Cluster Comput (2017) 20:1909–1924
DOI 10.1007/s10586-017-1015-0

A performance comparison of data and memory allocation strategies for sequence aligners on NUMA architectures

Josefina Lenis[1] · **Miquel Angel Senar**[1]

Received: 11 December 2016 / Revised: 29 May 2017 / Accepted: 23 June 2017 / Published online: 6 July 2017

Abstract Over the last several years, many sequence alignment tools have appeared and become popular for the fast evolution of next generation sequencing 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 have performed experiments with several popular sequence alignment tools on two widely available NUMA systems to assess the performance of different memory allocation policies and data partitioning 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 strategy that provides the best performance when a large number of processors and memory banks are used. In the case of data partitioning, the best results are usually obtained when the number of partitions used is greater, sometimes combined with an interleave policy.

Keywords NUMA · Data partitioning · Memory system performance · Genomic aligners · BWA · Bowtie · GEM · SNAP

1 Introduction

New genomic sequencing technologies have made a dramatic breakthrough in the development of genomic studies. The steady trend of reducing the sequencing cost and increasing the length of reads forces developers to create and maintain faster, updated and more accurate software. Sequence alignment tools have become essential for solving genomic variant calling studies. Numerous sequence alignment tools have been developed in recent years. They exhibit differences in sensitivity or accuracy [22] and most of them can execute in parallel on modern multicore systems. In general, writing parallel programs that exhibit good scalability on Non-uniform memory access (NUMA) architectures is far from easy. Achieving good system performance requires computations to be carefully designed in order to harmonize the execution of multiple threads and data accesses over multiple memory banks.

This work has been supported by MINECO-Spain under contract TIN2014-53234-C2-1-R.

✉ Josefina Lenis
josefina.lenis@uab.cat

Miquel Angel Senar
miquelangel.senar@uab.cat

[1] Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain

This paper is aligned with two previous works where we studied performance problems of alignment tools on NUMA systems [14,15]. In the first one, we analyzed the performance of Burrows-Wheeler Aligner, (BWA-ALN) [17], where we detected scalability problems exhibited by BWA-ALN, and we proposed simple system-level techniques to alleviate them. We obtained results up to fourfold speed higher than the original BWA-ALN multithread implementation. In the second work, 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 [16] (a newer version of BWA-ALN especially suited to dealing with longer reads), BOWTIE2 [13] (an ultrafast and memory-efficient tool for aligning sequencing reads to long reference sequences), GEM (GEnome Multi-tool) [19] and SNAP (Scalable Nucleotide Alignment Program) [24]. 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 this 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 novel hybrid execution strategy that combines both data partitioning and memory allocation policies.

This paper makes the following contributions:

1. We propose a general framework to define hybrid data and memory allocation policies that can be applied to parallel applications that use large data structures shared by all threads.
2. We provide a comprehensive set of experiments in two different NUMA systems to complete the performance comparison of several memory allocation policies and data partitioning strategies for four representative alignment tools.

The paper is structured as follows. Section 2 presents related work. Section 3 describes the basic concepts of NUMA systems and provides concrete details of the two systems used in our experiments. Section 4 introduces the problem of sequence alignment and a behavioral characterization of aligners used in this study. In Sect. 5, we introduce the methodology and all data distribution scenarios used to evaluate the performance improvement of aligners under study. Section 6 shows the results obtained in our experiments. The last section summarizes the main conclusions of our work.

2 Related work

Challenges in memory access on NUMA systems have been addressed by some approaches that tried to optimize locality at the OS level. The recent AutoNUMA patches for Linux [3] implement locality-driven optimizations 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 [4,8] 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). Both AutoNUMA and Carrefour 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.

Genome alignment problems have been considered by Misale et al. [20]. The authors implement a framework to work under BOWTIE2 and BWA to improve the local affinity of the original algorithm. Herzeel et al. [9] 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 2× performance improvement over the original BWA. In both cases—Misale et al. [20] and Herzeel et al. [9]—the source code of the applications -aligners- are modified, which might be a costly action and dependent on the application version. Abuin et. al. [1] 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. We present user-level guidelines of execution that help improve memory-bound aligners without modifying their source code, and, in some cases, without increasing the application initial requirements. Our contribution is based on the idea that application performance can be improved taking into account architecture characteristics and an application's memory footprint.

3 NUMA systems

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 [7]. 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. This is mainly, because creating a program that uses NUMA memory subsystems efficiently is not a trivial task. As an example of NUMA systems, we can see the following two figures (Figs. 1 and 2) that represent the two architectures we employed in this study, one manufactured by AMD and the other by Intel.

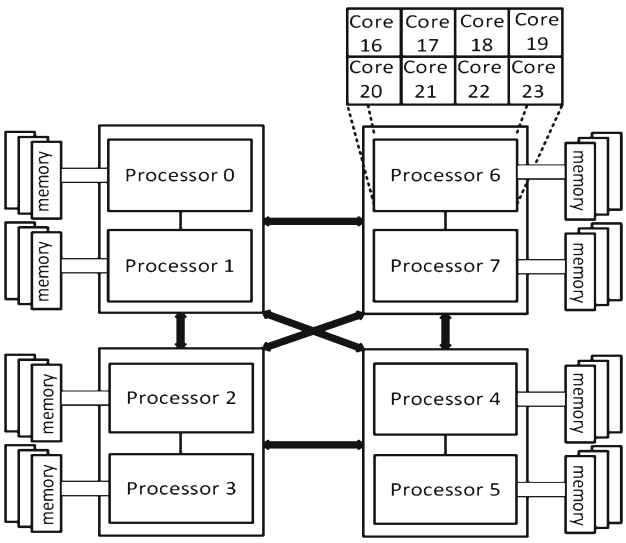

Fig. 1 Schematic diagram of the AMD Opteron 6376 architecture (Abu-Dhabi)

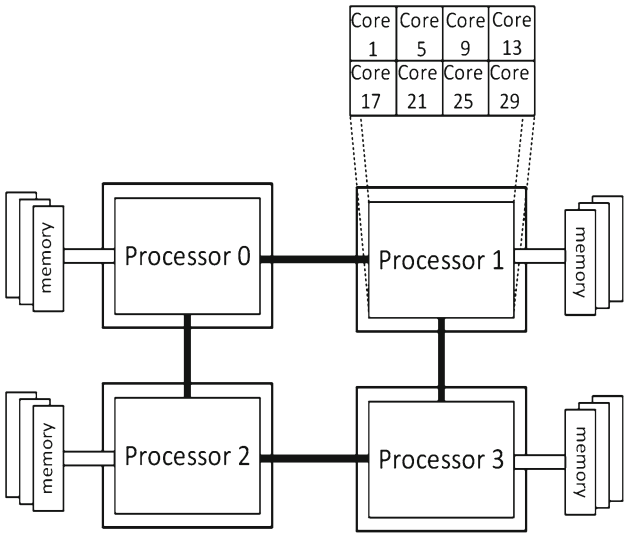

Fig. 2 Schematic diagram of the Intel Xeon E5 4620 architecture (Sandy Bridge)

The first system, shown in Fig. 1, 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 (Fig. 2) 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 four 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 [2].

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 in distant 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. 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 [12]. 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 [11] 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 [11]. *Numactl* can be used to control process mapping to cpuset and restrict memory allocation to specific nodes without altering the program's source code.

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 BWT [18]. In hash table based algorithms, given a query P, every substring of length s of it is hashed, and can later be easily retrieved. 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. 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 in the genome [18]. 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 [23]. 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 [6]. In [20], 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.

5 Allocation strategies and data partitioning

The experiments carried out in this study are the product of a series of systematics tests designed to evaluate the behavior of aligners in different 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. In this work, we extend the second part, adding new configurations consisting of combining instance creation with different memory allocation policies. Details of these two schemes are presented below.

5.1 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 three 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 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 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 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.

Table 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

In this study, we introduce a novel hybrid execution technique combining the partitioning techniques with the memory allocation policies explained in Sect. 5.1: localalloc and interleave. Figure 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 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 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 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.

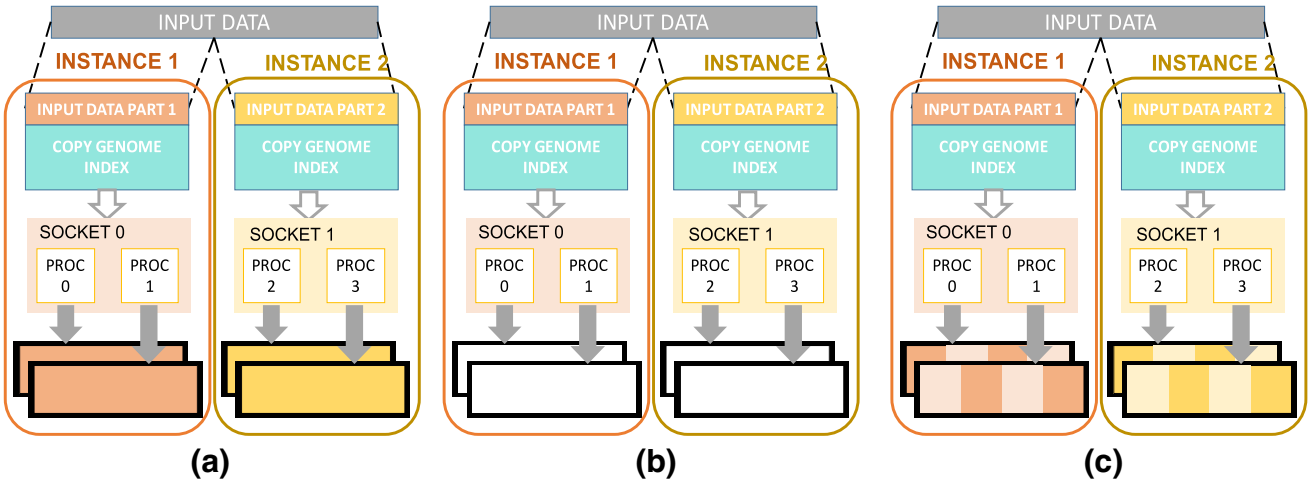

Fig. 3 Hybrid experimentation. **a** Partitioning, **b** Partitioning + Localalloc, **c** Partitioning + Interleave

6 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, maintained by The Genome Reference Consortium, and two data sets were used as input data:

– *Synthetic benchmark* [10]:

Single end, base length = 100, number of reads= 11M Size = 3.1GB

– *Segment extracted from NA12878* [25]:

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 1. Results were obtained as an average of ten executions. Figures 4, 5, 6 and 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 A1:, B1:, C1: and D1:, annexed at the end of this article.

6.1 Analysis of memory allocation policies

In Figures 4, 5, 6 and 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 Sect. 3 (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 [12]. 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 4, 5, 6 and 7), scalability decreases significantly beyond 32 threads in all four aligners. When aligners run on more than 32 cores, at least one NUMA node at two-hops distance is used. Therefore, all the speed up 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

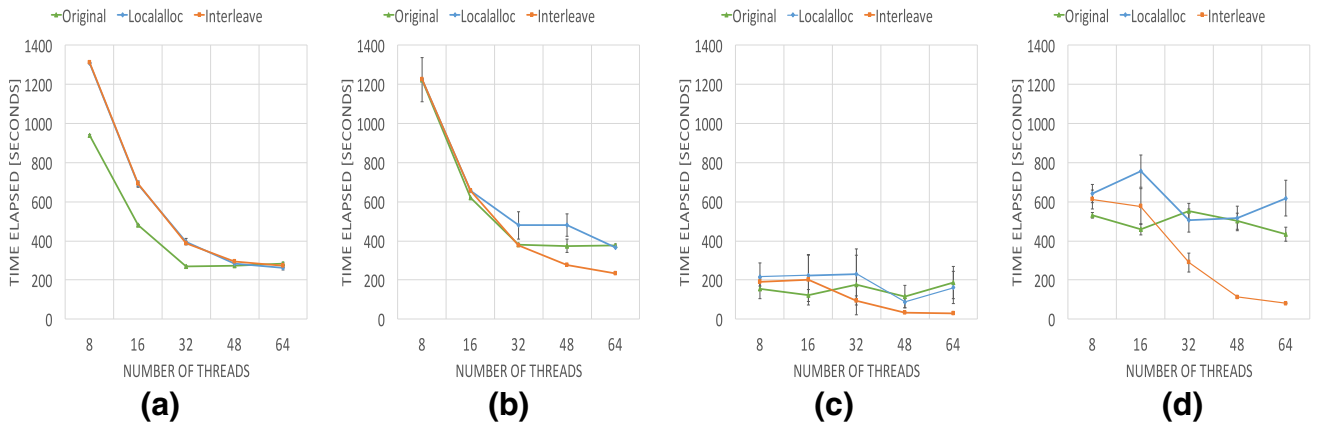

Fig. 4 Different memory allocation policies. The lower the better. Arch: AMD. Dataset: GCAT Synthetic Input. **a** BOWTIE2, **b** BWA-MEM, **c** GEM and **d** SNAP

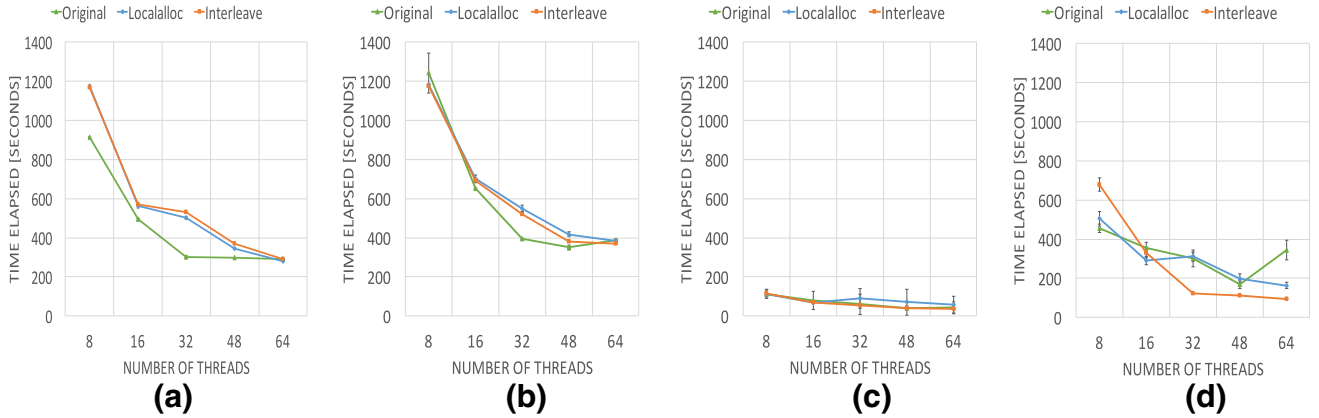

Fig. 5 Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: GCAT Synthetic Input **a** BOWTIE2, **b** BWA-MEM, **c** GEM and **d** SNAP

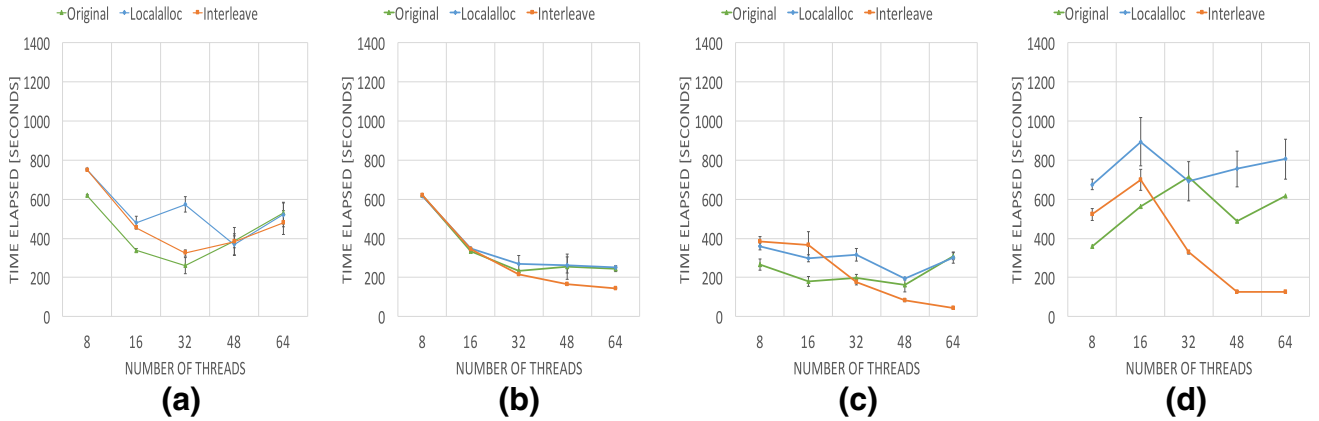

Fig. 6 Different memory allocation policies. The lower the better. Arch: AMD. Dataset: NA12878 Real Input. **a** BOWTIE2, **b** BWA-MEM, **c** GEM and **d** SNAP

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.

As explained in Sect. 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

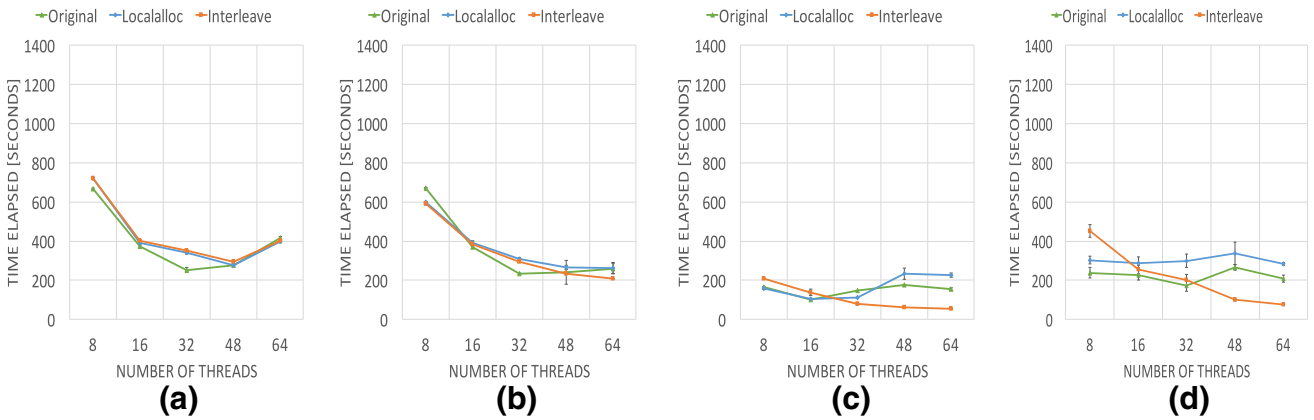

Fig. 7 Different memory allocation policies. The lower the better. Arch: INTEL. Dataset: NA12878 Real Input. **a** BOWTIE2, **b** BWA-MEM, **c** GEM and **d** SNAP

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 in these architectures due to reduced memory bandwidth between NUMA nodes [21]. 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 Sect. 3: 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 [5]. Not all aligners process the queries in the same way, and the work done is irregular. The 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 multithreads 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.

6.2 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. Once the number of instances is determined, a memory allocation policy is applied, thus generating the hybrid scenarios described in Sect. 5.2. In Table 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 Figs. 8, 9, 10 and 11, appears in the column *Names*.

Figures 8, 9, 10 and 11 show a complete speed up 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). Speed up 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 1.5× and 5× 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 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

Table 2 Instances created for each aligner

Aligner	#Inst × Threads	Policy	Name
	2 × 32t	Original	2Inst_Original
	4 × 16t		4Inst_Original
	8 × 8t		8Inst_Original
BOWTIE2	2 × 32t	Localalloc	2Inst_Localalloc
BWA-MEM	4 × 16t		4Inst_Localalloc
	2 × 32t	Interleave	2Inst_Interleave
	4 × 16t		4Inst_Interleave
SNAP	2 × 32t	Original	2Inst_Original
GEM	2 × 32t	Localalloc	2Inst_Localalloc
	2 × 32t	Interleave	2Inst_Interleave

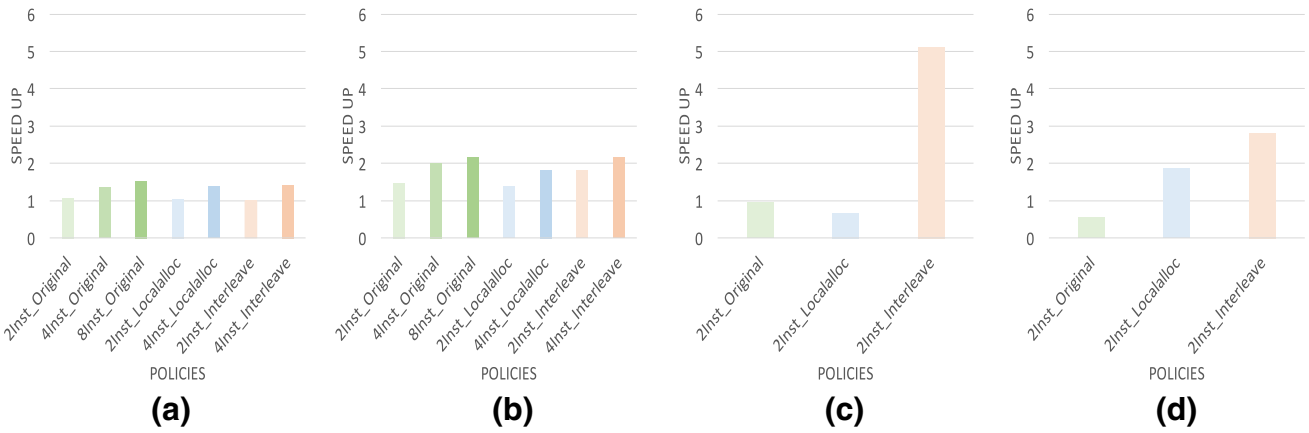

Fig. 8 Speed up comparison between all allocation strategies. The higher the better. Arch: AMD. Dataset: GCAT Synthetic Input. **a** BOWTIE2, **b** BWA-MEM, **c** GEM and **d** SNAP

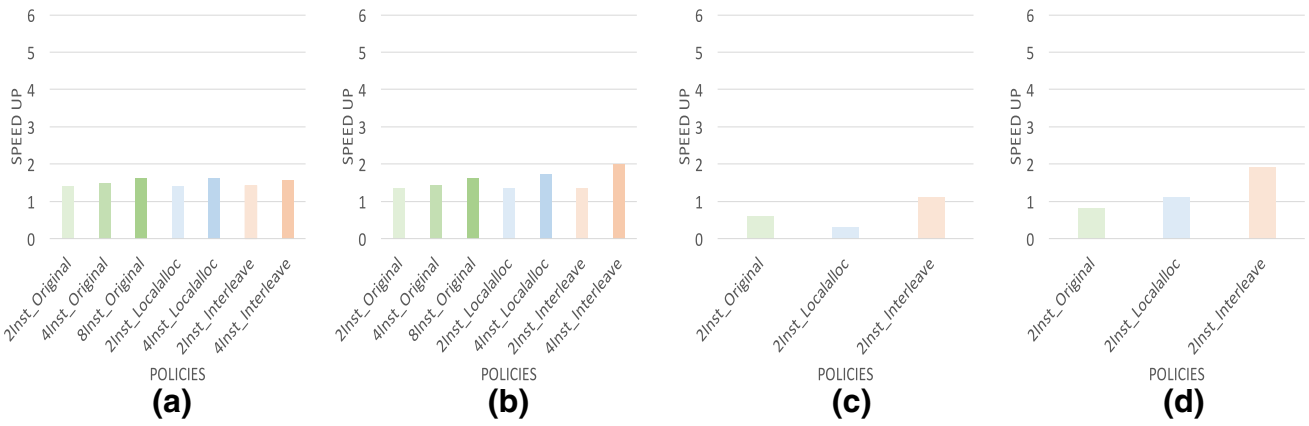

Fig. 9 Speed up comparison between all allocation strategies. The higher the better. Arch: INTEL. Dataset: GCAT Synthetic Input. **a** BOWTIE2, **b** BWA-MEM, **c** GEM and **d** SNAP

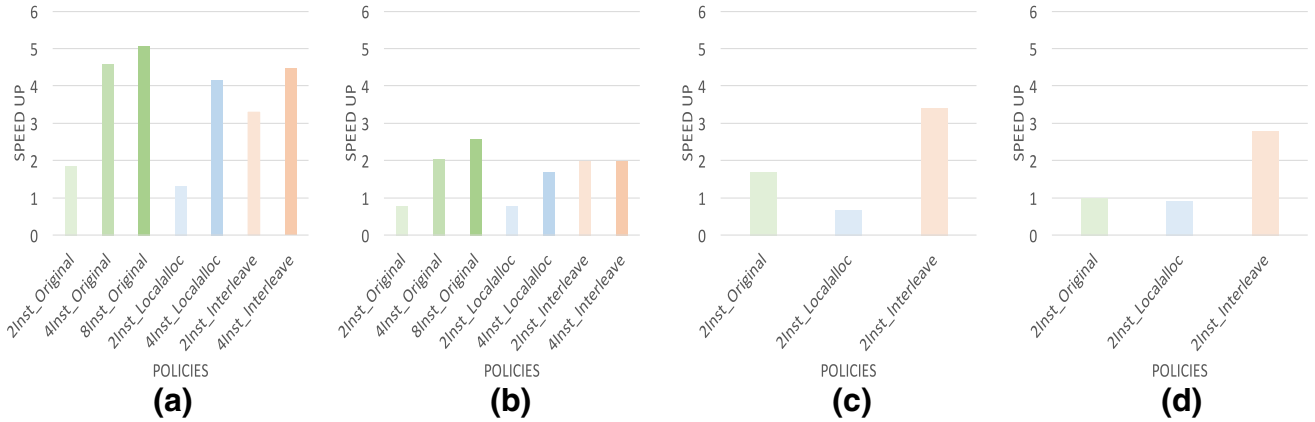

Fig. 10 Speed up comparison between all allocation strategies. The higher the better. Arch: AMD. Dataset: NA12878 Real Input. **a** BOWTIE2, **b** BWA-MEM, **c** GEM and **d** SNAP

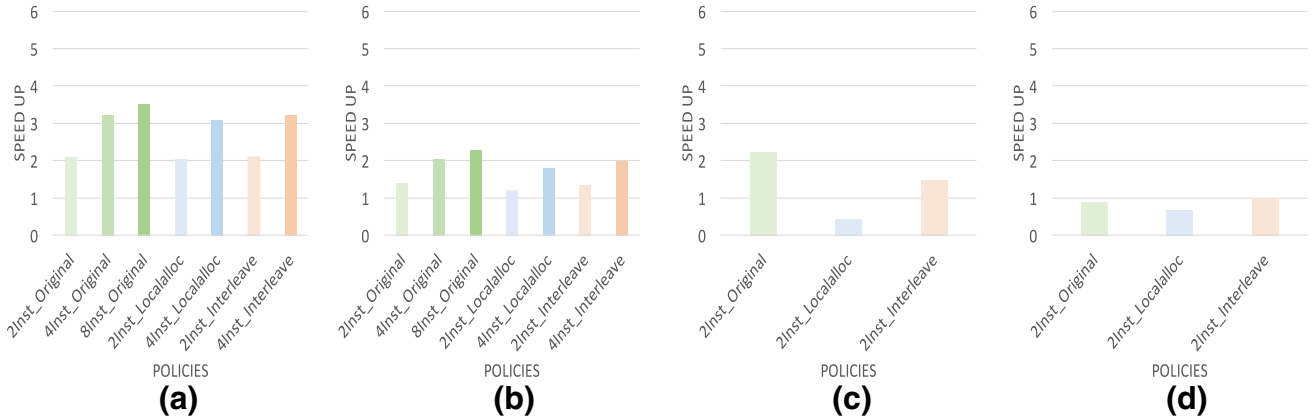

Fig. 11 Speed up comparison between all allocation strategies. The higher the better. Arch: INTEL. Dataset: NA12878 Real Input. **a** BOWTIE2, **b** BWA-MEM, **c** GEM and **d** SNAP

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 time up to 5× 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.

6.3 Summary results

Figure 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 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 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 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

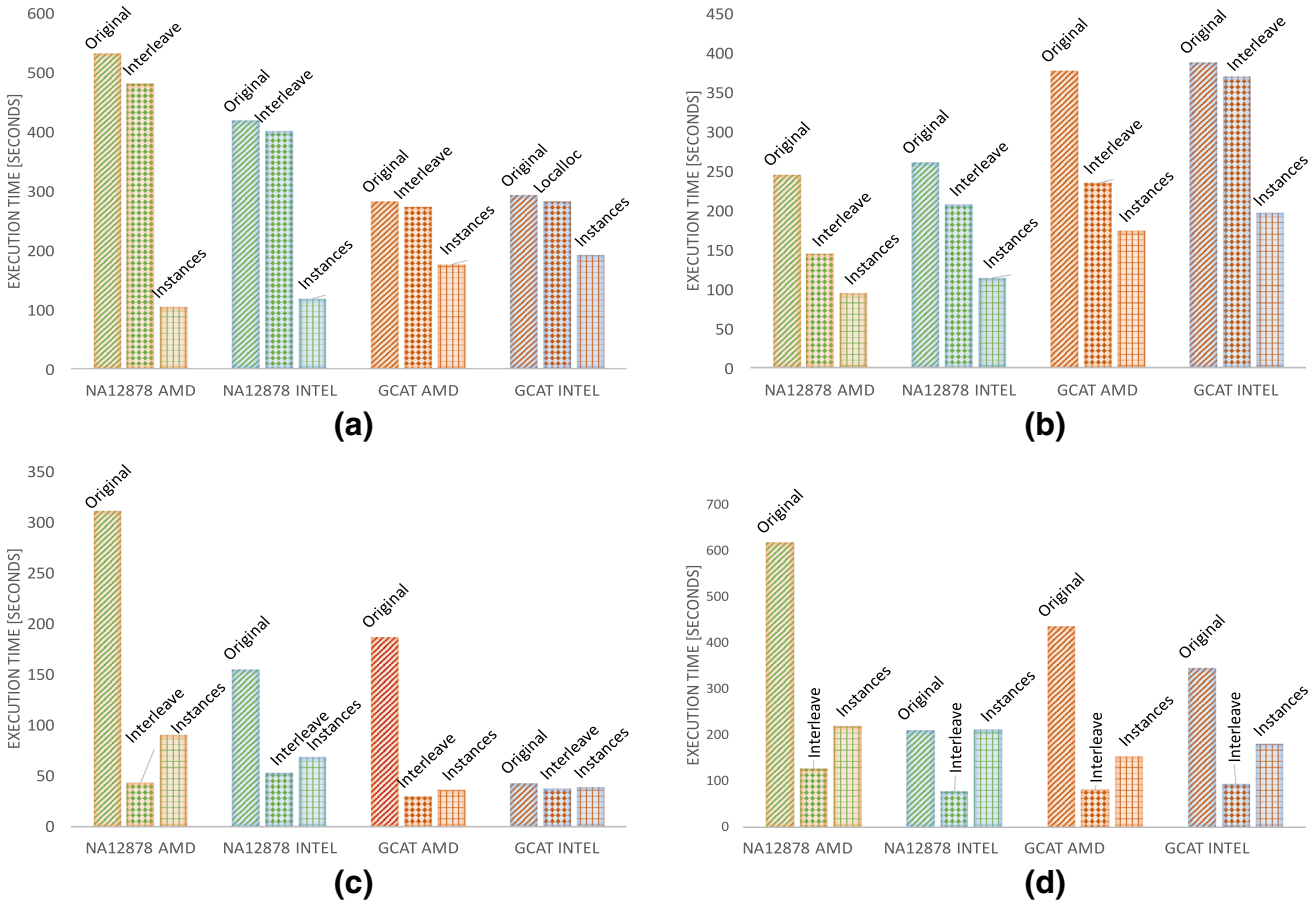

Fig. 12 Summary results for all aligners. **a** BOWTIE2, **b** BWA-MEM, **c** GEM and **d** SNAP

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.

7 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. In this paper, 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 rises, 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 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 inter-

leaving if the size of the memory banks is not large enough to hold genome indexes.

Improvements in execution times of 5× and 2.5× 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 5× and 2.8× 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.

Appendix A: Complete set of execution times for BOWTIE2

Table A1: Execution times for BOWTIE2 using different memory allocation policies

Threads	BOWTIE2											
	AMD											
	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
	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
Threads	INTEL											
	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
	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 A2: Execution times for BOWTIE2 using data partitioning

Name	Threads	BOWTIE2							
		AMD				INTEL			
		NA12878		GCAT		NA12878		GCAT	
		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

Appendix B: Complete set of execution times for BWA-MEM

Table B1: Execution times for BWA-MEM using different memory allocation policies

Threads	BWA-MEM											
	AMD											
	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
	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
Threads	INTEL											
	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
	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 B2: Execution times for BWA-MEM using data partitioning

Name	Threads	BWA-MEM							
		AMD				INTEL			
		NA12878		GCAT		NA12878		GCAT	
		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

Appendix C: Complete set of execution times for GEM

Table C1: Execution times for GEM using different memory allocation policies

Threads	GEM											
	AMD											
	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
	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
Threads	INTEL											
	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
	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 C2: Execution times for GEM using data partitioning

Name	Threads	GEM							
		AMD				INTEL			
		NA12878		GCAT		NA12878		GCAT	
		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

Appendix D: Complete set of execution times for SNAP

Table D1: Execution times for SNAP using different memory allocation policies

Threads	SNAP											
	AMD											
	NA12878						GCAT					
	Original		Localalloc		Interleave		Original		Localalloc		Interleave	
	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 D2: Execution times for SNAP using data partitioning

Name	Threads	SNAP							
		AMD				INTEL			
		NA12878		GCAT		NA12878		GCAT	
		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

References

1. Abuín, J.M., Pichel, J.C., Pena, T.F., Amigo, J.: BigBWA: approaching the Burrows-Wheeler aligner to Big Data technologies. Bioinformatics **31**(24), 4003–4005 (2015). doi:10.1093/bioinformatics/btv506
2. Braithwaite, R., McCormick, P., Feng, W.C.: Empirical memory-access cost models in multicore numa architectures. Virginia Tech Department of Computer Science (2011)
3. Corbet, J.: AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709 (2012)
4. Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., Mark, R.: Traffic management: a holistic approach to memory placement on NUMA systems. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 381–394 (2013)
5. Derrien, T., Estellé, J., Sola, S.M., Knowles, D.G., Raineri, E., Guigó, R., Ribeca, P.: Fast computation and applications of genome mappability. PloS ONE **7**(1), e30 (2012)
6. Fonseca, N.A., Rung, J., Brazma, A., Marioni, J.C.: Tools for mapping high-throughput sequencing data. Bioinformatics **28**(24), 3169–3177 (2012)
7. García-Risueño, P., Ibañez, P.E.: A review of high performance computing foundations for scientists. Int. J. Mod. Phys. C **23**(07), 1–33 (2012). doi:10.1142/S0129183112300011
8. Gaud, F., Lepers, B., Funston, J., Dashti, M., Fedorova, A., Quema, V., Lachaize, R., Mark, R.: Challenges of memory management on modern NUMA systems. Commun. ACM **58**, 59–66 (2015)
9. Herzeel, C., Ashby, T.J., Costanza, P., Meuter, W.D.: Resolving load balancing issues in BWA on NUMA multicore architectures. In: 10th International Conference PPAM 2013, vol. 8385, pp. 227–236. Springer, Berlin, Heidelberg (2014). doi:10.1007/978-3-642-55195-6
10. Highnam, G., Wang, J.J., Kusler, D., Zook, J., Vijayan, V., Leibovich, N., Mittelman, D.: An analytical framework for optimizing variant discovery from personal genomes. Nat. Commun. **6**, 6275 (2015). doi:10.1038/ncomms7275
11. Kleen, A.: An NUMA API for Linux. Tech. Rep. 2, SUSE Labs (2004)
12. Lameter, C., Hsu, B., Sosnick-Pérez, M.: NUMA (Non-uniform memory access): an overview. ACMQueue **11**, 1–12 (2013)
13. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nat. Methods **9**(4), 357–359 (2012). doi:10.1038/nmeth.1923
14. Lenis, J., Senar, M.A.: On the Performance of BWA on NUMA Architectures. In: 2015 IEEE Trustcom/BigDataSE/ISPA, pp. 236–241 (2015). doi:10.1109/Trustcom.2015.638
15. Lenis, J., Senar, M.A.: Optimized execution strategies for sequence aligners on NUMA architectures. In: 2016 Springer LNCS/EUROPAR_PBIO (2016)
16. Li, H.: Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. (2013). arXiv:1303.3997
17. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics **25**(14), 1754–1760 (2009). doi:10.1093/bioinformatics/btp324
18. Li, H., Homer, N.: A survey of sequence alignment algorithms for next-generation sequencing. Brief. Bioinform. **11**(5), 473–483 (2010). doi:10.1093/bib/bbq015
19. Marco-Sola, S., Sammeth, M., Guigó, R., Ribeca, P.: The GEM mapper: fast, accurate and versatile alignment by filtration. Nat. Methods **9**, 1185–1188 (2012). doi:10.1038/nmeth.2221
20. Misale, C., Ferrero, G., Torquati, M., Aldinucci, M.: Sequence alignment tools: One parallel pattern to rule them all? BioMed Res. Int. **2014** (2014). doi:10.1155/2014/539410
21. Molka, D., Hackenberg, D., Schöne, R.: Main memory and cache performance of intel sandy bridge and amd bulldozer. In: Workshop on Memory Systems Performance and Correctness, MSPC '14, pp. 4:1–4:10. ACM, NY, USA (2014). doi:10.1145/2618128.2618129
22. Shang, J., Zhu, F., Vongsangnak, W., Tang, Y., Zhang, W., Shen, B.: Evaluation and comparison of multiple aligners for next-generation sequencing data analysis. BioMed Res. Int. **2014**, 16 (2014). doi:10.1155/2014/309650
23. Trapnell, C., Salzberg, S.L.: How to map billions of short reads onto genomes. Nat. Biotechnol. **27**(5), 455–457 (2009)
24. Zaharia, M., Bolosky, W., Curtis, K.: Faster and more accurate sequence alignment with SNAP, pp. 1–10 (2011). arXiv:1111.5572v1
25. Zook, J.M., et al.: Extensive sequencing of seven human genomes to characterize benchmark reference materials. bioRxiv p. 26468 (2015). doi:10.1101/026468

Josefina Lenis received the BSc degree in computer science in 2012 from the Universidad Nacional de Tucumán and the MSc degree in high-performance computing and information theory in 2013 from the Universitat Autonoma de Barcelona (UAB). She is currently doing a PhD degree in high-performance computing at the UAB, working on optimizing performance of genomic aligners on NUMA architectures, by studying the impact of mapping algorithms and data allocation policies. Her research interests include computer architecture and parallel optimizations for heterogeneous HPC systems.

Miquel Angel Senar got a BS degree in Computer Science and a PhD degree in Computer Science from Universitat Autònoma de Barcelona (UAB) in 1988 and 1996, respectively. Since 2007 he is Full Professor at the department of Computer Architecture and Operating Systems at the UAB, where he teaches subjects related to computer architecture and parallel programming. Since 1988, he has been participating in national and international projects working on different aspects related to parallel and distributed systems. His current research interests focus on scheduling and resource management for parallel applications, performance engineering techniques applied to massively parallel architectures, and high performance computing in bioinformatics.