2015 IEEE Trustcom/BigDataSE/ISPA

On the performance of BWA on NUMA architectures

Josefina Lenis and Miquel Angel Senar
Universitat Autònoma de Barcelona (UAB), Bellaterra 08193, Spain
Email: {josefina.lenis, miquelangel.senar}@uab.es

***Abstract*—Rapid progress in genome sequencing techniques is creating the necessity of advanced algorithms to process such information in reasonable time. Alignment applications such as BWA (Burrows Wheeler Aligner) are essential for solving genomic variant calling studies. Although BWA takes advantage of multi-threading execution, it exhibits 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 BWA's scalability. We have analyzed performance problems of BWA on two NUMA systems: one based on Intel Xeon and the other one based on AMD Opteron. We present some simple techniques that can be applied at system level and do not require any application modification. Significant improvements in speedup were achieved when these techniques were applied to the execution of BWA on both systems.**

Keywords*—*NUMA, Memory System Performance, Burrows-Wheeler Aligner, NGS

I. Introduction

The availability of next-generation sequencing (NGS) has transformed biomedical research by increasing throughput capabilities and decreasing the cost of sequencing. This technological evolution is generating massive data sets that require efficient applications and algorithms to deal with the analysis of such data. A fundamental step in sequencing analysis is the alignment (mapping) of the generated reads to a reference sequence. This step, which involves the accurate positioning of reads onto a reference genome sequence, is highly important because it determines the global quality of the downstream analysis. Mappers have to be sensitive and accurate, and as fast as possible and not too computationally demanding [2]. 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 [10], while computational and time requirements have received comparatively less attention. As multicore architectures become widespread and parallel applications become more important, writing parallel programs that exhibit good scalability is far from easy. In particular, new multicore systems are non-uniform memory architectures (NUMA) and achieving good system performance requires that computations are co-located with the data they access. This paper presents performance analysis of a well-known mapper, Burrows-Wheeler Aligner (BWA) [3], on NUMA architectures. BWA is one of the fastest and most popular read sequence aligners available; it uses a multithread scheme, being suitable to exploit all computational resources of a multicore system. From this study, we analyze scalability problems exhibited by BWA and we proposed simple system-level techniques to alleviate them. We obtained results up to ~3-fold speed up over original BWA multithread implementation. These techniques do not require changes in the application's code and, therefore, they could be easily applicable to other applications that exhibit similar memory access patterns to the ones exhibited by BWA.

The paper is structured as follows. Section II describes basic concepts of NUMA systems and provides concrete details of the systems used in our experiments. Section III introduces the problem of sequence alignment and provides an overview BWA, one of the most widespread aligners that has been used in our analysis. In Section IV, we introduce the methodology and all the execution scenarios used to improve the performance of BWA. Section V shows the results obtained in our experiments. Section VI presents related work and last section summarizes the main conclusions of our work.

II. 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 is done using global memory addresses [5]. Each processor has its own memory and can access to memory associated with the other processors in a coherent way but the latency of doing so is greater than accessing its own local memory. A processor and its respective memory is called NUMA node. As an example of NUMA systems, we can see the following two figures (figure 1 and figure 2) that represents two architectures we employed during this study, one manufactured by AMD and the other by Intel.

The first system, shown in figure 1, is a four-socket AMD Opteron Processor 6376, 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) 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 in 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

978-1-4673-7952-6/15 $31.00 © 2015 IEEE
DOI 10.1109/Trustcom.2015.638

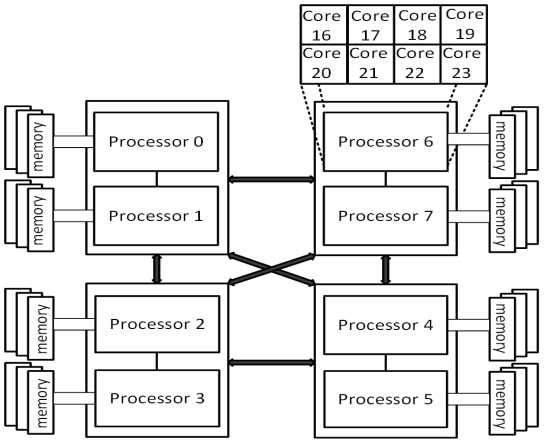


Fig. 1: Schematic diagram of the AMD Opteron 6376 architecture (Abu-Dhabi)

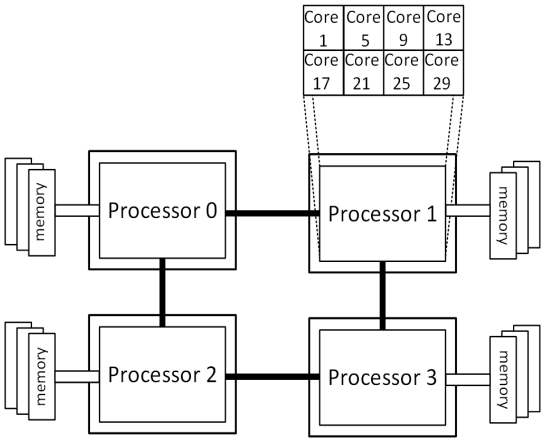


Fig. 2: Schematic diagram of the Intel Xeon E5 4620 architecture (Sandy Bridge)

transfer), the data rate (single or double), and the actual bandwidth between memory controller and the DDR3 memory modules [18].

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 take profit of 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 on the overall execution time due distance penalty in distant memory banks. A second issue that can also increase execution time is contention; applications employing large number of threads and storing all data in a single bank, generate race between threads congesting the connections links and downgrading accesses 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 on a memory bank corresponding to its local CPU. Specifying memory policies for a process does not cause any memory allocation [6]. 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 some 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 [7] called libnuma. This user space shared library can be linked to applications and provides explicit control of allocation policies to user programs. NUMA execution environment for a process can also be set up by using the *numactl* tool [7]. *Numactl* can be used to control process mapping to cpuset and restrict memory allocation to specific nodes without altering the program's source code.

III. Sequence Alignment

Aligning (or mapping) sequencing reads to a reference genome is the first step in many comparative genomic workflows, including variant calling, isoform quantitation and differential gene expression. In many cases, the alignment step is the slowest [8] because aligning a set of reads to the reference genome implies that a right position has to be found within 3 billion possible positions and this is not a computationally trivial task [9]. Mappers can be classified in two main groups: based on hash tables or based Burrow Wheeler Transform (BWT) [19]. In hash table based algorithms, every substring of length k is hashed. For each query P, one can easily retrieve all suffixes of that start with P_1 ... P_K. 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 genome into an FM-index, the look up performance of the algorithm improves for the cases where a single read matches multiple locations in the genome [19]. 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 [11]. The computational time required by a mapper to align a given set of sequences and the computer memory required are critical characteristics, even for mappers based on BWT. If a mapper is extremely fast but the computer hardware available for performing a given analysis does not have enough memory to run it, then the mapper is not very useful. Similarly, a mapper is not useful either if it has low memory requirements but it is very slow. Hence, ideally, a mapper should be able to balance speed and memory usage while reporting the desired mappings [12].

A. 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 analysed. 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 [13]. For this work we centered on BWA original algorithm *aln* [3]. Parallelization is done using a simple scheme that divides independent reads among threads. Therefore, we expected to obtain reductions in runtime proportional to the number of employed threads. And logically we as well expected to obtained a minimum execution time when the number of threads is maximum. However, results collected are not consistent with this hypothesis. In figure 3 we show execution times for BWA multithread performing several experiments increasing the number of threads by 8 up to 64 threads in our experimental platforms.

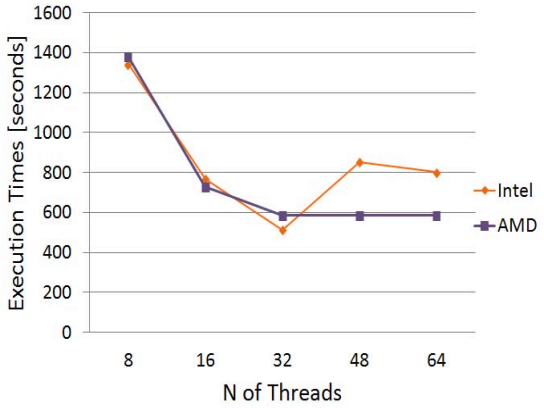


Fig. 3: BWA Scalability

The fact that all threads have to access to the reference genome index, which is located at the memory bank of the main thread produces a bottleneck and increases overall memory access time. 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. Hence, cache reuse is low and remote accesses to the memory bank that contains the index of the reference genome are high. Distribution of work between independent threads does not reduce total execution time due to access to shared data structures. This behavior of BWA is a clear drawback when the number of threads increases employing new cores in remote NUMA nodes. Depending on the architecture, latency of the memory accesses and overall performance drop due to contention can either amplify or not have the effects of this drawback. In AMD, BWA performance exhibited a limited scalability beyond 16 threads. Execution with more than 16 threads implies that more than one socket is being used so some threads are being allocated in remote sockets. The increment of distributed work is counteract by the remotes accesses latency. On Intel system on the other hand the behavior is different. NUMA issues are not as relevant as on AMD [4]. BWA reduces its execution time as the numbers of cores increases up to 32 cores. 32 cores is the number of ”physical” cores present on the Intel system. Beyond that point, hyperthreading is needed to rise the number of threads that can be executed simultaneously.

IV. Experiments and Methodology

In the present work, our goal focuses on the exploration of solutions that could be applied to improve application performance without changing the source code. Therefore, we investigated mechanisms that can be easily applied by using system-level tools. A first set of experiments consisted on the execution of a single instance of BWA varying data and thread allocation. We tested configurations that enforced locality between threads and memory banks as well as configurations where shared data structures are spread evenly on different memory banks. A second set of experiments was based on the idea of replication: multiple independent instances of the same application were executed simultaneously, so the main shared data was replicated on each memory bank to keep memory accesses locally and reduce remote accesses. In all our experiments we used the *numactl* Linux tool, mentioned in section II, to map threads to cores and allocate data with parameters *–membind* and *–physcpubind*, respectively. Details of these two schemes are presented below.

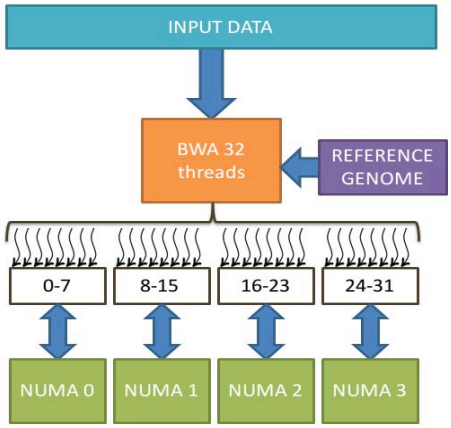


Fig. 4: Scheme A: Single instance multithreading

Scheme A: Single instance with multiple threads. The first scheme consists on executing a single instance of BWA multithread. We focused on 5 particular cases: using 8, 16, 32, 48 and 64 threads because, as explained in section II, each processor has 8 cores and 1 memory bank associated; so 8, 16, 32, 48 and 64 threads will imply a minimum usage of 1, 2, 4, 6 and 8 NUMA nodes respectively on AMD architecture and 4 NUMA nodes on Intel architecture. Figure 4 illustrates an example of one instance of BWA running with 32 threads. The input data is composed of two files. One that has all the reads that need to be mapped and the second one with the reference genome index. Independently of the number of threads, all of them will map their own reads against the same genome.

This scheme was tested with two different configurations:

- **BWA + local:** In this case, BWA was executed in such a way that local affinity was maximized. This means that threads were assigned to a reduced number of cores and data was allocated within NUMA nodes as near as possible to the cores where the program was being executed. For instance, when BWA run with 8

threads, a suitable mask was applied to *numactl* so that only cores from 0 to 7 would be used and data was allocated to the nearest memory bank corresponding to NUMA node 0; when BWA was executed with 16 threads, cores 0-15 and NUMA nodes 0 and 1 were used, and so on.

- **BWA + interleave:** This configuration consisted on the use of the interleave policy for memory allocation instead of local. For instance, in an execution of 16 threads, all threads were assigned to cores from 0 to 15 and data was allocated in a round robin fashion between nodes 0 and 1, employing the parameter –*interleave* of *numactl*.

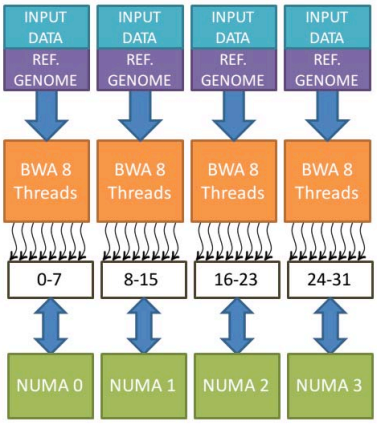


Fig. 5: Scheme B: Multiple instances multithreading

Scheme B: Multiple instances with multiple threads. This scheme considered each NUMA node as a symmetric multi-processor unit, capable of running a independent instance of BWA. Each one of these instances being a multithreaded application. Independent instances of BWA were created, each one running in a single NUMA node (all independent instances were running with 8 threads). The input file with all the reads was manually divided into the number of instances. Figure 5, illustrates this configuration with 4 independent instances that are executed simultaneously. Input data is 1/4 the size of the original and the reference genome is replicated 4 times. The total number of threads will be *instances x 8* (32 threads). The scenarios tested were for 2, 4, 6 and 8 independent instances.

The reference human genome used is GRCh37, maintained by The Genome Reference Consortium and the data sets used as input data is a simulated Illumina set created with seqAN [1] (single end, base length = 100, number of reads= 10M).

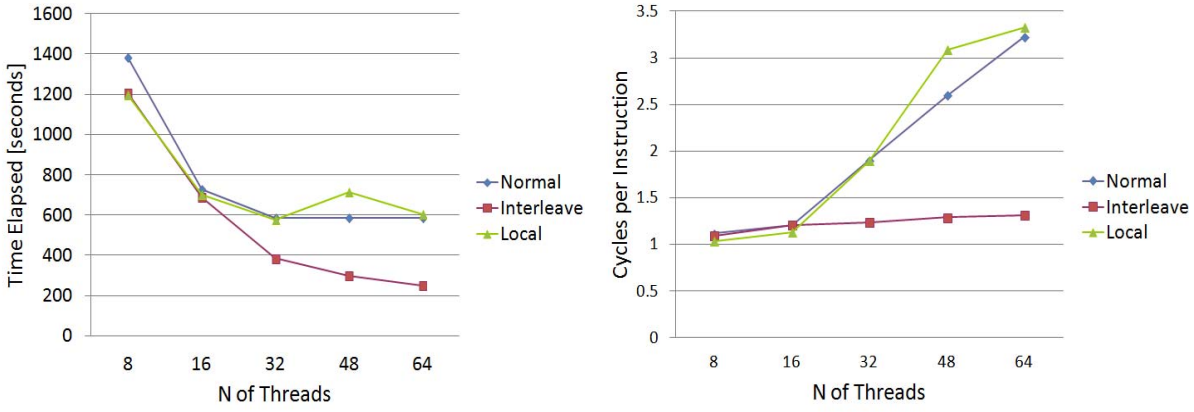
V. Results

Figures 6 and 7 show results obtained by normal BWA compared to the two configurations corresponding to scheme A. Average of five runs are plotted. By normal BWA we refer to the execution of BWA without any explicit control on NUMA nodes. Letting the operating system to apply default allocation policies (we observed that Linux allocates threads in a dispersed way across all NUMA nodes). ”BWA+Local” is intended to increase program locality. This is achieved by using *numactl* and enabling specific cores and memory banks. For instance, if BWA was executed with 8 threads, we used the parameter *physcpubind*=0-7 (AMD) to enable cores of the first NUMA node and *membind*=0 to enable its corresponding memory bank. When BWA was executed in this way, all threads and all data were forced to be allocated in one NUMA node. However, as the number of cores and the number of enabled memory banks increases, the locality guarantee is slightly reduced. In particular, when 64 threads are used, BWA behaves the same with or without *numactl* because all memory banks and all cores are enabled and allocated according to the operating system policies.

Figure 6a shows execution times on AMD system. As seen, normal BWA and ”BWA+Local” exhibit a similar behavior. For the case of 8 threads, increasing locality - as ”BWA+Local” does- reduces the latency over normal execution. Similar limitations in scalability are observed when more than 16 threads are used. According to the characteristics of our system, memory accesses of 2 hops will always occur when more than 16 threads are used. Both, BWA and ”BWA+Local” suffer contention problems due to accesses to the reference genome. A difference is observed between normal BWA and ”BWA+Local” in the case of 32 threads. This could be explained because ”BWA+Local” always has a set of 8 cores located at 2 hops of the reference genome. This boost the cycles per instructions, affecting the execution time as can be seen in figure 6b. In contrast, BWA seems to benefit from the default Linux policy in which threads might be located closer to the reference genome. When BWA is running employing interleave as allocation policy the reference genome is spread -in a round robin fashion- through all memory banks. Bottleneck problems are mitigated and contention latencies due to reference genome accesses average out. By using interleave, BWA's scalability still increases beyond 16 threads and provides a significantly improvement of ~2.4x in execution time for 64 threads compared to normal BWA.

On Intel architectures, executions turned into different results. Although both architectures have 4 sockets, AMD presents 8 NUMA nodes and a more complex interconnection set with higher latency penalty [4]. On Intel, asymmetry is not as noticeable as on AMD. As can be seen in figure 7, no strategy emerges clearly as the best. Applications that are not NUMA-aware running on Intel architectures do not experience an increase in time penalties in a similar to that experienced on AMD architectures. Nevertheless, they don't benefit from NUMA actions (i.e. interleaving).

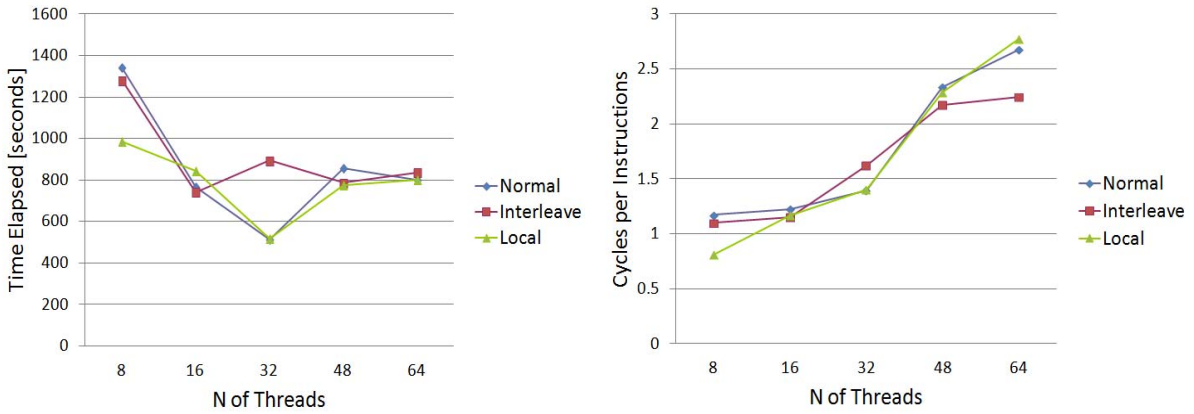
Figures 8a and 8b show a complete comparison between all the strategies presented in this article. Speed up was calculated against the wall time of normal BWA. Best performance is achieved by the configuration of multiple independent instances of BWA on both architectures. As seen, some cases performed slightly worse than normal BWA (especially on Intel). On both architectures, however, the use of multiple independent instances arises as the best solution. Replication of the genome reference reduces simultaneous accesses to a remote bank and application also benefits from multithreading parallelization. Execution times decreased because input data was divided into smaller chunks -the size of the chunk is *1/instances* - each thread has less queries to map and the reference genome is locally stored. As multiple instances of BWA use a particular copy of the reference genome index, memory requirements increase. However, modern systems are



(a) Runtimes comparison (AMD)

(b) CPI comparison (AMD)

Fig. 6: Multithread executions in AMD Opteron



(a) Runtimes comparison (Intel)

(b) CPI comparison (Intel)

Fig. 7: Multithread executions in Intel

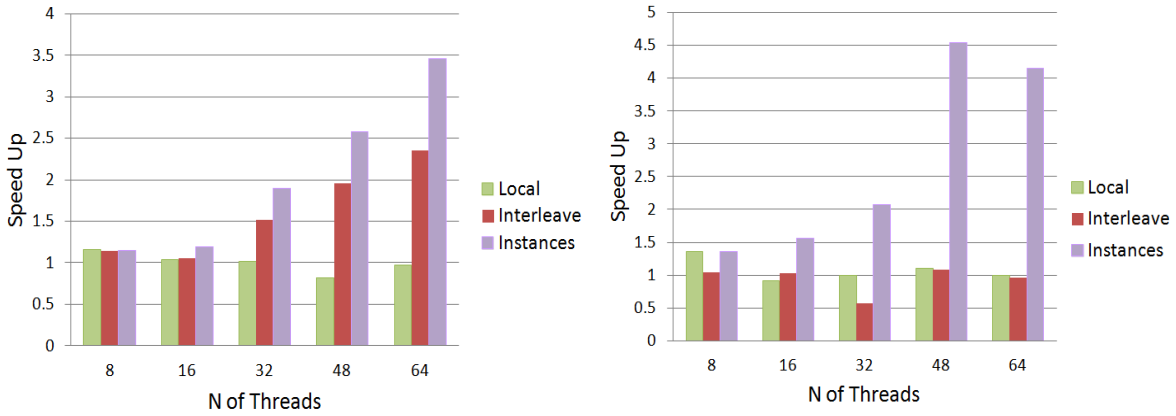
usually equipped with large memories and NUMA banks have plenty of space to allocate such copies.

VI. Related Work

Multicore NUMA systems have become the backbone of parallel processing and research about NUMA-aware applications have been published in the last years. In Yinan *et al.* [15], the authors compare shuffling algorithms against a NUMA-aware algorithm achieving up to 3x speedup. Another example is the analysis carried out by Frasca *et al.* [16] where novel dynamic task distribution NUMA-aware techniques are introduced. Genome alignment problems have been considered by Misale [17]. The author created a modified version of Botwie2 [8] known aligner improving local affinity of the original algorithm. Herzeel*et al.* [13] 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 in general that also requires that source code is available. In contrast to these cases, improvements presented in this article are obtained without altering the original code of BWA and they are independent of existing code versions.

VII. Conclusions

In this paper, different scenarios were studied to see the influence of NUMA architectures (Intel and AMD) when executing an application of genomic alignment as BWA. Although this is an application capable of running on a multicore architectures, its memory access pattern exhibits limited scalability when threads are distributed beyond a certain NUMA distance. The two problems present in NUMA architectures (latency and contention) can be mitigated using multiple independent instances. This solution does not require



(a) Speed Up (AMD)

(b) Speed Up (Intel)

Fig. 8: Comparision of all strategies

any changes to the application and can be applied both to AMD and Intel architectures with no privilege permissions. Some extra memory consumption is generated to allocate all data structures that are replicated on NUMA nodes. But this requirement is easily achieved in modern systems and BWA (and other similar tools based on BWT) can benefit from it. Execution time were reduced by a factor of ~3.5x for AMD and ~4x for Intel architecture when this configuration was used.

Besides this solution, contention problems can still be reduced in AMD architectures by using an interleave policy that distributes the reference genome index accross all memory banks in the system. This simple technique also achieves a significant improvement in run time: up to 2.4x the performance of BWA running with system default policies. We have used BWA in our experiments, but we expect that similar results could be achieved with other mapping tools that are based on BWT and use a reference genome index that is shared by multiple threads.

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. And it is an important factor to be taken into account when improving the overall performance of any application. This fact should also be taken into account when conducting studies that compare different tools because each one might require a different NUMA setup in order to achieve its optimal performance.

Acknowledgements

This research was supported by MICINN-Spain under contract TIN2011-28689-C02-01.

References

[1] M. Holtgrewe, "Mason – a read simulator for second generation sequencing data". Technical Report TR-B-10-06, Institut für Mathematik und Informatik, Freie Universität Berlin.,2010, https://www.seqan.de.

[2] S. Caboche, C. Audebert, Y. Lemoine, and D. Hot, "Comparison of mapping algorithms used in high-throughput sequencing: application to Ion Torrent data," *BMC Genomics*, vol. 15, p. 264, 2014.

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

[4] Lars Bergstrom, "Measuring NUMA effects with the STREAM benchmark", CoRR, vol. abs/1103.3225, 2011.

[5] P. García-Risueño and P. E. Ibáñez, "A review of High Performance Computing foundations for scientists," arXiv:1205.5177v1 [physics.comp-ph], pp. 1–33, 2012.

[6] C. Lameter, B. Hsu, and M. Sosnick-Pérez, "NUMA (Non-Uniform Memory Access): An Overview," 2013.

[7] A. Kleen, "An NUMA API for Linux," 2004.

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

[9] M. P. Dolled-Filhart, M. Lee, C. Ou-Yang, R. R. Haraksingh, and J. C.-H. Lin, "Computational and bioinformatics frameworks for next-generation whole exome and genome sequencing.," ScientificWorldJournal., vol. 2013, pp. 1–10, Jan. 2013.

[10] A. Hatem, D. Bozdag, and Ü. V. Çatalyürek, "Benchmarking short sequence mapping tools," Proc. - 2011 IEEE BIBM 2011, pp. 109–113, 2011.

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

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

[13] C. Herzeel, T. J. Ashby, P. Costanza, and W. De Meuter, "Resolving Load Balancing Issues in BWA on NUMA Multicore Architectures," vol. 8385, pp. 227–236, 2014.

[14] M. Holtgrewe, "Mason – A Read Simulator for Second Generation Sequencing Data," Life Sci., no. October, p. 18, 2010.

[15] Y. Li, I. Pandis, R. Müller, V. Raman, and G. Lohman, "NUMA-aware algorithms: the case of data shuffling.," in CIDR, 2013.

[16] M. Frasca, K. Madduri, and P. Raghavan, "NUMA-aware graph mining techniques for performance and energy efficiency," 2012 Int. Conf. High Perform. Comput. Networking, Storage Anal., pp. 1–11, Nov. 2012.

[17] C. Misale, "Accelerating Bowtie2 with a lock-less concurrency approach and memory affinity," 2014 22nd PDP, pp. 578–585, Feb. 2014.

[18] R.Braithwaite, P. McCormick, and F.Wu-chun Feng. "Empirical memory-access cost models in multicore NUMA architectures.", ICCP Virginia Tech Department of Computer Science 2011.

[19] H. Li and N. Homer. "A Survey of Sequence Alignment Algorithms for next-Generation Sequencing." Bioinformatics, vol. 11, pp. 473–483 May.2010