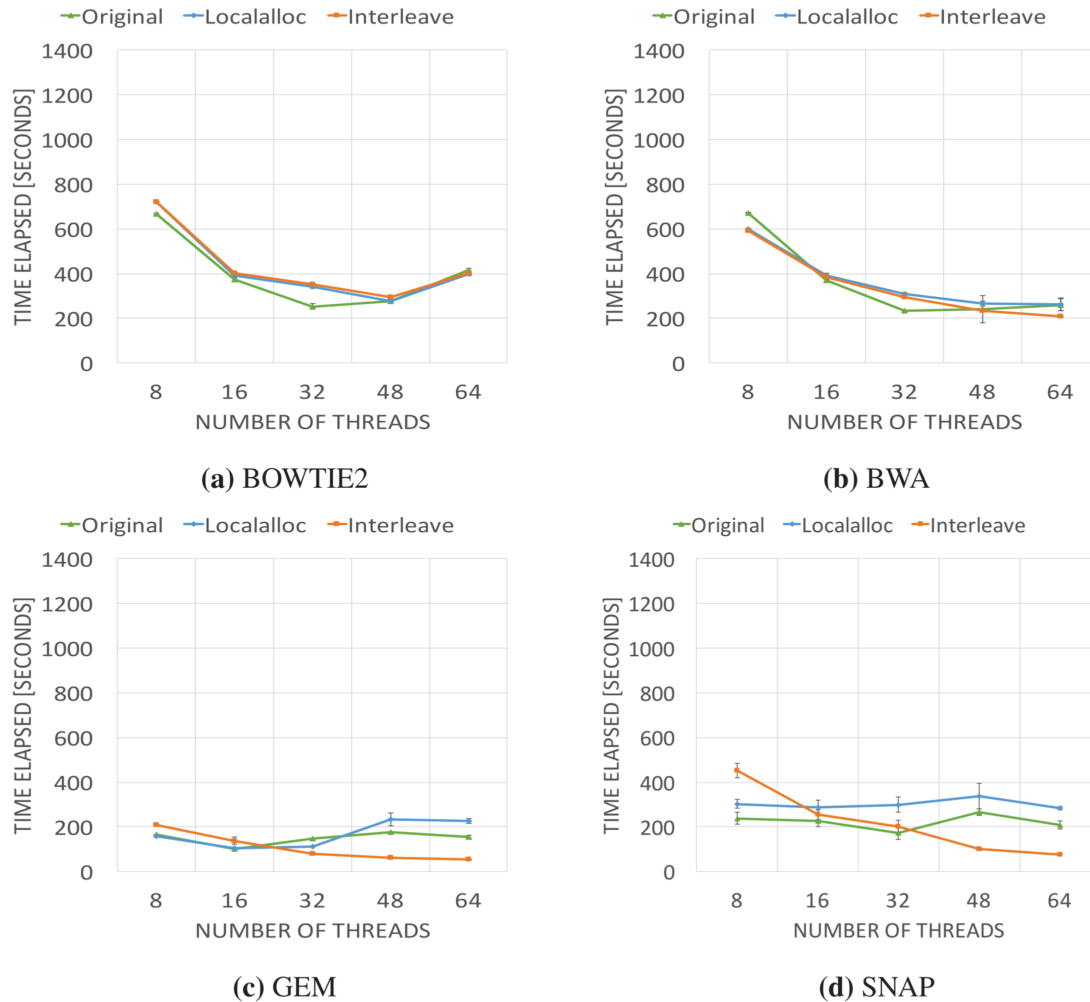


Figure 5.7: Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: A12878 Real Input

validates that this also happens in more complex application (than synthetic benchmark with sequential accesses).

5.3.2 Data partitioning and replication strategies

With the execution of multiple simultaneous instances, we aimed to attenuate the traffic of socket-interconnection buses and therefore reduce contention between threads. Latency will also improve because locality will increase and memory accesses will be available to local nodes. However, the way that instances are created is strongly conditioned by the system architecture that the aligner is running on. To generate the instances for each scenario, we

have taken into account the aligners' memory requirements (in particular, the size of the index generated by each aligner) and the amount of available space on the memory banks. Within the four chosen aligners for this study, there are two - BOWTIE2 and BWA - with indexes that are small enough to fit on one memory bank. For these aligners, we have been able to create as many instances as there where available NUMA domains. As the maximum number of threads is the same on both architectures, the same test can be executed on both systems. The maximum number of instances possible for AMD is 8 instances of 8 threads. This constitutes the most desirable situation because all instances would have all needed data stored locally, and remote accesses to remote banks would be significantly reduced. Unfortunately, this is not the situation for GEM and SNAP. Their indexes require more than one bank, therefore their instances are not entirely isolated and, although replication strategies are used, remote accesses and memory contention cannot be avoided completely. For these two aligners, only 2 instances of 32 threads were possible. In order to complete the experimentation, we have designed tests with other possible combinations of *instance x threads* for BOWTIE2 and BWA: 4 instances of 16 threads and 2 instances of 32 threads. For all scenarios the subset of NUMA nodes used, were selected using the ***NUMA performance aspects***. Once the number of instances is determined, a memory allocation policy is applied, thus generating the hybrid scenarios described in Section 5.2.2. In Table 5.2, we can see a list of all the hybrid scenarios that were tested in this part of the experimentation. The name of each case, as used in Figures 5.8, 5.9, 5.10 and 5.11, appears in the column *Names*.

Figures 5.8, 5.9, 5.10 and 5.11 show a complete speedup comparison of all strategies when the maximum number of cores are being used. This means that whole systems were used (with 64 threads and all memory banks). Speedup has been computed by using the execution time achieved by each aligner when it was executed on the whole system with its default setup (i.e. with no memory allocation policy and data partitioning strategy applied) as a baseline.

From these experiments it is observed that all aligners benefit from execution through the creation of instances in all cases. However, some slight differences can be observed in the aligners' behavior. On the one hand, aligners with small indexes (BOWTIE2 and BWA-MEM) improve their execution times in all scenarios in which instances are used, regardless of the memory allocation policy applied. The remarkable case is BOWTIE2, which presents speed-ups between 1.5x and 5x comparing to its original configuration. The best results are obtained when using the largest number of instances (*8Inst Original* in all Figures), which corresponds to the case that maximum locality is achieved, also reducing memory

Table 5.2: Instances created for each aligner

Aligner	#Inst x Threads	Policy	Name
BOWTIE2 BWA-MEM	2x32t	Original	2Inst_Original
	4x16t		4Inst_Original
	8x8t		8Inst_Original
	2x32t	Localalloc	2Inst_Localalloc
	4x16t		4Inst_Localalloc
	2x32t	Interleave	2Inst_Interleave
	4x16t		4Inst_Interleave
SNAP GEM	2x32t	Original	2Inst_Original
	2x32t	Localalloc	2Inst_Localalloc
	2x32t	Interleave	2Inst_Interleave

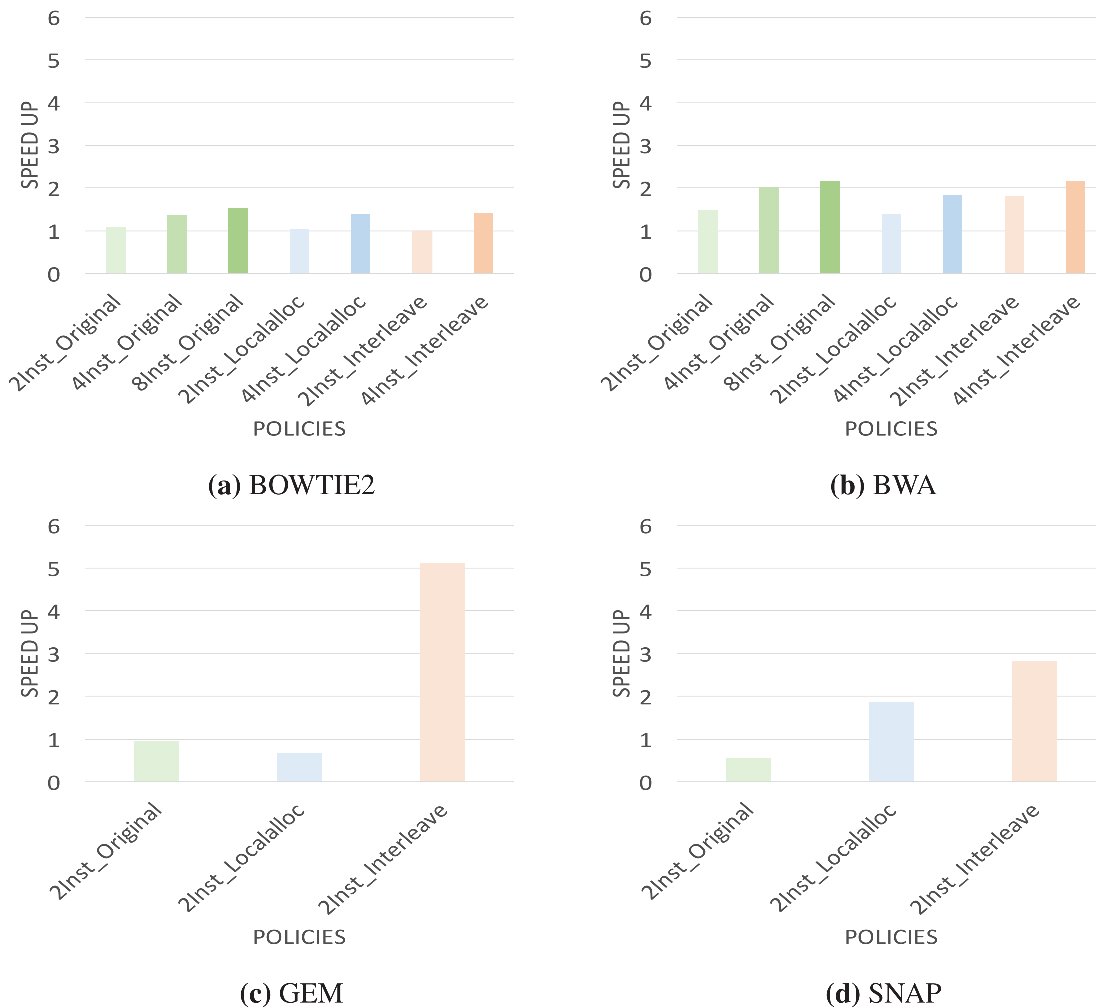
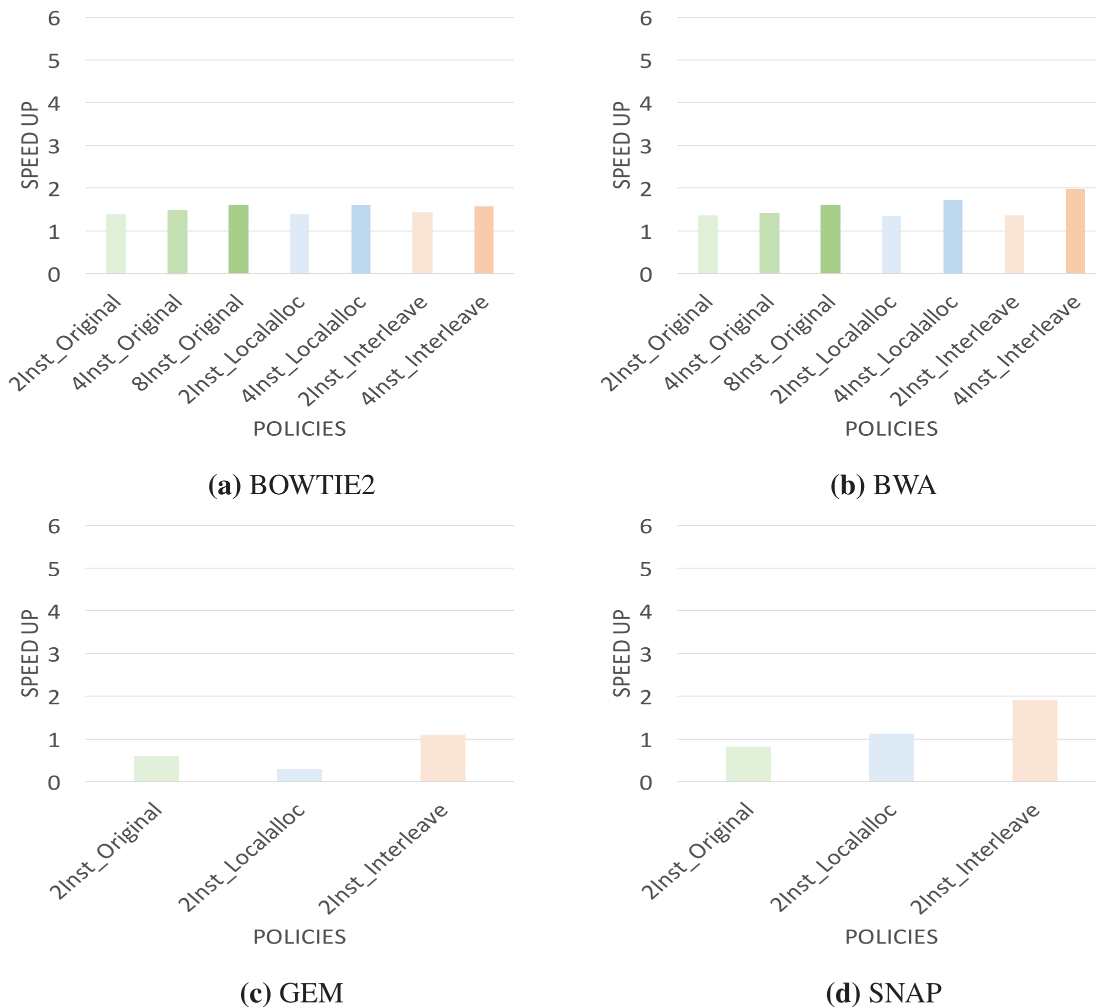


Figure 5.8: Different memory allocation policies. The lower the better. Arch: AMD. Dataset: GCAT Synthetic Input

controller congestion. When using a smaller number of instances aligners also benefit from the combination with the memory allocation policies. The interleaving allocation provides slightly better results compared to the other two cases. This means that if BWA-MEM or BOWTIE2 are executed with four instances of 16 processors each, allocating memory in a round-robin fashion provides better results than allocating memory in any other way. This hybrid schema provides the best trade-off between locality increase and contention avoidance.

On the other hand, aligners with larger indexes (GEM and SNAP) always benefit from the creation of instances combined with interleave allocation. Execution times were not always better than the default ones when instances were combined with the other allocation schemes. The strategy that combines multiple instances and memory interleaving improves execution



(a) BOWTIE2

(b) BWA

(c) GEM

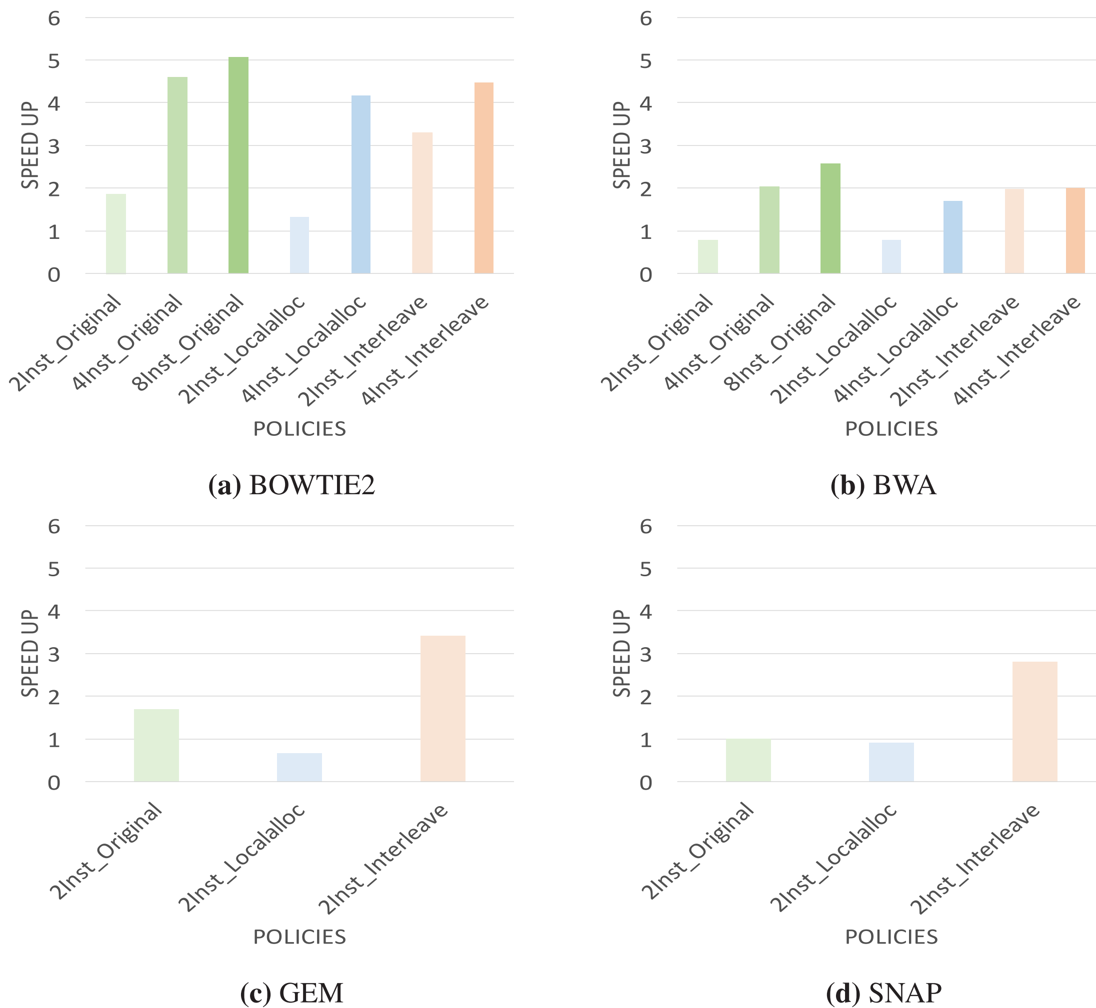
(d) SNAP

Figure 5.9: Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: GCAT Synthetic Input

time up to 5x in the case of GEM and 2.8X in the case of SNAP. However, it is worth noting that none of these results are better than using an interleave allocation policy alone.

5.3.3 Summary results

Figure 5.12 summarizes the main results achieved by each aligner when they were executed with the maximum number of resources (i.e. using 64 threads). In each figure, we can see 4 sets of tests: one for each input and one for each architecture, four combinations of experiments in total. For each set, the first column represents the execution time of the aligner without any added memory policy or data partition. The second column is the best



(a) BOWTIE2

(b) BWA

(c) GEM

(d) SNAP

Figure 5.10: Different memory allocation policies. The lower the better. Arch: AMD. Dataset: NA12878 Real Input

time achieved when the aligner was executed with a memory policy, and the last column is the best execution time achieved with data partitioning and independent instances. As said before, aligners exhibit random access patterns to memory and are also sensitive to the input data. However, as Figure 5.12 shows, significant improvements in execution times were achieved by all 4 aligners when memory interleaving or multiple instances were used. According to these results, we can deduce a rule of thumb that has shown to be valid for this set of representative aligners. For BOWTIE2 and BWA-MEM, where the size of the index is much smaller than the capacity of a memory bank, data partitioning arises as the best solution because it allows us to minimize the usage of socket interconnection links (QPI and HyperTransport). For aligners with larger indexes, such as SNAP and GEM, even when

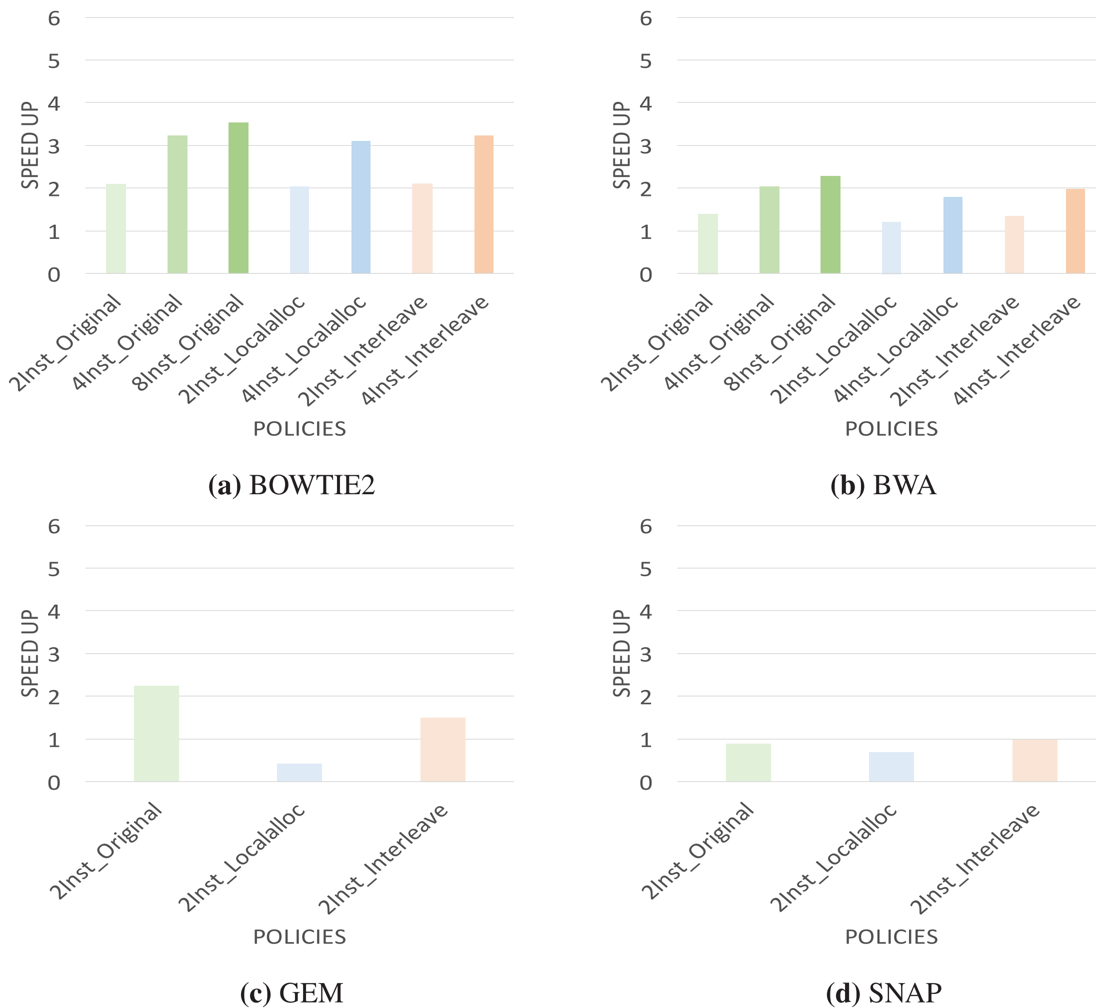
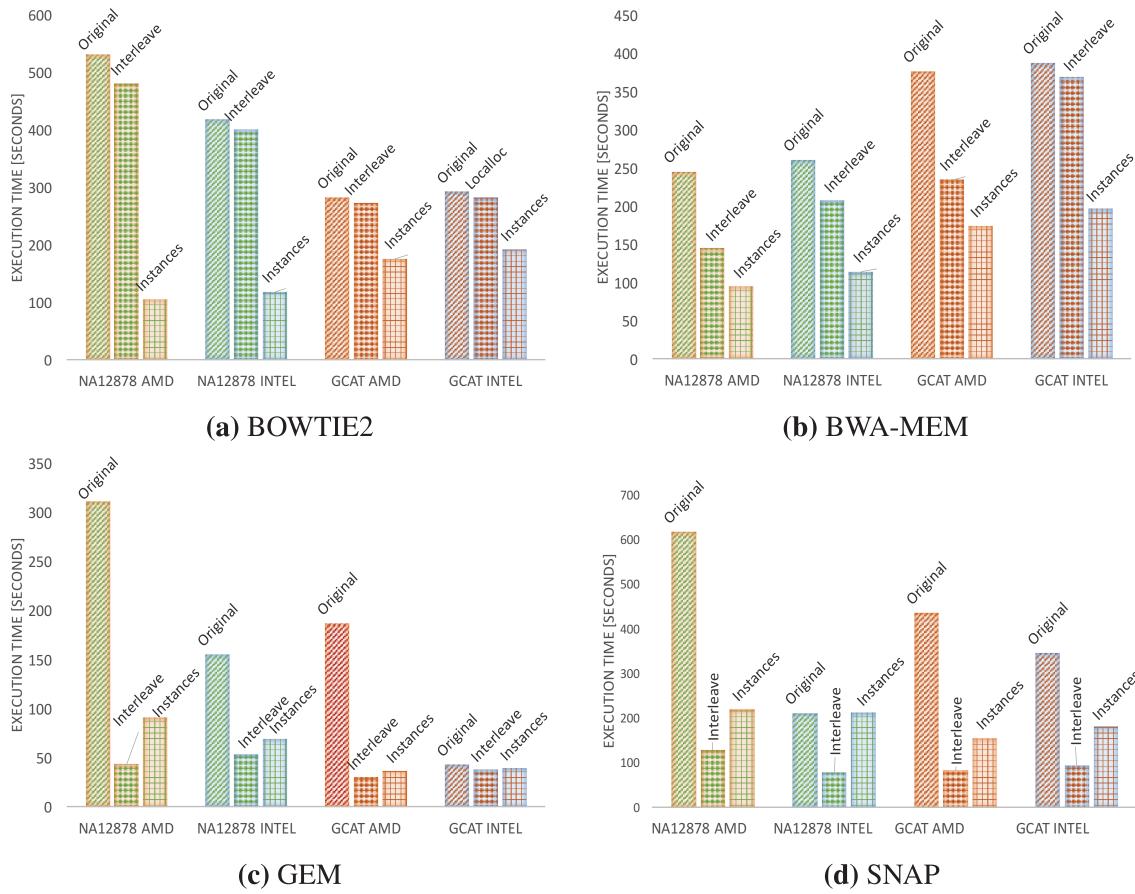


Figure 5.11: Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: A12878 Real Input

data partitioning is employed, more than one memory bank is required to store the index, and accesses through interconnection links cannot be avoided. For these aligners, execution time is mostly reduced when a pure interleave policy is employed, ensuring that accesses are equally distributed among all memory banks and, therefore, contention is minimized.

Improvements in execution times were greater when real input was used. Bigger improvements were also obtained on the AMD system. The NUMA architecture of this system has longer distances between processors and memory banks and shows latencies greater than those of the Intel system. Therefore, aligners suffer greater penalties in AMD architectures in terms of memory accesses in general; but, by applying NUMA-aware strategies, aligners also show more substantial improvements in such systems.



(a) BOWTIE2 **(b)** BWA-MEM **(c)** GEM **(d)** SNAP

Figure 5.12: Execution times: Summary results for all aligners. The lower the better

5.4 Annexed results

Table A3: Execution times for BOWTIE2 using different memory allocation policies

BOWTIE2

AMD

	NA12878						GCAT					
	Original		**Localalloc**		**Interleave**		**Original**		**Localalloc**		**Interleave**	
Threads	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**
8	620.37	0.80	752.28	0.57	749.88	0.59	937.96	0.48	1310.05	0.06	1311.13	0.09
16	339.21	2.77	480.94	6.47	455.81	1.60	481.81	0.28	687.68	1.68	692.95	2.15
32	260.75	15.73	573.00	6.91	326.25	4.93	271.29	1.36	396.29	4.02	387.34	1.26
48	385.47	13.52	369.64	14.60	383.85	7.82	271.58	1.50	285.21	1.05	293.66	0.86
64	529.92	10.51	520.64	11.82	480.47	12.50	282.33	2.93	262.09	4.38	272.07	2.72

INTEL

	NA12878						GCAT					
	Original		**Localalloc**		**Interleave**		**Original**		**Localalloc**		**Interleave**	
Threads	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**
8	721.51	0.41	722.62	0.27	666.22	0.59	915.07	0.10	1174.50	0.51	1169.95	0.26
16	403.03	1.00	403.99	3.43	373.43	1.86	495.40	0.62	563.76	1.72	569.07	0.33
32	350.98	1.15	347.59	2.18	252.68	5.08	303.09	1.69	501.55	1.36	530.68	0.39
48	293.05	1.92	279.86	1.58	275.15	2.71	296.58	1.14	344.19	1.26	368.96	0.75
64	400.77	1.60	402.51	1.75	417.47	1.77	292.12	0.86	282.03	1.10	292.53	0.54

Table A4: Execution times for BOWTIE2 using data partitioning

BOWTIE2									
		AMD				INTEL			
		NA12878		GCAT		NA12878		GCAT	
Name	**Threads**	**Mean [s]**	**Error [%]**	**Mean [s]**	**Error [%]**	**Mean [s]**	**Error [%]**	**Mean [s]**	**Error [%]**
2Inst_Original	32	284.78	5.25	201.50	3.63	199.40	10.14	272.50	4.85
4Inst_Original	16	115.03	9.14	188.86	6.06	129.14	7.89	214.53	6.16
8Inst_Original	8	104.52	13.55	175.36	6.28	118.23	2.33	191.58	2.42
2Inst_Localalloc	32	398.46	16.59	202.33	6.26	205.20	22.72	278.52	2.77
4Inst_Localalloc	16	127.08	8.70	175.09	1.00	134.51	8.20	211.12	2.31
2Inst_Interleave	32	160.59	4.87	197.56	1.31	199.39	10.14	293.21	1.45
4Inst_Interleave	16	118.75	4.43	180.48	4.63	129.14	7.89	206.66	2.07

Table B5: Execution times for BWA-MEM using different memory allocation policies

BWA-MEM

AMD

	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
Threads	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**
8	621.40	0.85	618.78	0.41	620.07	0.33	1223.40	9.22	1224.56	0.16	1225.20	0.23
16	333.76	0.33	348.35	0.45	346.37	0.71	619.03	0.40	658.15	0.60	655.84	0.90
32	234.89	0.71	269.83	15.78	214.69	0.26	381.98	1.44	479.43	14.90	377.16	0.42
48	254.82	25.22	264.07	15.57	164.24	1.68	374.68	9.25	480.27	11.91	276.52	0.40
64	244.95	6.02	253.13	2.45	145.10	1.49	376.32	2.20	367.73	0.67	234.23	0.40

INTEL

	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
Threads	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**	**Mean [s]**	**Error[%]**
8	671.26	0.25	598.68	0.39	591.44	1.21	1242.25	8.23	1181.80	0.32	1175.52	0.48
16	368.48	1.10	391.26	3.24	385.35	0.65	651.29	0.48	703.42	2.71	691.03	1.23
32	234.07	1.37	309.14	2.70	293.92	0.67	393.66	0.33	550.59	2.84	521.19	0.35
48	239.97	25.11	265.37	4.11	234.67	0.76	350.85	3.75	417.55	3.18	380.09	0.27
64	260.20	10.04	263.35	10.85	207.75	1.24	387.17	1.60	384.51	2.25	368.91	1.02

Table B6: Execution times for BWA-MEM using data partitioning

BWA-MEM

		AMD				INTEL			
		NA12878		GCAT		NA12878		GCAT	
Name	Threads	Mean [s]	Error [%]	Mean [s]	Error [%]	Mean [s]	Error [%]	Mean [s]	Error [%]
2Inst_Original	32	315.47	17.32	254.19	10.03	186.52	3.61	285.88	2.50
4Inst_Original	16	120.62	26.46	187.19	11.30	128.07	4.82	273.23	3.74
8Inst_Original	8	95.18	14.60	174.18	8.13	113.95	18.42	240.59	3.04
2Inst_Localalloc	32	316.14	19.45	272.18	6.90	214.15	16.13	289.98	2.24
4Inst_Localalloc	16	143.99	9.20	204.88	10.59	144.75	6.67	224.44	9.41
2Inst_Interleave	32	123.69	0.92	208.05	4.07	194.23	14.67	283.68	0.67
4Inst_Interleave	16	122.814	9.51	173.85	0.11	131.57	8.24	196.14	1.14

Table C7: Execution times for GEM using different memory allocation policies

GEM

AMD

	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
Threads	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]
8	359.59	9.24	675.36	4.06	522.62	6.06	156.29	5.40	217.24	5.71	190.61	8.39
16	562.60	11.49	894.56	13.89	700.63	7.67	121.81	13.01	223.38	14.22	201.40	14.34
32	712.83	4.44	691.45	14.40	329.49	2.90	175.94	14.47	232.01	12.09	94.98	14.57
48	488.35	15.41	756.27	12.09	127.51	4.14	115.13	8.22	87.39	13.86	34.55	0.67
64	616.17	15.99	805.56	12.82	127.24	2.76	186.47	10.18	162.08	10.21	30.18	1.01

INTEL

	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
Threads	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]
8	166.94	0.17	157.82	1.92	209.87	1.88	111.14	1.31	113.73	2.58	115.83	1.11
16	102.06	0.48	105.65	2.54	137.71	11.45	79.30	1.55	70.23	1.65	69.41	1.02
32	147.04	0.55	111.87	0.77	78.21	1.28	60.77	1.75	91.90	2.69	53.66	8.33
48	175.41	1.14	233.83	11.70	61.23	1.22	41.56	4.08	70.93	3.75	42.08	8.82
64	154.66	4.26	225.36	4.92	53.02	1.88	42.42	9	58.02	3.24	37.72	9.37

Table C8: Execution times for GEM using data partitioning

GEM

		AMD				INTEL			
		NA12878		GCAT		NA12878		GCAT	
Name	Threads	Mean [s]	Error [%]	Mean [s]	Error [%]	Mean [s]	Error [%]	Mean [s]	Error [%]
2Inst_Original	32	183.21	7.10	197.09	23.35	68.99	12.04	70.77	23.68
2Inst_Localalloc	32	457.95	6.57	277.87	7.92	362.08	22.92	139.14	17.79
2Inst_Interleave	32	90.68	14.34	36.39	10.91	104.25	9.63	38.77	5.40

Table D9: Execution times for SNAP using different memory allocation policies

SNAP

AMD

	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
Threads	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]
8	266.12	10.19	359.73	5.32	385.81	6.42	531.21	2.91	641.46	7.31	611.84	8.03
16	179.59	13.67	297.83	6.04	367.85	14.57	458.95	5.85	757.73	10.96	577.34	15.55
32	196.58	9.30	316.51	10.42	176.06	7.57	552.31	7.52	504.56	11.93	290.12	14.34
48	161.99	21.72	195.39	3.93	83.21	10.28	501.86	8.12	516.69	12.12	112.99	5.42
64	310.72	5.84	300.50	8.82	43.07	1.93	434.68	7.88	618.66	14.59	81.35	2.33

INTEL

	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
Threads	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]	Mean [s]	Error[%]
8	238.91	10.97	303.45	6.86	451.87	7.10	456.73	4.75	504.73	7.53	679.12	4.94
16	226.77	10.96	287.53	10.75	255.52	8.15	355.90	8.08	292.29	7.51	331.94	8.21
32	172.94	15.69	299.46	11.00	199.86	15.00	301.85	14.12	312.13	7.61	122.85	0.55
48	266.62	5.76	339.21	16.89	102.13	6.21	168.38	11.42	199.75	11.69	111.74	0.38
64	209.95	8.57	284.43	2.53	77.62	0.72	344.04	14.46	163.80	9.42	93.25	0.37

Table D10: Execution times for SNAP using data partitioning

SNAP

		AMD				INTEL			
		NA12878		GCAT		NA12878		GCAT	
Name	Threads	Mean [s]	Error [%]	Mean [s]	Error [%]	Mean [s]	Error [%]	Mean [s]	Error [%]
2Inst_Original	32	613.81	3.01	788.64	9.87	233.31	21.30	415.65	15.33
2Inst_Localalloc	32	675.62	5.04	233.34	9.68	301.45	14.33	309.68	19.68
2Inst_Interleave	32	219.15	15.11	153.74	16.91	211.57	8.89	179.95	14.78

6

Conclusions and Future lines

"Anyone who keeps the ability to see beauty never grows old."

- Franz Kafka

6.1 Conclusions

Knowing the underlying architecture where applications are running is a key aspect to achieving their optimal performance. If an application is memory-bound, it might suffer performance issues when executed in NUMA systems. We proposed a series of systematic steps to evaluate and characterize NUMA systems. These steps were carried out employing well-known memory benchmarks like STREAM and lmbench. We quantify the effects of poor locality, sub-optimal combination of subset of NUMA nodes and race condition of parallel applications through four dimensions called *NUMA performance aspects*. With this, We this gained understanding about the architecture employed we design an experimentation to validate if the behavior detected with the synthetic benchmarks could be extrapolated to real and complex applications.

We evaluated several genomic aligners and we have seen that they exhibit poor scalability in modern NUMA systems because they are penalized by contention and/or remote memory bank accesses. Our experiments have shown that increasing data locality may not always produce the expected outcome. As the number of threads increases, all aligners show poor scalability. In our study, we have shown that this phenomenon is not only related to remote memory accesses taking place but also due to the memory contention generated by the race of multiple threads trying to access a single shared data structure. Minimizing memory contention is a key aspect in increasing the performance of aligners.

Our experiments have found that congestion causes the most serious NUMA problems for a representative set of genomic aligners. Congestion happens when the rate of requests to memory banks or the rate of traffic over interconnects is too high. As a consequence, memory accesses are delayed and execution time increases. We have evaluated several strategies that can be applied to alleviate this problem so that the application can take advantage of all the available processors in existing NUMA systems. These strategies do not require changes to the original application code, and they don't require either kernel modification or privilege permissions. We have explored several solutions that are based on combining two concepts: congestion avoidance and increased locality. Congestion avoidance looks to balance the traffic among multiple memory banks. Genomic aligners with large reference indexes (GEM and SNAP) especially benefit from this strategy. Increased locality was reinforced by running aligners in multiple instances. Aligners with small indexes (BWA-MEM and BOWTIE2) show significant improvements in execution time thanks to this strategy, which could also be combined with memory interleaving if the size of the memory banks is not large enough to hold genome indexes.

Improvements in execution times of 5x and 2.5x were obtained for BOWTIE2 and BWA-MEM, respectively, when the aligners were executed with the maximum number of threads (64). For other aligners with larger indexes (i.e. SNAP and GEM), the *interleave* technique proved to be a better choice because the index is distributed across the system memory banks and mitigates the contention produced when all threads try to access the same data structure. Improvements of up to 5x and 2.8x were obtained for GEM and SNAP, respectively.

Although there is no single strategy that emerges as the best for all scenarios, the proposed strategies of this study improved the performance of all the aligners. This is not a minor achievement taking into account that the behavior of the aligners is quite susceptible to variation depending on the nature of the input data and the system architecture they run on.

It is reasonable to assume that NUMA systems in the future will have more NUMA nodes and more complicated interconnection topologies. This will imply that NUMA effects will continue to be a concern. Therefore, it will be necessary to apply techniques such as those presented in this work so that parallel applications can be optimized efficiently to take advantage of all the resources that will be available in those systems. We have evaluated several strategies that don't require changes to the applications. However, we expect that larger improvements could be achieved if NUMA-awareness is integrated into the design of new aligners.

6.2 Future Lines

As future lines we see several possible open paths:

- **Implement a unified tool:** As a first step we want to implement methodology as a tool. We would like to codify a program that groups all the benchmarks used in this research, allowing the user to characterize the architecture automatically. Currently it is implemented through scripts, but we believe it would be better fit to do it as an autonomous tool. Ideally, this tool would allow an inexperienced user to better understand the limitations of the system, and obtain hints on how to execute parallel applications, for example if it is convenient to execute her application using the interleave policy, or maybe instances replication, or nothing at all. Also it would allow to graphically interpret and quantify the *NUMA performance aspects*. This information could be really useful to parallel programmers too. They could design better allocation strategies, through the usage of numalib library, identify the stages

of their code and select an allocation policy accordingly.

- **Create NUMA pragmas:** As a long term goal we would like to focus our guideline not only in characterizing the architecture but also the parallel applications. We want to create *NUMA pragmas* to instrument code to detect applications' contention problems in NUMA systems. This could be reported by existing performance analysis tools that use our tool as an add-on. In a second step, specific pragmas could be interpreted by compiler an apply directly creation and execution of instances and the data replication.

- **Extend our model to all NGS workflow:** Some of the execution strategies introduced in this research can be extended to the following stages of the NGS workflow. For the variant calling stage, where the parallelization of the tools is limited, we have some preliminary results that show scalability problems on NUMA systems. It would be interest to test our proposed guideline with several known applications of this stage, specially GATK [66]. When working with a set of stages sharing intermediate data, like genomic workflows, the spectrum of possible strategies for improving the usage of memory increases. One promising idea is to apply instance and data replication to the alignment and variant calling stages, but passing the data between them using a NUMA-aware RAM drive instead of writing files on disk.

Bibliography

[1] P. García-Risueño and P. E. Ibañez, "A review of High Performance Computing foundations for scientists," *Int. Journal of Modern Physics C*, vol. 23, no. 07, pp. 1–33, 2012.

[2] J. Weinbub, "Frameworks for micro- and nanoelectronics device simulation," Ph.D. dissertation, Technischen Universität Wien, 8 2010.

[3] J. Lenis and M. A. Senar, "On the Performance of BWA on NUMA Architectures," in *2015 IEEE Trustcom/BigDataSE/ISPA*, 2015, pp. 236–241.

[4] ——, "Optimized execution strategies for sequence aligners on NUMA architectures," in *2016 Springer LNCS/EUROPAR_PBIO*, 2016.

[5] J. Lenis and M. À. Senar, "A performance comparison of data and memory allocation strategies for sequence aligners on NUMA architectures," *Cluster Computing*, vol. 20, no. 3, pp. 1909–1924, 2017. [Online]. Available: https://doi.org/10.1007/s10586-017-1015-0

[6] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform." *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[7] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv preprint arXiv: 1303.3997*, vol. 00, p. 3, 2013.

[8] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, 2012.

[9] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The GEM mapper: fast, accurate and versatile alignment by filtration." *Nature methods*, vol. 9, pp. 1185–8, 2012.

[10] M. Zaharia, W. Bolosky, and K. Curtis, "Faster and More Accurate Sequence Alignment with SNAP," *arXiv preprint arXiv:1111.5572v1*, pp. 1–10, 2011.

[11] J. Corbet, "AutoNUMA: the other approach to NUMA scheduling," https://lwn.net/Articles/488709, 2012.

[12] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and R. Mark, "Traffic management: a holistic approach to memory placement on NUMA systems," in *Proc. of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 381–394.

[13] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quema, R. Lachaize, and R. Mark, "Challenges of memory management on modern NUMA systems," *Communications of the ACM*, vol. 58, pp. 59–66, 2015.

[14] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on NUMA systems: asymmetry matters," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 2015, pp. 277–289. [Online]. Available: https://dl.acm.org/citation.cfm?id=2813788

[15] D. Beniamine, M. Diener, G. Huard, and P. Navaux, "Tabarnac: visualizing and resolving memory access issues on numa architectures," *Proceedings of the 2nd Workshop on Visual Performance Analysis - VPA15'*, pp. 1–9, 11 2015.

[16] Z. Majo and T. R. Gross, "(Mis)understanding the NUMA memory system performance of multithreaded workloads," in *Proceedings - 2013 IEEE International Symposium on Workload Characterization, IISWC 2013*. IEEE, sep 2013, pp. 11–22. [Online]. Available: http://ieeexplore.ieee.org/document/6704666/

[17] D. Molka, D. Hackenberg, and R. Schöne, "Main memory and cache performance of intel sandy bridge and amd bulldozer," in *Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14. NY, USA: ACM, 2014, pp. 4:1–4:10.

[18] R. Braithwaite, P. McCormick, and W.-c. Feng, "Empirical memory-access cost models in multicore numa architectures," *Virginia Tech Department of Computer Science*, 2011.

[19] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 12 1995.

[20] ——, “STREAM: Sustainable Memory Bandwidth in High Performance Computers,” 1991-2007, a continually updated technical report.

[21] L. McVoy and C. Staelin, “Lmbench: Portable tools for performance analysis,” in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’96. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268299.1268322

[22] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[23] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro Domínguez, J. C. Pichel Campos, and F. Fernández Rivera, “Using an extended Roofline Model to understand data and thread affinities on NUMA systems,” *Annals of Multicore and GPU Programming: AMGP, ISSN 23413158, Vol. 1, Nº. 1, 2014, págs. 56-67*, vol. 1, no. 1, pp. 56–67, 2014. [Online]. Available: https://dialnet.unirioja.es/servlet/articulo?codigo=4744516

[24] C. Misale, G. Ferrero, M. Torquati, and M. Aldinucci, “Sequence alignment tools: One parallel pattern to rule them all?” *BioMed Research International*, vol. 2014, 2014.

[25] C. Herzeel, T. J. Ashby, P. Costanza, and W. D. Meuter, “Resolving Load Balancing Issues in BWA on NUMA Multicore Architectures,” in *10th Int. Conf PPAM 2013*, vol. 8385. Springer Berlin Heidelberg, 2014, pp. 227–236.

[26] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, “BigBWA: Approaching the Burrows-Wheeler aligner to Big Data technologies,” *Bioinformatics*, vol. 31, no. 24, pp. 4003–4005, 2015.

[27] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[28] J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK Benchmark: past, present and future,” *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003. [Online]. Available: http://dx.doi.org/10.1002/cpe.728

[29] L. Dagum and R. Menon, “Openmp: an industry standard api for shared-memory programming,” *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[30] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, Aug. 1995. [Online]. Available: http://doi.acm.org/10.1145/209937.209958

[31] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[32] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2010.

[33] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[34] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998. [Online]. Available: http://doi.acm.org/10.1145/289918.289920

[35] D. Bonachea, "Gasnet specification, v1.1," Berkeley, CA, USA, Tech. Rep., 2002.

[36] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, Oct. 1993. [Online]. Available: http://doi.acm.org/10.1145/167962.165874

[37] C. Lameter, B. Hsu, and M. Sosnick-Pérez, "NUMA (Non-Uniform Memory Access): An Overview," *ACMQueue*, vol. 11, pp. 1–12, 2013.

[38] A. Kleen, "An NUMA API for Linux," SUSE Labs, Tech. Rep. 2, 2004.

[39] S. Shende, "Profiling and tracing in linux," 1999.

[40] R. Kufrin, "Measuring and improving application performance with perfsuite," *Linux J.*, vol. 2005, no. 135, pp. 4–, Jul. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1072710.1072714

[41] V. Weaver, "System-wide performance counter measurements: Offcore, uncore, and northbridge performance events in modern processors," Tech. Rep., 6 2017.

[42] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ser. ICPPW '10. Washington,

DC, USA: IEEE Computer Society, 2010, pp. 207–216. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2010.38

[43] D. Terpstra, H. Jagode, H. You, e. M. S. Dongarra, Jack", M. M. Resch, A. Schulz, and W. E. Nagel, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[44] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006. [Online]. Available: https://doi.org/10.1177/1094342006064482

[45] "Amd's 6376 specification," https://www.amd.com/en/products/cpu/6376.

[46] "Intel's e5-4620 specification," https://ark.intel.com/products/75286/Intel-Xeon-Processor-E5-4620-v2-20M-Cache-2_60-GHz.

[47] U. Drepper, "What every programmer should know about memory," 2007.

[48] J. Leyton, "Finding memory bottlenecks with stream," http://www.admin-magazine.com/HPC/Articles/Finding-Memory-Bottlenecks-with-Stream#.

[49] M. P. Dolled-Filhart, M. Lee, C.-w. Ou-yang, R. R. Haraksingh, and J. C. Lin, "Computational and Bioinformatics Frameworks for Next-Generation Whole Exome and Genome Sequencing," *The Scientific World Journal*, vol. 2013, pp. 1–10, 2013.

[50] S. Pabinger, A. Dander, M. Fischer, R. Snajder, M. Sperk, M. Efremova, B. Krabichler, M. R. Speicher, J. Zschocke, and Z. Trajanoski, "A survey of tools for variant analysis of next-generation genome sequencing data," *Briefings in Bioinformatics*, vol. 15, no. 2, pp. 256–278, 2014. [Online]. Available: +http://dx.doi.org/10.1093/bib/bbs086

[51] M. Baker, "De novo genome assembly: what every biologist should know," *Nature Methods*, vol. 9, no. 4, pp. 333–337, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1038/nmeth.1935

[52] F. Torri, I. D. Dinov, A. Zamanyan, S. Hobel, A. Genco, P. Petrosyan, A. P. Clark, Z. Liu, P. Eggert, J. Pierce, J. A. Knowles, J. Ames, C. Kesselman, A. W. Toga, S. G. Potkin, M. P. Vawter, , and F. Macciardi, "Next generation sequence analysis and computational genomics using graphical pipeline workflows," 2012.

[53] G. H. Fernald, E. Capriotti, R. Daneshjou, K. J. Karczewski, and R. B. Altman, "Bioinformatics challenges for personalized medicine," *Bioinformatics*, vol. 27, no. 13, pp. 1741–1748, 2011. [Online]. Available: +http://dx.doi.org/10.1093/bioinformatics/btr295

[54] C. Trapnell and S. L. Salzberg, "How to map billions of short reads onto genomes," *Nature Biotechnology*, vol. 27, no. 5, pp. 455–457, 2009.

[55] B. J. Raphael, "Chapter 6: Structural variation and medical genomics," *PLOS Computational Biology*, vol. 8, no. 12, pp. 1–11, 12 2012. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1002821

[56] L. Feuk, A. R. Carson, and S. W. Scherer, "Structural variation in the human genome," *Nature Reviews Genetics*, 2006.

[57] S. T. Sherry, M. H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin, "dbSNP: the NCBI database of genetic variation." *Nucleic acids research*, vol. 29, no. 1, pp. 308–311, Jan. 2001. [Online]. Available: http://dx.doi.org/10.1093/nar/29.1.308

[58] P. D. Stenson, E. V. Ball, M. Mort, A. D. Phillips, J. A. Shiel, N. S. Thomas, S. Abeysinghe, M. Krawczak, and D. N. Cooper, "Human gene mutation database (hgmd®): 2003 update," vol. 21, no. 6, pp. 577–581, 2003, exported from https://app.dimensions.ai on 2018/09/05. [Online]. Available: https://app.dimensions.ai/details/publication/pub.1017002566andhttp://onlinelibrary.wiley.com/doi/10.1002/humu.10212/pdf

[59] N. Bindal, S. A. Forbes, D. Beare, P. Gunasekaran, K. Leung, C. Y. Kok, M. Jia, S. Bamford, C. Cole, S. Ward, J. Teague, M. R. Stratton, P. Campbell, and A. P. Futreal, "Cosmic: the catalogue of somatic mutations in cancer," *Genome Biology*, vol. 12, no. 1, p. P3, Sep 2011. [Online]. Available: https://doi.org/10.1186/1465-6906-12-S1-P3

[60] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing." *Briefings in bioinformatics*, vol. 11, no. 5, pp. 473–83, 2010.

[61] N. a. Fonseca, J. Rung, A. Brazma, and J. C. Marioni, "Tools for mapping high-throughput sequencing data: Supplement," *Bioinformatics*, pp. 1–9, 2012.

[62] D. M. Church, V. A. Schneider, T. Graves, K. Auger, F. Cunningham, N. Bouk, H.-C. Chen, R. Agarwala, W. M. McLaren, G. R. Ritchie, D. Albracht, M. Kremitzki, S. Rock, H. Kotkiewicz, C. Kremitzki, A. Wollam, L. Trani, L. Fulton, R. Fulton, L. Matthews, S. Whitehead, W. Chow, J. Torrance, M. Dunn, G. Harden, G. Threadgold, J. Wood, J. Collins, P. Heath, G. Griffiths, S. Pelan, D. Grafham, E. E. Eichler, G. Weinstock, E. R. Mardis, R. K. Wilson, K. Howe, P. Flicek, and T. Hubbard, "Modernizing reference genome assemblies," *PLOS Biology*, vol. 9, no. 7, pp. 1–5, 07 2011. [Online]. Available: https://doi.org/10.1371/journal.pbio.1001091

[63] G. Highnam, J. J. Wang, D. Kusler, J. Zook, V. Vijayan, N. Leibovich, and D. Mittelman, "An analytical framework for optimizing variant discovery from personal genomes." *Nature communications*, vol. 6, p. 6275, 2015.

[64] J. M. Zook and et al., "Extensive sequencing of seven human genomes to characterize benchmark reference materials," *bioRxiv*, p. 26468, 2015.

[65] T. Derrien, J. Estellé, S. M. Sola, D. G. Knowles, E. Raineri, R. Guigó, and P. Ribeca, "Fast computation and applications of genome mappability," *PloS one*, vol. 7, no. 1, p. e30377, 2012.

[66] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, 2010.