Optimized Execution Strategies for Sequence Aligners on NUMA Architectures

Josefina Lenis and Miquel Angel Senar(✉)

Universitat Autònoma de Barcelona (UAB), 08193 Bellaterra, Spain
{josefina.lenis,miquelangel.senar}@uab.es

Abstract. Alignment applications are essential for solving genomic variant calling studies. We have analyzed performance problems of four popular aligners from the literature. They constitute representative examples of the two most commonly used algorithmic strategies: hash tables and Burrows-Wheeler Transform. Although they take advantage of multithreading execution, they exhibit significant scalability limitations on systems with a non-uniform memory architecture (NUMA). Data sharing between independent threads and irregular memory access patterns constitute performance limiting factors that affect the studied aligners. We have also evaluated various data distribution strategies that do not require changes to the applications. Significant improvements in speedup were achieved when these techniques were applied to the execution of these aligners on a NUMA system.

Keywords: NUMA · Memory system performance · Genomic aligners · NGS

1 Introduction

New sequencing technologies set the pace of the rapid progress in genomic studies. The steady trend of reducing the sequencing cost and increasing the length of reads force developers to create and maintain more accurate, faster and updated software. Numerous sequence aligning tools have been developed in recent years. They exhibit differences in sensitivity or accuracy [16] and most of them can execute in parallel in modern multicore systems. In general, writing parallel programs that exhibit good scalability on non-uniform memory architectures (NUMA) is far from easy. Achieving good system performance requires that computations are carefully designed in order to harmonize execution of multiple threads and data accesses over multiple memory banks.

This paper is aligned with our previous work where we analyzed the performance of BWA-ALN, (Burrows-Wheeler Aligner) [11], on NUMA architectures. In that study, 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 up over original BWA-ALN multithread implementation. In the present work, we extend the study to BWA-MEM [10] (a newer version of

F. Desprez et al. (Eds.): Euro-Par 2016 Workshops, LNCS 10104, pp. 492–503, 2017.
DOI: 10.1007/978-3-319-58943-5_40

BWA specially suited to deal with longer reads) and to other three well-known mappers, namely, BOWTIE2 [8], GEM [13] and SNAP [18]. These mappers are widely used by the scientific community and real production centers, and frequently updated by its developers. We have applied various data distribution strategies to these mappers, as we did with BWA-ALN, and we obtained promising results on all cases, reducing memory-bound drawbacks and increasing scalability.

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

2 Related Work

Genome alignment problems have been considered by Misale [14]. The author implements a framework to work under BOTWIE2 and BWA improving local affinity of the original algorithm. Herzeel *et al.* [4] replaces the pthread-based parallel loop in BWA by 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. On both cases - Misale 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.* [1] presented a big data approach to solve BWA scalability problems. They introduce a tool name BigData that enables to run BWA in several machines although it does not provide a clear strategy to divide the data or to set the number of instances. In contrast, our approach can be applied to different aligners with minimum 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 improving 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 application's memory footprint.

3 NUMA Systems

In NUMA systems, main memory is physically distributed in banks among different processors but it looks like one single large memory from a logical perspective, so accesses to different parts is done using global memory addresses [3]. Each processor has its own memory bank and can access to it through its memory controller. A processor and its respective memory bank is called NUMA node. A program running in a particular processor can also access data stored in memory banks associated to other processors in a coherent way but at the cost of increased latency compared to accesses to its own local memory bank. In general, parallel applications that may run using multiple processors are not usually designed taking into account the NUMA architecture. Mainly, because creating a program that uses efficiently NUMA memory subsystems is not a trivial task. Figure 1 shows an example of NUMA architecture that corresponds to the system that we used in this study.

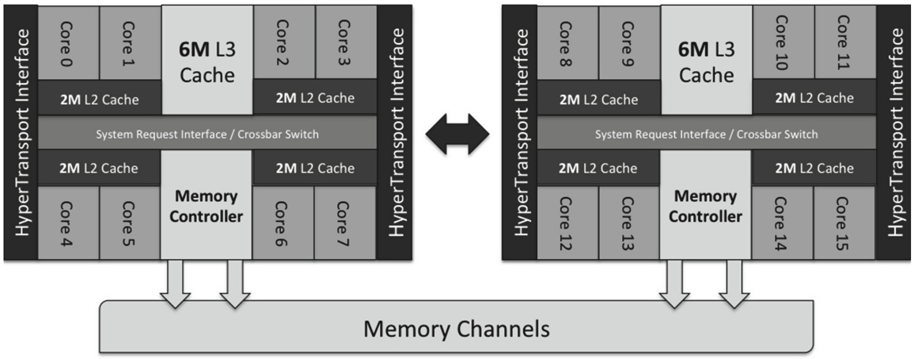


Fig. 1. AMD Bulldozer micro-architecture

It is a four-socket AMD Opteron Processor 6376 (Bulldozer microarchitecture), 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 128 GB of memory, divided into 8 modules of 16 GB DDR3 1600 MHz each. Nodes are connected by HyperTransport links. Information about the NUMA system configuration can be retrieved on Linux systems by using the *numactl – hardware* command. This command displays the available nodes and access costs to different NUMA nodes. As seen in Table 1, access (or distance) costs within a local NUMA node is 10; this is shown in the diagonal values of the table. According to this information, access to an intermediate distance node costs 1.6x more, and access to the more distant nodes costs more than twice (2.2x). Distance between NUMA nodes is frequently referred to as hops. Where 0 hop is the minimum distance and 2 hops is the maximum.

Table 2 shows the results of a small experiment that we carried out on our system to bear out the accuracy of the information obtained. We modified an available open source benchmark [6], and adapted it to our architecture. The program is written in C and was compiled with GCC version 4.9.1, without optimization flags (−O0). The experiment consisted in reading an array of 100M elements. Each access was performed in such a way that prefetching was skipped

Table 1. Distance map on AMD 6376.

NUMA	0	1	2	3	4	5	6	7
0	10	16	16	22	16	22	16	22
1	16	10	22	16	16	22	22	16
2	16	22	10	16	16	16	16	16
3	22	16	16	10	16	16	22	22
4	16	16	16	16	10	16	16	22
5	22	22	16	16	16	10	22	16
6	16	22	16	22	16	22	10	16
7	22	16	16	22	22	16	16	10

Table 2. Bandwidth per thread [MiB/s]

Source	1 Thread	64 Threads
0	3300	700
1	2450	250
2	2200	330
3	1700	220
4	2200	250
5	1700	220
6	2200	330
7	1700	208

and a memory access (and a last level cache miss) was ensured every time. The array was allocated in node 0 and accessed by threads allocated in all cores (64). In the first column, we can see the bandwidth achieved when the array was accessed sequentially by one thread at a time. The "source" indicates to which NUMA node the thread was bound to. Second column shows bandwidth measurements when the same array was accessed by all the available threads at the same time. It is worth mentioning that the displayed values of the bandwidth correspond to the worst case scenarios. According to Table 2 accessing a local node is approximately 3300 MiB/s; bandwidth for medium distance nodes (1 hop) is 2200 MiB/s, the penalty being 1.5x; and accesses from a two-hops node incurs a penalty of 1.9x (bandwidth equals to 1700 MiB/s). Table 2 shows a case that is not revealed in Table 1: accesses from a thread running on the node located at the same socket exhibits a bandwidth of 2450 MiB/s (that might be seen as a node between 0 and 1 hop). Penalty in access latencies between processors and memory is one of the main problems suffered by NUMA-unaware applications. However, another problem arises when applications use a centralized data structure that is located in a single memory bank. When a large number of threads needs to access to this shared data structure, congestion problems might generate a significant degradation in memory accesses, as shown in the second column of Table 2.

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 Burrow Wheeler Transform (BWT) [12]. In hash table based algorithms, given a query P every substring of length s of it is hashed, and can be later easily retrieved. SNAP is an example of 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 [12]. Examples of BWT base 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 [17]. 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 [2]. In [14], Misale et al. defines 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 aligners that we used in this study. Therefore, continuous accesses to the single shared data structure -index- by all threads can increase its memory degradation performance. 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

In our previous work [9], we presented a series of execution strategies to improve BWA-ALN performance without modifying its source code. In this paper, we have applied our methodology to 4 aligners (GEM3, BOWTIE2, BWA-MEM and SNAP) in order to assess its benefits as a general methodology that can be applied to aligners that exhibit the features mentioned at the end of the previous section. We have developed a series of steps to characterize the behaviour of a memory-bound application and define its best execution strategy (see Fig. 2).

5.1 Analysis and Optimization of Shared Data Distribution (Part A)

Part A of our methodology consists in analyzing whether an aligner is sensitive to different memory allocations. In order to achieve this we carried out 3

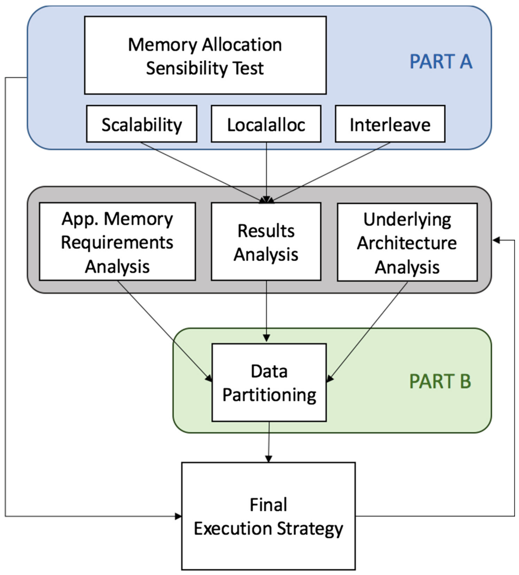


Fig. 2. Proposed methodology to find the best execution strategy

experiments: The first is a traditional scalability study in which we focused on 5 particular cases: using 8, 16, 32, 48 and 64 threads, because 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. For the other two cases we used the Linux Tool *numactl* to set a memory policy allocation. With the parameter – *localalloc* the data was allocated in the current node where the program is being executed. 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 using a round robin fashion between selected nodes. All 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 this part 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 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 Replication and Partitioning Strategies (Part B)

The objective for part B is to reduce the usage of the interconnection bus. This is achieved by data replication and partitioning techniques that imply the execution of simultaneous instances of the application (aligner). For aligners that have a small index as BOWTIE2 and BWA, data partitioning is not that challenging because the entire index fits in one memory bank. In these cases, we can consider each NUMA node as a symmetric multi-processor unit, capable of running an independent instance of an aligner. Independent instances were created, each one running in a single NUMA node (all independent instances were running with 8 threads). For GEM and SNAP we also run independent instances but the index does not fit in one NUMA node. Each one of these instances is multithread. The input file with all the reads was divided into the number of instances. Figure 3, illustrates this configuration with 4 independent instances that are being executed simultaneously. Input data is 1/4 the size of the original and the reference genome is replicated 4 times.

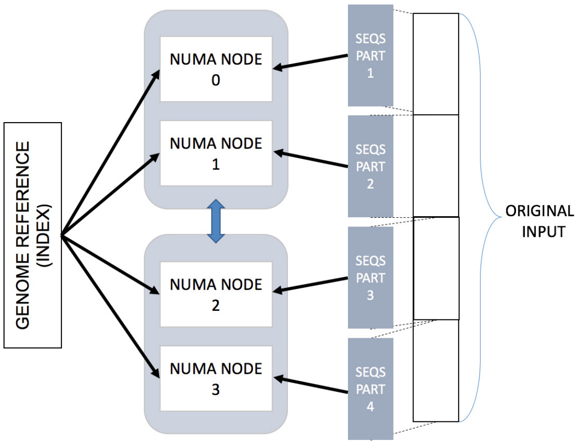


Fig. 3. Data partitioning

For aligners like GEM and SNAP, where the index size is equal or larger than the size of a memory bank, data partitioning becomes more complex because it involves more than one NUMA node. In Table 3, we can see the sizes of the indexes used. This information is crucial for designing how to split the data. Knowing the underlying architecture is also critical. Our system has memory banks of 16 GiB, which is not enough to run SNAP, even if two memory banks are used, the index would barely fit in. This is why we only run 2 simultaneous instances of SNAP (64 GiB each) and 4 instances of GEM (32 GiB each).

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* [5]:
  Single end, base length = 100, number of reads = 11M Size = 3.1 GB
- *Segment extracted from NA12878* [19]:
  Single end, base length = 100, number of reads = 22M Size = 5.4 GB

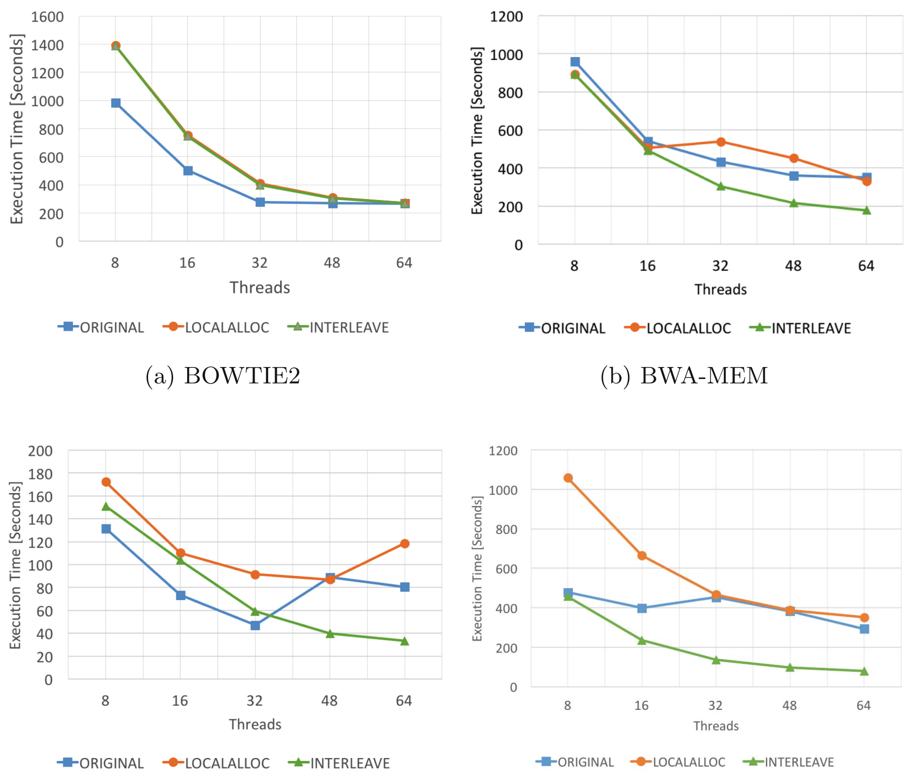
All aligners were compiled using GCC 4.9.1 and we used the latest version available for each, as shown in the second column of Table 3. All results were obtained as an average of five executions.

Table 3. Detailed information about the aligners.

Aligner	Version	Index (GB)	Data partitioning
BOWTIE2	2.2.6	3.9	8x8 threads
BWA-MEM	0.7.12	5.1	8x8 threads
GEM	3.0	15.0	4x16 threads
SNAP	1.0.18	29.0	2x32 threads

In the first part of our experimentation (Part A), we obtained the execution times of the four different aligners shown in Fig. 4. By original we refer to the execution of a given aligner with its default parameters without any particular allocation policy or NUMA control, and letting the operating system handle the allocation. On Linux systems this will normally involve spreading the threads through the system and using 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 [7]. Allocation policy takes effect only when a page is first requested by a process. If we focus on the original execution (shown by a light blue line in Fig. 4), 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 are used. Therefore, all the speed up gain due to multithreading is mitigated by the latency of remote accesses and traffic saturation of interconnection links. For aligners BWA-MEM Fig. 4b, GEM3 Fig. 4c and SNAP Fig. 4d it can be clearly seen that interleave policy reduces the execution time, specially for the limited scalability scenarios (with 48 and 64 threads).

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 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 specially prone to happen in this architecture due to reduced memory bandwidth between NUMA nodes [15]. This reason explains why using localalloc policy does not produce any improvement in



(a) BOWTIE2

(b) BWA-MEM

(c) GEM

(d) SNAP

Fig. 4. Different memory allocation policies. DATASET: synthetic benchmark (Color figure online)

execution times. BOWTIE2 Fig. 4a does not follow this trend; BOWTIE2 running on its defaults configuration performs better than using a explicit memory policy. We could infer some memory optimization might take place at the index load stage but a move precise analysis of the source code would be required to provide more accurate conclusions. In the second part of our experimentation (part B), we use data partitioning and data replication techniques to create multiple instances and run them simultaneously. We found, in our previous work, that this was the best solution for BWA-ALN. Figure 5 shows a complete comparison of all strategies, calculated using the wall time of the original execution with 64 threads (max resources).

Replication of the reference genome index reduces at the same time latency and contention problems while the benefits from multithreading parallelization are maintained: queries are distributed in different groups of threads that share a particular copy of the index stored in a local bank. BOWTIE2 and GEM also increase their performance when creating instances. Although for

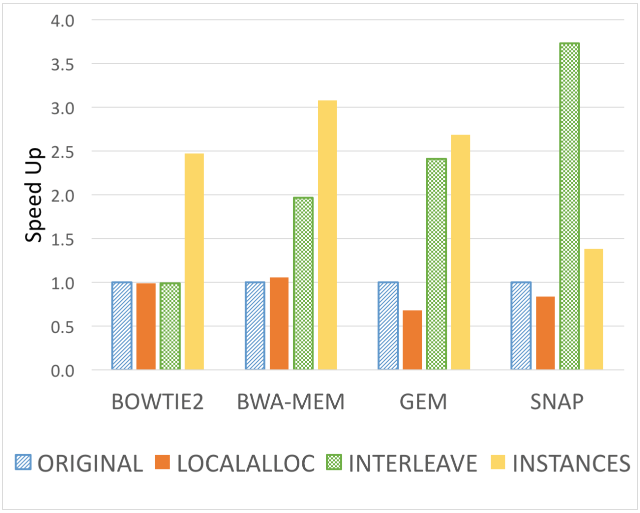


Fig. 5. All strategies. DATASET: synthetic benchmark

Table 4. Complete set of results for dataset NA12878

Aligner	Policy	Execution time [s]					SpeedUp
		Number of threads					Max. threads
		8	16	32	48	64	64
BOWTIE2	Original	679.89	361.20	223.74	279.13	431.11	1.1
	LocalAlloc	826.56	476.09	296.53	305.50	433.93	0.99
	Interleave	834.58	486.59	314.99	290.75	471.62	0.914
	Instances	–	–	–	–	111.11	3.38
BWA-MEM	Original	537.77	340.97	312.95	315.26	307.18	1
	LocalAlloc	618.67	329.11	344.84	354.91	296.83	1.03
	Interleave	482.20	300.89	233.23	193.21	170.23	1.80
	Instances	–	–	–	–	61.93	4.96
GEM3	Original	246.30	132.56	80.82	66.016	60.94	1
	LocalAlloc	405.15	256.65	303.17	202.25	273.75	0.22
	Interleave	327.24	187.32	100.40	64.74	52.58	1.21
	Instances	–	–	–	–	57.46	1.06
SNAP	Original	465.72	237.83	218.67	297.17	297.17	1
	LocalAlloc	705.76	923.40	393.85	343.18	361.02	0.92
	Interleave	199.97	396.54	98.43	79.04	65.27	4.5
	Instances	–	–	–	–	223.39	1.33

GEM this strategy is only slightly better than using interleave. This can be explained because the index does not fit into one memory bank; even when multiple instances are created inter-node traffic can not be avoided. A similar result is observed with SNAP where instances represent an improvement of 40% over Original SNAP but its large index data structure forces it to use four NUMA nodes per instance and to have two-hops distance accesses. Complementary results for dataset NA12878 can be found in Table 4. These results are in line with the results discussed above.

7 Conclusions

Knowing the underlying architecture where applications are running is a key aspect to achieve their optimal performance. If an application is memory-bound, might suffer drawback in performance when executed in NUMA systems. In this paper, we presented an approach to detect whether an aligner is being penalized by contention or/and remote memory bank accesses and whether it is susceptible to improve its execution time, by applying some simple system-level techniques that do not require changes on the original application code. When *interleave* or *instances* based techniques were applied, execution time was reduced in all cases tested. Aligners in which the index size is less than half the size of a single memory bank, data partitioning arises as the best solution because it completely avoids traffic between nodes and ensures only local accesses. In this case, a speedup of 2.5x and 3.1x was obtained for BOWTIE2 and BWA-MEM respectively. It is noteworthy that *instances* based implies an increment on memory requirements. BOWTIE2 and BWA-MEM can easily meet this requirement in modern systems. For other aligners with larger indexes (i.e. SNAP and GEM), *interleave* technique might 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 albeit HyperTransport traffic cannot be reduced. Improvements of 2.5x and for SNAP is 3.6x were obtained for SNAP and GEM, respectively. GEM still achieved and additional slight improvement when the instance based technique was applied because its memory requirements are larger than BWA-MEM and BOWTIE2 but smaller than SNAP. These techniques can be implemented easily and do not require modifying the source code of the applications neither to have privilege permissions. Any user can add these strategies to its current running jobs. As we have seen, very simple configurations at the time of executing an application can generate significant differences in execution times when running on NUMA systems. This adds an extra layer of complexity to the basic techniques of parallelism and performance evaluations. It is an important factor to be taken into account when improving the overall performance of any application.

Acknowledgements. This research was supported by MICINN-Spain under contract TIN2014-53234-C2-1-R.

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)
2. Fonseca, N.A., Rung, J., Brazma, A., Marioni, J.C.: Tools for mapping high-throughput sequencing data: supplement. Bioinformatics 1–9 (2012)
3. 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)
4. Herzeel, C., Ashby, T.J., Costanza, P., Meuter, W.: Resolving load balancing issues in BWA on NUMA multicore architectures. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013. LNCS, vol. 8385, pp. 227–236. Springer, Heidelberg (2014). doi:10.1007/978-3-642-55195-6_21
5. 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)
6. Klöckner, A.: Lec8-Demo (2012). http://github.com/hpc12/lec8-demo
7. Lameter, C., Hsu, B., Sosnick-Pérez, M.: NUMA (Non-Uniform Memory Access): an overview. ACMQueue 1–12 (2013)
8. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nat. Methods **9**(4), 357–359 (2012)
9. Lenis, J., Senar, M.A.: On the performance of BWA on NUMA architectures. In: 2015 IEEE Trustcom/BigDataSE/ISPA, pp. 236–241 (2015)
10. Li, H.: Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM, p. 3 (2013). arXiv preprint: arXiv:1303.3997
11. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics **25**(14), 1754–1760 (2009)
12. Li, H., Homer, N.: A survey of sequence alignment algorithms for next-generation sequencing. Brief. Bioinform. **11**(5), 473–483 (2010)
13. 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)
14. Misale, C., Ferrero, G., Torquati, M., Aldinucci, M.: Sequence alignment tools: one parallel pattern to rule them all? BioMed Res. Int. **2014**, 1–12 (2014)
15. 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 2014, pp. 4:1–4:10. ACM, New York (2014)
16. 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**, 1–16 (2014)
17. Trapnell, C., Salzberg, S.L.: How to map billions of short reads onto genomes. Nat. Biotechnol. **27**(5), 455–457 (2009)
18. Zaharia, M., Bolosky, W., Curtis, K.: Faster and more accurate sequence alignment with SNAP, pp. 1–10 (2011). arXiv preprint: arXiv:1111.5572v1
19. Zook, J.M., et al.: Extensive sequencing of seven human genomes to characterize benchmark reference materials, p. 26468 (2015). (bioRxiv)