Parallel Computing 87 (2019) 11–24

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Exploring efficient data parallelism for genome read mapping on multicore and manycore architectures☆

Shaolong Chen*, Miquel Angel Senar

Universidad Autonoma de Barcelona, Computer Architecture and Operating System, Barcelona, Spain

ARTICLE INFO

Article history:
Received 1 September 2018
Revised 28 March 2019
Accepted 28 April 2019
Available online 15 May 2019

Keywords:
Sequence alignment
Heterogeneous architecture
Intel xeon
Intel xeon phi
NUMA Node

ABSTRACT

Nowadays heterogeneous architectures formed by multicore and manycore systems have become attractive solutions to cope with the data booming in genomic-based studies. Our work explores the efficient usage of heterogeneous architectures in such area. In particular, we have studied the use of manycore components like the Xeon Phi accelerator, which has proved to be a convenient choice because it allows an easy migration of applications developed for multicore servers based on the $\times 86$ architecture. Our study also focuses on the problem of sequence alignment, which is one of the fundamental and most costly computational stages in most genome variant studies. We concentrate our attention on BWA, one of the most popular sequence aligners, and we have focused our attention on three types of heterogeneous systems, one containing Intel multi-core CPUs and accelerators, one that are made up of several multi-core servers, and one large-scale system. Each with different characteristics in terms of number of CPUs, number of cores and system organization memory. Although the problem of alignment of sequences fits in the embarrassingly parallel pattern, achieving good performance and good scalability in heterogeneous environments can be complex. We have analyzed different strategies based on the distribution of data and the replication of certain data structures and we found that MDPR (Multi-level Data Parallelization and Replication) strategy has shown the best results in all the heterogeneous platforms tested. Its results have surpassed other strategies proposed in the literature and have shown its malleability to be used in different heterogeneous environments without the need to apply specific adjustments according to the underlying architecture. In the design of MDPR, different static and dynamic data distribution strategies have also been evaluated. The best results were obtained by the static strategy, which has a significant preprocessing cost. However, the dynamic strategy of data distribution using a round-robin mechanism obtained similar times without the need for the preprocessing stage. Although our proposal was applied to BWA using human genome data samples, this strategy can be easily applied to other sequence datasets and alignment tools that have similar operating principles with those of BWA aligner.

1. Introduction

In recent years, heterogeneous systems consisting of different architectures have become a significant trend in the high performance computing arena [1]. Manycore architectures, such as GPUs, have obtained considerable popularity, despite the fact that the specific programming languages such as OpenCL and CUDA require significant coding skills [2]. Intel Xeon Phi [3] is another manycore coprocessor, which has became an attractive solution for existing programs on multicore systems that can be easily ported because Xeon Phi is also based on the $\times 86$ architecture.

Compared to computing evolution (doubling the number of transistors every 18 months), the amount of sequence genome data only doubles every 12 months [4]. However, this is an spectacular growth of data that constitutes an important challenge that still requires efficient applications to process them.

Sequence alignment, variant calling and variant annotation comprises three fundamental operations in genome data studies [5]. Sequence alignment is a crucial step that can provide primary consequences for the other two operations. It involves mapping short reads to a genome sequence reference as accurately as

☆ **Conflict of interest.** None.

* Corresponding author.
E-mail addresses: shaolong.chen@uab.cat (S. Chen), miquelangel.senar@uab.cat (M.A. Senar).

https://doi.org/10.1016/j.parco.2019.04.014

possible [6]. Although many aligners have been developed in recent years, such as MAQ, SOAP and BLAST, they all exhibit a significant execution time and a large memory footprint. Although, its execution time is similar to other existing aligners, BWA (Burrows-Wheeler Aligner) [7] takes advantage of the BWT indexing technique that decreases memory requirements of the alignment process. This reduction in memory needs combined with its mapping accuracy have made BWA one of the most popular aligners in the scientific community.

In this paper, we explore different strategies that allow BWA to run on heterogeneous systems that combine multicore CPUs with manycore accelerators or different multicore systems. We analyze some proposed strategies in the literature and propose enhancements and new strategies to improve global performance. This paper extends our previous studies in which we proposed some data parallelization/replication strategies and made an evaluation of BWA-based aligners in manycore-based systems. In those works our study focused only on heterogeneous systems that incorporate accelerators. In one case, we analyzed improvements when the applications were executed in native mode [8], and in [9] we studied different variants to execute BWA in symmetric mode. The MDPR strategy analyzed in the current paper derives from these previous works and it has been generalized and extended so that it can be executed indistinctly not only in accelerator-based servers but in other types of heterogeneous environments formed by multiple CPUs organized around different NUMA architectures. The strategy MDPR utilizes multiple instances in order to reduce memory bottlenecks by decreasing memory congestion and improving memory locality. We present a study where we analyze the performance of different BWA variants on multicore and manycore system. We have augmented also MDPR with different mechanisms of dynamic data distribution and we have evaluated them with two representative datasets and three heterogeneous architectures. The main contributions of this paper can be summarized as follows:

- We propose a strategy (named MDPR) that has been applied to BWA by adding new mechanisms of data parallelism and data replication. The strategy exhibits a hierarchical structure based on different instances of computation that include replication of certain data structures and the distribution of input data among all parallel instances.
- We analyze three data distribution strategies for MDPR, one static and two dynamic (even and round-robin).
- We have carried out an experimental study comparing different versions of BWA on multicore and manycore systems, including the strategies proposed in this work and other alternatives proposed in the literature. This study has shown us the best alternatives to run BWA on manycore systems based on Intel Xeon Phi and on heterogeneous systems based on Intel and AMD multicore processors.

The rest of the paper is organized as follows. Section 2 describes related work in performance improvement of sequence alignment algorithms, including BWA, on parallel architectures. Some general background in multicore and manycore architectures is presented in Section 3 and BWA aligner is described in Section 4. Section 5 discusses DP, DR, DPR and MDPR strategies. Section 6 describes our experimental environment and Section 7 shows the results we obtained in the experiments. Finally, Section 8 discusses the main conclusions of this research and highlights some future work.

2. Related work

Our work explores effective data parallelism solutions for the BWA aligner that are suitable for heterogeneous architectures. Many sequence aligners have been studied on multicore system, such as Blast [10], Bowtie [11], SOAP2 [12] and BWA [13], but few works have been applied to Intel Xeon Phi manycore systems or other heterogeneous systems.

Several researchers have studied the problem of read alignment in multicore architectures with shared memory. Zhang et al. [14] illustrated that sequence alignment is memory-sensitive and tried to optimize the usage of BWT index by decreasing their irregular memory accesses. Kathiresan et al. [15] presented several parallelization strategies, including memory cache improvements, that could be useful on multicore system. Macedo et al. [16] introduced a hybrid programming schema using OpenMP and MPI to facilitate multiple sequence alignment application on heterogeneous multicore clusters, but they do not explore their approach to sequence alignment in manycore architectures. Olivier et al. [17] compared different task scheduling algorithms on multicore systems to evaluate their impact in overall performance. Lenis et al. [18] investigated memory interleaving and data parallelization strategies for NUMA multicore systems. Chen et al. [19] studied an offloading acceleration scheme applied to BWA aligner in FPGA architectures. Houtgast et al. [20,21] made an effort to adopt alignment applications on GPU- and FPGA-based architectures. Several proposals of alignment performance improvement on FPGA are discussed in [22]. Additionally, many papers [23] studied sequence alignment improvements on manycore architecture based on GPUs.

Particular proposals applied to BWA have also been produced in recent years. Many of them describe different approaches to improve the performance of BWA aligner on some particular architecture. Zhang et al. [14] found aligner BWA is a memory-sensitive application by optimizing cache mechanisms on multicore architectures. Houtgast et al. [24] accelerated BWA by offloading some highly parallel kernels on a GPU system. Houtgast et al. [25] utilized a simple adaptive load balance algorithm on GPUs and achieved a considerable performance improvement. Other existing approaches are using Hadoop technology, such as BigBWA [26] that achieved a significant improvement of performance on multiple nodes. Or SparkBWA [27] that tried to use another big data technology, Spark, to boost BWA aligner. MICA [28] explored the extra parallelism available in each Xeon-Phi core. Cui et al. [6] proposed mBWA which works on Xeon-Phi systems using the offload mode. mBWA utilizes a multi-level parallelization strategy that includes data IO parallelization and a parallel pipeline in reads alignment. Tian et al. [29] presented several practical SIMD vectorization techniques on Xeon-Phi architectures. However, such SIMD techniques require complex code rewriting in order to incorporate them with the existing genome aligners. Memeti et al. [30] and Wang et al. [31] provided solutions for large scale DNA analysis which exploits thread affinity level and SIMD techniques in the Xeon Phi, but their solution does not deal with data parallelism among threads or instances. pBWA [32] is an efficient parallel version of BWA based on the Open MPI library. It not only preserves the multi-thread capability provided by BWA but also adds efficient parallelization for core alignment functions on the heterogeneous cluster. Unfortunately, it uses Pthread library and cannot be evaluated for a large number of instances of pBWA on the Xeon Phi architecture due to its memory limitations. Herzeel et al. [33] evaluated an Intel Cilk implementation of BWA on heterogeneous systems that corrected data load imbalance produced by differences in data reads. We compared this BWA Cilk version with the original BWA Pthread version and an BWA OpenMP-based version proposed in our previous work [9].

3. Multicore-manycore architectures

Nowadays, systems based on NUMA (Non Uniform Memory Access) architectures are the most used in the field of high performance computing. These systems are often used in conjunction

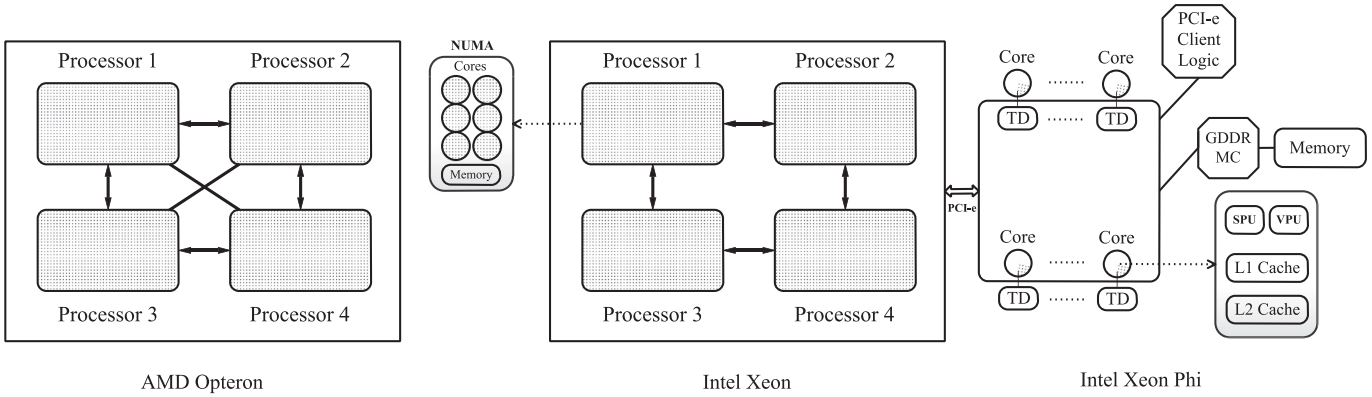

Fig. 1. Heterogeneous multicore-manycore architecture.

with manycore accelerators that increase the number of cores available in the system at a reasonable cost. In this section we provide a brief description of the basic architectures that we have used in our study, namely, multicore systems with NUMA nodes and manycore accelerators.

Fig. 1 shows the main architectures of the systems used in this work. Two multicore systems are depicted at the left side and center of the figure (an AMD-Opteron based server and an Intel Xeon-based server). Intel Xeon Phi is at the right side. Our AMD Opteron and Intel Xeon systems have a NUMA node with different link levels across processors.

AMD-Opteron and Intel Xeon systems have both a NUMA architecture with four independent sockets, each socket has a one or two multicore processors and each processor is attached to a local memory bank. Each processor and its corresponding memory bank is referred as a NUMA node. One of the most significant characteristics of NUMA systems is that accesses from threads running in local cores (located in the same NUMA node) have lower latency than accesses that go to physical memory located in different NUMA nodes (as shown in Fig. 2). This asymmetry in memory accesses involves additional complexity when executing parallel applications because access to remote banks increases the execution time and, in the case of genome alignment, introduces also congestion problems.

Intel Xeon Phi accelerator (seen at right side of Fig. 1) has a bi-ring architecture with 57 to 61 cores inside. Each core in the Xeon Phi has a SPU, a VPU with 512-bit SIMD capability, and a private 512KB L2 cache that is maintained fully coherent by a global-distributed tag directory (TD) [34], which forms a unified shared L2 cache of 30.5MB. Xeon Phi can achieve 1 TFlops in double precision or 2 TFlops in single precision. In addition, memory bandwidth can reach 352GB per second theoretically thanks to 16 memory channels. The memory controllers (GDDR MC) and the PCIe Client Logic provide a direct interface to the GDDR5 memory and the PCI express bus, respectively.

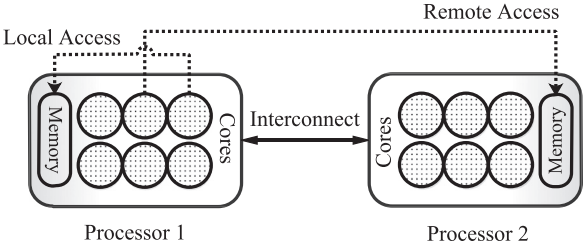

Fig. 2. NUMA memory architecture.

The Intel Xeon Phi has an $\times 86$ architecture that allows three execution modes. An application can be executed in three different ways when submitted to a Xeon Phi system [35]:

- Native mode: the application is executed on each component (processor or accelerator) independently.
- Symmetric mode: the accelerator is seen as a regular node in the system and the application is executed on all components (main processor and an accelerator).
- Offload mode: the application is executed on the main processor but selected highly parallel sections pass to the accelerator.

4. Genome aligner: BWA

Genome read aligners provide the relative position of short reads within a reference genome. Despite the particular differences of each aligner, they all share a similar mode of operation that can be summarized as follows:

1. There is a set of reads that can be mapped to the reference genome independently.
2. There is a reference genome data structure (genome index) that is read-only data and is used to map each individual read.
3. The results consist in populating a shared data structure that would be written on an output file at the end of the alignment.

Read aligners need to construct an index for short reads or for genome reference or for both. Depending on the index construction, read aligners can be classified into three groups [23]: hash table-based, FM-index based and merge sorting based. The former two groups contain the majority of aligners used by the scientific community. BLAST, MAQ, SOAP and ZOOM are examples of hash table-based aligners. Their main drawback is the large memory consumption required for the index. In contrast, the memory requirements of aligners based on the FM-index mechanism are significantly lower, which explains their increasing popularity.

We focus our study on BWA, written by Li [7], which is one of the most popular sequence alignment tools from the FM-index based family. BWA consists of three algorithms, BWA-backtrack, BWA-SW and BWA-MEM. Our study focuses on BWA-backtrack that has already been used by other works in the literature, such as mBWA [6], pBWA [32] and BMIC [36]. Thus, we will try to provide a view as complete as possible of the performance that each of the different strategies can obtain when applied to the same basic application.

BWA creates the index of genome reference beforehand. Later, short read data is processed using fixed sized chunks, in a round-

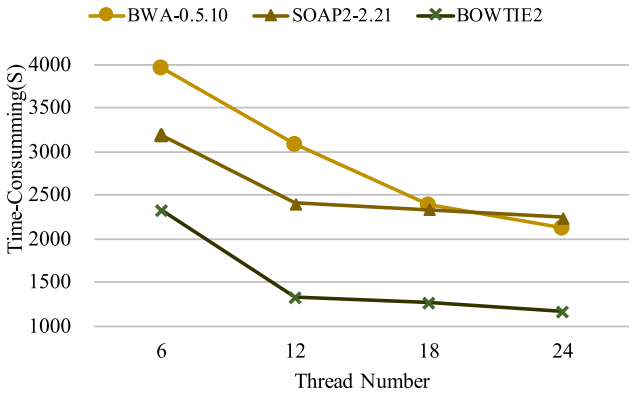

Fig. 3. Scalability on 2-socket 2-NUMA 24-thread system.

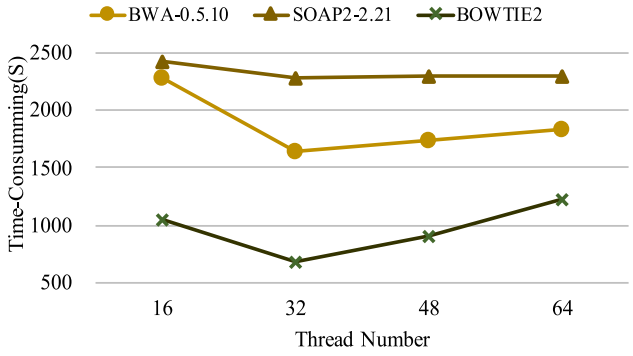

Fig. 4. Scalability on 4-socket 4-NUMA 64-thread system.

robin pattern among threads. Each thread in BWA processes one chunk of data, 256 K by default, that is independently mapped to the genome reference. Like many other aligners, BWA has multi-threading capability (using the Pthread library) to solve read mapping operations in parallel since there is no data dependency in this action. The addition of multi-threading to an aligner does not guarantee that processing resources are used efficiently. In fact, it is quite common to observe that efficiency decreases when the thread count grows large enough [8].

This phenomenon is shown in Figs. 3 and 4. These figures illustrate the results obtained by three sequence aligners (BWA, SOAP2 and BOWTIE2) that are all FM-index-based application when mapping two large short read genome examples of 17.6GB against human genome. Each figure illustrates the results obtained when aligners are executed on two different servers. Both servers are based on NUMA architecture but they differ in the total number of processors, cores and NUMA nodes available. The first system (Fig. 3) has 2-sockets with 2-NUMA Intel Xeon CPU with 12 cores each). Twenty-four threads can be executed simultaneously using Hyper Thread technology. The second system (Fig. 4) is a 4-socket 4-NUMA Intel Xeon CPU. Each socket contains 8-core processor, so the total number of cores is 32 and 64 threads can be executed simultaneously. In both cases, each socket is connected to a memory bank, which constitutes a NUMA node. NUMA nodes are connected by QuickPath Interconnect link.

The performance degradation experimented by all three strategies can be observed in Fig. 3 as the parallelization degree increases. Execution times are reduced until half of the available cores in each architecture are used. From that point, the execution times hardly improve or even get worse slightly. This abnormal phenomena in performance appears when memory becomes the application bottleneck and it could be explained as follows, 1) the increasing number of accesses to remote memory bancks in the system and, 2) the memory congestion exhibited by the bank where the reference genome is allocated.

Moreover, the library and the organization used to handle application threads can cause certain variations in the execution times obtained. In our paper [9], it was observed that the parallelization using Pthread, OpenMP and Cilk presented noticeable differences in performance, with OpenMP and Cilk being the best alternatives to Pthread, which was the library originally used in BWA. Besides the thread library, the main performance problems in genome read aligners arise from the use of memory in NUMA systems. Memory allocation is done according to a first-touch policy and this means that the genome reference index is allocated in a single bank, that will become a bottleneck when multiple threads try to access it [18].

5. Strategies based on data parallelization and replication

The scalability problems of BWA are mainly due to the contention generated by the need for multiple threads to access the same memory bank where the reference genome is located. To mitigate this problem we propose a series of alternatives based on a basic common scheme. In this scheme, there is a first level of parallelism based on groups of threads that handle groups of reads. On the other hand, the reference genome will be replicated in different instances, each of which will be accessible by different thread groups. The number of groups of threads and the number of replicas of the reference genome can vary and can be organized in different ways. The final objective is to analyze which variant obtains the best results in multicore or manycore architectures. In the manycore systems we are interested in finding strategies that are efficient using all the resources of the system (that is, executing in symmetric mode). And we are also interested in solutions for manycore systems that can be easily transferred to multicore systems composed of servers with different characteristics in terms of performance. The variants that have been implemented and tested are the ones described below.

5.1. Data management on homogeneous architectures: DP, DR and DPR strategies

We are exploring data management that involves both parallelism and replication. As shown in Fig. 5, two datasets are involved in the alignment process. N corresponds to the set of all short reads, and n is the number of read subsets in which N is divided. M represents the reference genome, and m is the total number of replicas made of such genome. Therefore, read subsets are referred as $N1, .., Ni, \ldots, Nn$ and reference replicas are referred as $M1, ..Mj, \ldots Mm$ according to Fig. 5.

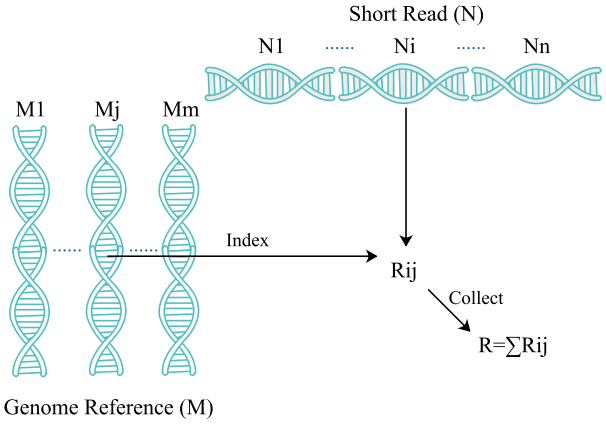

Fig. 5. Sequence alignment procedure.

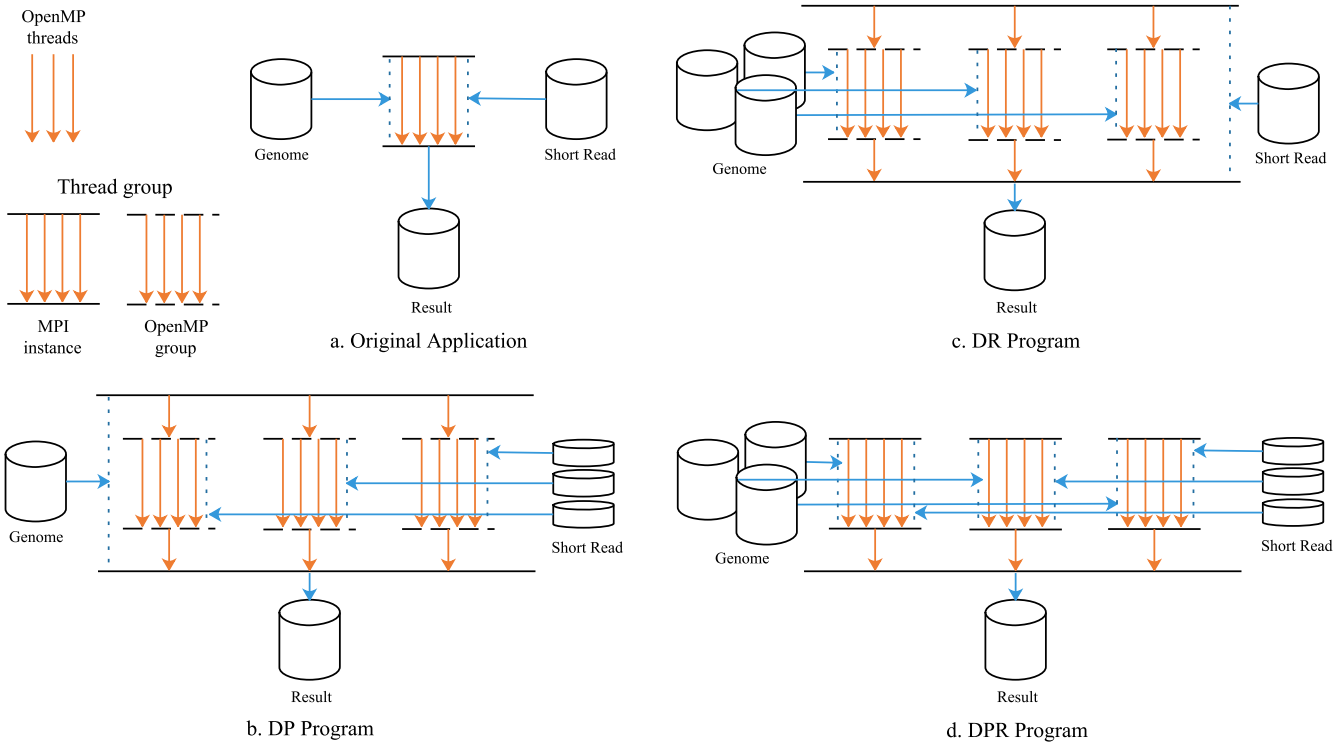

Fig. 6. Implementation of DP, DR and DPR.

Although the reference genome could also be divided, and thus increasing the degree of parallelism available in the application and reducing memory consumption, this approach has been discarded in our study for three main reasons. First, the index of each part of the reference genome would have to be reconstructed at run-time, increasing the total execution time (in the traditional approach, the index is built only once offline and that index is used in all the executions of the aligner). Second, the final results of each subset (or blocks) have to be recombined to be assigned the correct position within the whole reference genome. Finally, a certain loss of precision in the final results is experienced due to reads that might map in the cuts between successive blocks of the reference genome.

In our approach, BWA is organized as a set of thread groups (Ng) that are responsible for mapping some reads. In practice, a thread group represents a group of threads in OpenMP or a process in MPI, as shown in Fig. 6. Each thread group is executed using a different number of threads depending on the hardware cores available in the system. One thread group in the alignment is logically organized as a collection of threads that work on independent datasets of short reads and share the genome index among all the threads. For a total number of potential threads Nt and a number of thread groups Ng, each thread group consists of Nt/Ng threads. For example, Nt is 24 and 240 in the cases of Intel Xeon and Xeon Phi, respectively. When applying two thread groups ($Ng = 2$), 12(24/2) and 120(240/2) threads are included in each thread group on the Intel Xeon and on the Xeon Phi, respectively.

The process of alignment has three main phases:

1. Loading the index for the genome reference Mj; the time of this phase is $T(Mj) = T(M)$;
2. Mapping short read subset Ni to genome reference Mj, which achieves mapping sub-result $Rij = Map(Ni, Mj)$; the time of this step is $T(Rij)$;
3. Collecting all alignment sub-results and merging them into final output $R = \sum Rij$; the time of this step time is $T(R)$;

Total execution time is $T = T(M) + T(R) + \sum_{i=1,j=1}^{i=n,j=m} T(Rij)$. Actually, the mapping consumes most of the execution time because loading the index and collecting subresults mainly involve linear operations of reading and writing files. Hence, we can simplify the expression to $T = \sum_{i=1,j=1}^{i=n,j=m} T(Rij)$, being this stage the one that can benefit from parallelization.

There are different variations that can be applied to the values for n and m.

1. Original execution: $n = 1, m = 1$.
   This case corresponds to the original structure of BWA. As shown in Fig. 6a, multiple OpenMP threads are processing all short reads using a single reference genome.
2. DP strategy: $n > 1, m = 1$.
   In this case (see Fig. 6b), short reads are evenly divided into to n parts based on the number of thread groups that are used (Ng), and a single copy of the reference genome is used by all thread groups. We refer to this strategy as data parallelization (DP). Alignment process runs in every thread group with the shared genome index and its relative subset of short read. Subset results are finally gathered and merged from all thread groups.
3. DR strategy: $n = 1, m > 1$.
   In this case, the reference genome is replicated m times, while short reads are shared among all thread groups. We refer to this strategy as data replication (DR). As shown in Fig. 6c, the reference genome is replicated for every thread group in memory. The alignment process in each thread group accesses to the shared set of short reads. Finally, the results are merged as in the previous case.
4. DPR strategy: $n > 1, m > 1$.
   In this case, both short reads and reference genome have several parts or copies (the set of reads is divided in n parts

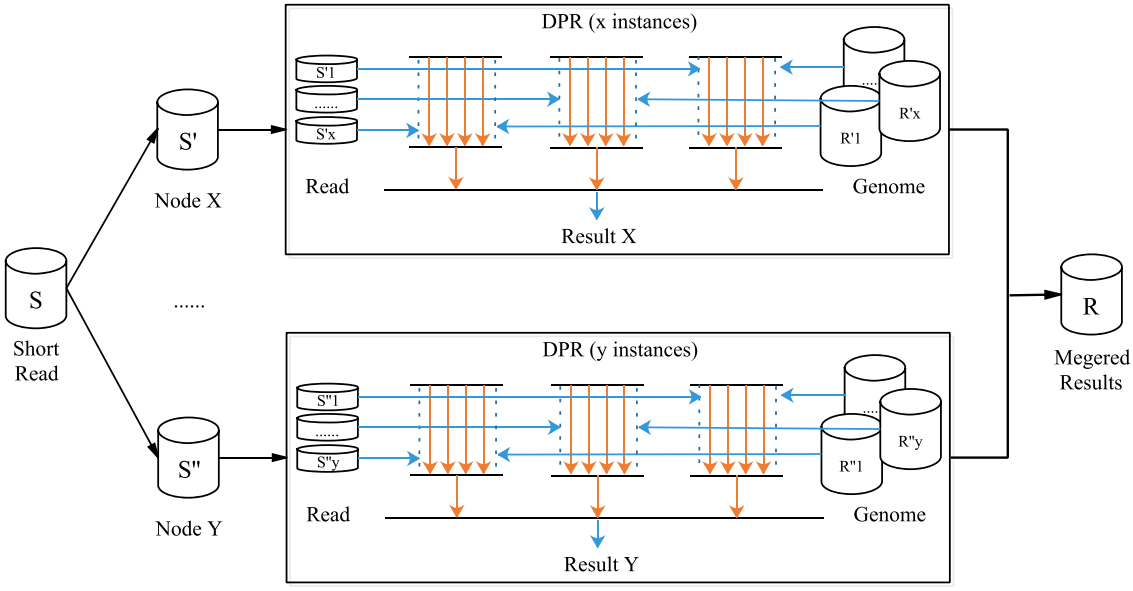

Fig. 7. MDPR strategy.

and reference genome is replicated m times, being n and m the same value). As shown in Fig. 6d, each group of threads has one reference genome replica and short read dataset is evenly divided into as many parts as thread groups. This strategy is referred as DPR as it combines the previous two strategies.

5.2. Data management on heterogeneous architecture: MDPR strategy

The strategies described above can be applied in homogeneous multicore systems with shared memory. In the case of systems based on accelerators (executing in symmetric mode) or in the case of independent multicore systems, we propose a new strategy that generalizes DPR. It is referred as MDPR (Multi-level Data Parallelization and Replication) and is illustrated in Fig. 7. The design of MDPR involves several challenges related to the use of an environment based on heterogeneous and distributed computing resources.

On the one hand, a work distribution mechanism must be established using message passing primitives that do not introduce excessive overhead in terms of communication and synchronization. MDPR minimizes communication and synchronization needs because there is no central process in charge of reading and distributing the input data, and collecting output results. Each participant process in the computation reads its data directly and only needs to be informed of the file positions where this data is located. Additionally, results are produced independently by each process and they are finally combined by the central process through a simple append operation.

On the other hand, the configuration of the application and the distribution of the work must be done in such a way that a similar execution time is guaranteed in each of the nodes that make up the system. MDPR has been designed in such a way that it can be dynamically configured according to the characteristics of the NUMA architecture (adjusting the number of instances used) and incorporates data distribution mechanisms (both static and dynamic) that allow the distribution of data in an unbalanced way in order to adapt the whole application to the relative performance exhibited by each node of the system. These mechanisms and their subsequent evaluation are described below.

The short read dataset (S) is partitioned into several subsets: one subset for each node in the system (Fig. 7 shows the case of two nodes: node X(S') and node Y(S'')). We have several thread groups or instances in node X (denoted as $S'1, \ldots, S'i, \ldots, S'x$, $1 <= i <= x$) and several thread groups in node Y (denoted as $S''1, \ldots, S''j, \ldots, S''y$, $1 <= j <= y$). Thread groups are generated by taking into account the memory requirements of the aligner (in particular, the size of the reference genome) and the amount of available space on the memory banks in each system. Reference genome replicas are denoted as $R'1, \ldots, R'x$, and $R''1, \ldots, R''y$. Each BWA instance executes several threads to align short reads assigned to it. It is worth mentioning that the number of instances on a heterogeneous architecture does not require to be identical. In fact, in the experimental part we will show results obtained with various combinations that we tested.

It is important to note that the division of the data set between all the nodes of the system does not imply a significant increase in communications because each group of threads is only provided with the appropriate pointers to the file where all the reads are located. That is, the distribution of data is not done by a master process that reads the data file and distributes the blocks to the other processes by using message passing mechanisms. In our case, the master process only communicates the start and end points of each block to each process; afterwards, each process will read directly that part of the file that is accessible through the shared file system that is attached to all the computing nodes (Lustre, in our case).

According to this schema, the best performance should be obtained if all thread groups consume a similar time. However, the heterogeneity of the whole system may lead to different execution times if short reads are distributed evenly. This load imbalance situation can be ameliorated by partitioning the short read dataset in an uneven way. This partitioning can be done in different ways and we have evaluated three possibilities: a static distribution that divides the dataset off-line and two dynamic distribution mechanisms that divide the dataset at run-time. Details of each mechanism follows:

1. Static distribution.
   In this mode, size of datasets S' and S'' are computed beforehand and the initial dataset of short read is divided statically at the start of execution. Each node reads then its

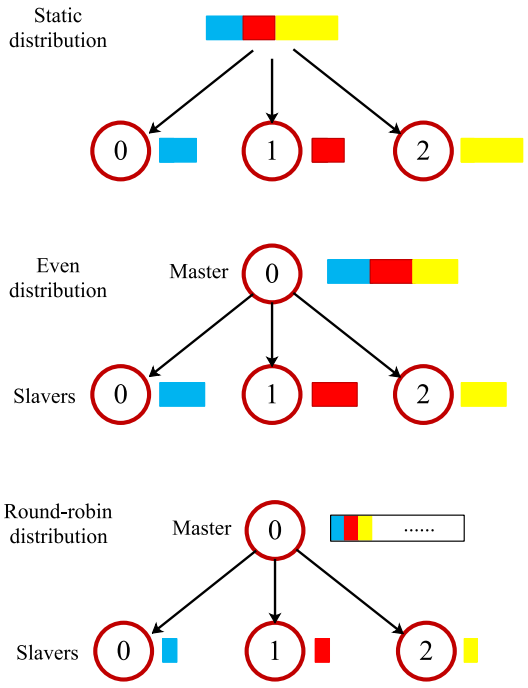

Fig. 8. Data distribution mechanisms.

block at the beginning without the need to perform new distribution operations later. The ratio between S' and S'' is computed according to the relative performance achieved by the best result achieved by DPR in each architecture. For instance, in the case of two nodes (X and Y), for a certain number of instances on the node X and on the node Y (x and y instances, respectively), the time-cost Tx and Ty are measured. Then the Ty/Tx ratio between node X and node Y is used to divide the input data S. Namely, the ratio between S' and S'' is computed according to the relative performance achieved in each node. Hence, for the node X we have $size(S') = size(S) * Ty/(Tx + Ty)$ and for the node Y we have $size(S'') = size(S) * Tx/(Tx + Ty)$. Obviously, this static distributions does not introduce any overhead at run-time but it requires the previous execution of several examples that to calculate the relative performance of each node of the system. This mechanism is illustrated at the upper part of Fig. 8.

2. Even distribution.
   In this case, all instances of MDPR receive a block of data of the same size. If the total number of instances is $(x+y)$ and the total size of the dataset (S) is $size(S)$, each instance process a block of size $(size(S)/(x+y))$. The ratio between node X and node Y could set up as x/y. In node X we have $size(S') = size(S) * x/(x+y)$, and in node Y we have $size(S'') = size(S) * y/(x+y)$. Similarly to what happens in the static distribution case, the original dataset is divided at the beginning of the execution and no significant overhead is introduced. However, this scheme does not lead in general to a good load balance if relative performance of nodes is significantly different. As shown in the central part of Fig. 8, master instance generates equal size divisions of the original dataset and each instance (slave) will process it.
3. Round-robin distribution.
   This distribution mechanism implies that a master instance distributes chunks of reads to slave instances when they have finished previous chunks. This mechanism should guarantee a better load balance than the previous one at the expense of a larger overhead at run time. Originally, each BWA thread processes a set of reads of the same size (256 K, by default). That size has shown a good trade-off between the computation time for mapping all reads in the set and the I/O time consumed with that set. In our implementation, this set size has been maintained and, therefore, chunck size is calculated by multiplying 256 K by the number of threads that are executed in each instance. For instance, if we execute BWA aligner in a system that has 64 available threads, chunk size should be at least 16 M (256 K*64) if one instance is applied or 8 M (256 K*32) if two instances are applied. This mechanism is illustrated at bottom part in Fig. 8, where chunks of short read are distributed following a round-robin way. An instance receives a new chunck when it has completed the previous one. As mentioned previously, chunk distribution simply implies the communication of the start and end points of the corresponding chunk.

6. Experimental environment

To evaluate the different versions of BWA that have been described in this article and compare them with other versions proposed in the literature, we have used two datasets and three different architectures. Below we show the details of these data sets and these architectures.

Table 1 shows the sample datasets we used in our experiments (SRR766060 and YH110112). The first one was obtained from the 1000 Genomes Project [37] and the second one corresponds to a genome sequence of a Han Chinese individual [38]. HS37D5 corresponds to the human reference genome.

Table 2 summarizes the main characteristics of the three systems used in our experiments (referred as S1, S2 and S3). Sandman and Sandman-Xeon Phi (S1) constitutes an heterogeneous system

Table 1
Experiment dataset.

Name	Data set	Data size	Read length	Read number	–
Short Read	SRR766060(SRR)	17.4GB	100	34.2 M	Paired-end
–	YH110112(YH)	4.4GB	100	14.8 M	Segment
Genome Reference	HS37D5	3GB	–	–	Human Genome

Table 2
Experiment system.

System	Hostname	Type	Processor	NUMA	Core	Thread	Memory	Data Width
S1	Sandman	Intel Xeon E5 2620	2	2	12	24	64GB	64bits
	Sandman-Xeon Phi	Intel Xeon Phi 7120	1	1	60	240	16GB	512bits
S2	Penguin	Intel Xeon E5 4620	4	4	32	64	128GB	64bits
	Batman	AMD Opteron Processor 6376	4	8	32	64	128GB	64bits
S3	AMD+Intel	AMD Opteron+Intel Xeon	17	25	126	252	576GB	64bits

made of 2 Intel Xeon E5-2620 processors and one Intel Xeon Phi 7120 accelerator. Sandman has 2 processors, 2-NUMA nodes (with 32GB of memory in each node), 12 cores and 24 available threads with hyper threading enabled. Sandman-Xeon Phi consists of 60 cores, 240 available threads and 16GB memory on card. Each core runs at 1333 Mhz and can process 4 hardware threads. The second system S2 that we used in our experiments consists of two multicore servers: Penguin (equipped with Intel processors) and Batman (equipped with AMD processors). Penguin has 4 Xeon processors, 4-NUMA nodes, 32 cores, 64 threads and 32GB memory size per processor, and Batman has 4 sockets (with 2 processors per socket which implies a total of 8-NUMA nodes). Furthermore, a large-scale system (S3) that was used in MDPR experiments consisting of 6 nodes (combining AMD Opteron and Intel Xeon processors), with a total of 126 physical cores and the capability of executing 252 threads. The architectures of these systems are illustrated in Fig. 1.

All experiments were executed using Intel Parallel Studio XE. And a minimum of 6 executions of each independent case with the same environment and configuration were carried out for the purpose of getting mean values with high confidence intervals.

7. Experimental results and evaluation

In this section we show several experiments comparing different versions of BWA in order to find the one that provides better performance. The first set of experiment shows a comparison between the basic mode of BWA (native mode) and strategies from the literature that run in offload mode and in symmetric mode on Xeon Phi systems. The next experiments analyze the behavior of the strategies based on data parallelization or data replication described in Section 5.1. Finally, experiments carried out with the MDPR strategy are shown to evaluate the performance of the three mechanisms of data distribution both on our manycore system and on our multicore systems.

7.1. Evaluation of parallelism on basic execution modes

These experiments evaluate the scalability of three basic execution modes of BWA according to the description provided in Section 3 (i.e., native, offload and symmetric modes). We ran the applications on our manycore system (Sandman and Sandman-Xeon Phi, as described in Table 2) and we used two datasets. We tested the BWA versions shown in Table 3. BWA-Xeon and BWA-Xeon Phi correspond to the original version of BWA running in native mode either in the Xeon part of the system or in the Xeon Phi part of the system. sBWA is a straightforward implementation

Table 3
BWA aligners on basic execution modes.

Architecture	Native mode	Symmetric mode	Offload mode
Xeon	BWA-Xeon	sBWA	
Xeon Phi	BWA-Xeon Phi		mBWA

of BWA running in symmetric mode with two instances of BWA aligner (one in Xeon and one in Xeon Phi) connected by MPI services. Finally, mBWA [6], runs BWA in an offload mode. It starts in the Xeon part and the main time consuming parts are off-loaded to the Xeon Phi part. Fig. 9 shows average execution times obtained by each strategy for each dataset. The longer column in one color represents the results from dataset YH, while the shorter column in the same color stands for the results from dataset SRR.

Maximal parallelism in our system is 24 threads in Xeon (Sandman) and 240 threads in Xeon Phi (Sandman-Xeon Phi). Hence, BWA-Xeon case illustrates scalability of 6, 12, 18 and 24 threads. BWA-Xeon Phi and mBWA cases use 60, 120, 180 and 240 threads. Execution of sBWA was done with the following number of threads: 6 (on the Xeon) and 60 (on the Xeon Phi), 12 and 120, 18 and 180, and 24 and 240, respectively. All strategies show a similar tendency in each dataset. Despite the high number of threads used, BWA-Xeon Phi exhibits the worst execution times, with sBWA being the best performing version. sBWA always outperforms native and offload cases. But, although sBWA simultaneously takes advantage of available cores both on the Xeon and on the Xeon Phi, the performance improvement is not linear with the number of threads.

These results can be explained because sequence alignment has large data I/O operations and is very sensitive to the time costs of such operations. Reading the reference genome and the datasets, and writing the final mappings are more expensive in terms of execution time in the Xeon Phi than in the Xeon. In the case of offload mode, mBWA performs main data I/O operation in the Xeon part instead of in the Xeon Phi part, and offloads highly parallel loops into the Xeon Phi part. Thus when comparing the performance of the same amount of threads in mBWA with that of BWA-Xeon Phi, mBWA spends approximately two thirds of the execution time spent by the BWA-Xeon Phi. The same situation occurs with BWA-Xeon and BWA-Xeon Phi, the former case reaches better performance than the latter one, despite a lower amount of threads applied in BWA-Xeon. In the case of sBWA, the asymmetric distribution of initial data compensates the loss in performance incurred by data I/O operations on the Xeon Phi.

Fig. 9 also illustrates another problem that affects all four aligners: scalability does not improve significantly as a growing number

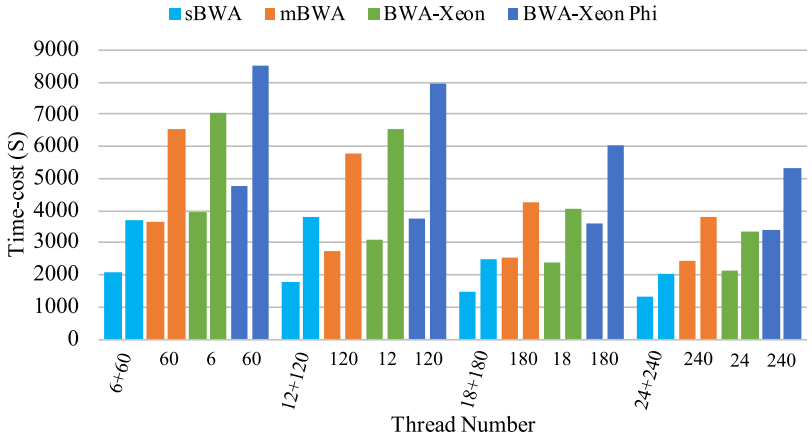

Fig. 9. Scalability of BWA aligners (Identical color in short column: SRR and in long column: YH).

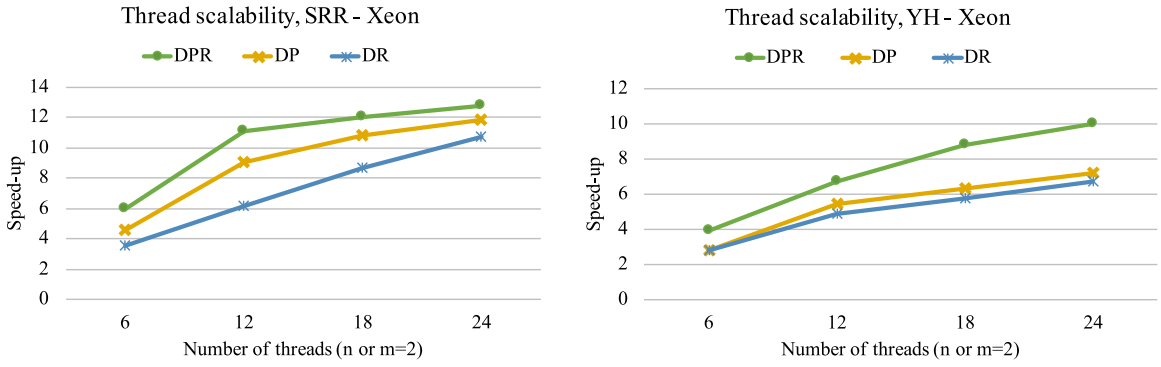

Fig. 10. Scalability comparison on Sandman.

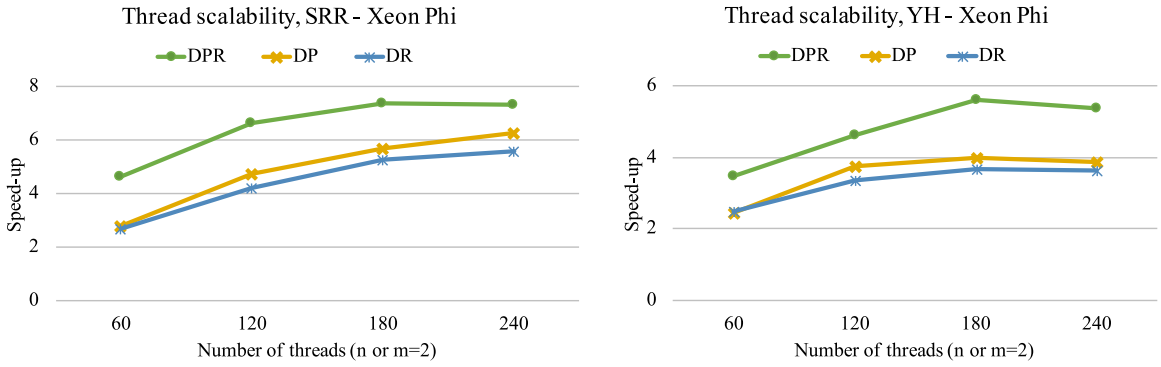

Fig. 11. Scalability comparison on Sandman-Xeon Phi.

of threads is used. As mentioned in Section 4, BWA uses a shared data structure containing the reference genome. This data structure is loaded at one particular memory bank of the system by the master thread. Thus this memory bank turns into a congestion bottleneck of performance as the number of threads grows. This memory bank needs to support more concurrent accesses coming from different threads to populate the corresponding caches.

7.2. Evaluation of data parallelism strategies

In this subsection, we show the results obtained in the evaluation of the three different data parallelism strategies described in Section 5.1. All strategies were executed in system S1.

7.2.1. Scalability of DP, DR and DPR

We set up two thread groups for execution in the strategy of DP, DR and DPR, namely $n = 2$ in DP, $m = 2$ in DR and $n = m = 2$ in DPR. Figs. 10 and 11 show relative speedups on Intel Xeon and Xeon Phi, respectively. The speed-up is calculated under dataset SRR using as a baseline case the execution time of original BWA with one thread running on Sandman.

In general, the three strategies have a consistent behavior in all cases and it can be concluded that DPR obtains the best performance, followed by DP and DR. Results in the Xeon exceed by a factor of 10 approximately those obtained in the Xeon Phi. The overhead of I/O explains part of these lack of performance, as it was mentioned in the previous section. This loss in performance is also explained because BWA does not take advantage of the 512-bit SIMD units available in the Xeon Phi. On the other hand, the effect of memory contention is attenuated thanks to the use of replicas of the reference genome. This effect is greater in the case of the Xeon system because it has two clearly separated NUMA nodes. In the case of Xeon Phi, the use of replicas allows increasing the number of concurrent accesses. But in this case, the improvement is not so significant because there are no clearly-separated NUMA nodes and the performance is improved until a point is reached where the ring connecting all the memory banks is saturated.

7.2.2. Analysis of thread groups and data replicas in DP, DR and DPR

One of the factors associated with the design of the DP, DR and DPR strategies is the number of thread groups or the number of replicas used by them. With the experiments shown in this subsection we want to answer questions like what is the ideal number of groups or replicas that should be used in a particular architecture? or can we use any practical rule that can be applied to deduce such number?

We executed DP, DR and DPR varying the number of thread groups and replicas that were used (x and y). In all cases, we used the maximal number of available cores. This means that 24 and 240 threads were used in all the cases on the Xeon part (Sandman) and on the Xeon Phi part (Sandman-Xeon Phi), respectively. The number of OpenMP threads used at each instance (tx and ty) on the Xeon and Xeon Phi are $tx = 24/x$ and $ty = 240/y$. For instance, if 2 thread groups are applied at the whole system, each Xeon thread group utilizes 12 OpenMP threads and each Xeon Phi thread group utilizes 120 OpenMP threads. Thus, the number of thread groups 2, 3, 4, 6 and 12, could stand for the number of OpenMP threads 12, 8, 6, 4 and 2 in each thread group in Fig. 12 and 120, 80, 60, 40 and 20 in each thread group in Fig. 13. For DR and DPR, the number of thread groups is limited by memory consumption of each group. Basically, one thread group of BWA aligner consumes about 5GB memory, which is mainly required to store the human genome sequence reference (4.4GB). According to the memory size available on the Xeon part (64GB) and on the Xeon Phi part (16GB), 12 and 3 should be the maximum number of thread groups that can be launched at each part. We tested DP, DR and DPR using thread groups of 2, 3, 4, 6 and 12 thread groups on both systems, but in the cases of DPR-BWA and DR-BWA only 3

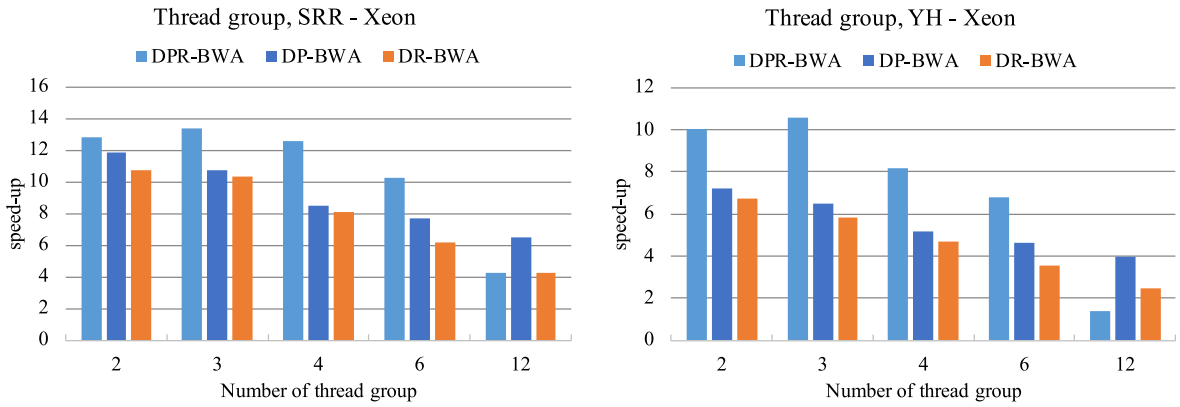

Fig. 12. Performance comparison on Sandman.

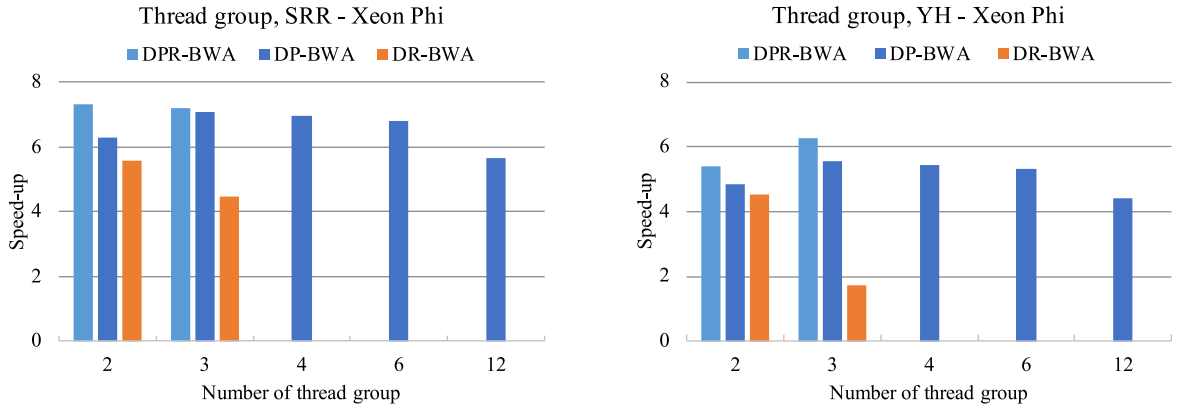

Fig. 13. Performance comparison on Sandman-Xeon Phi.

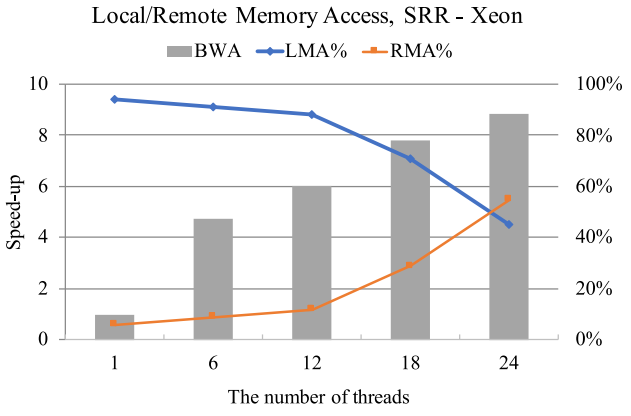

Fig. 14. BWA memory access.

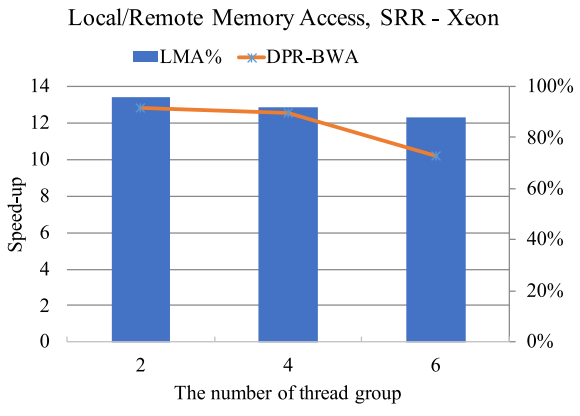

Fig. 15. DPR-BWA memory access.

thread groups were applied in the experiment of Xeon Phi due to the memory limitations.

Results are shown in Figs. 12 and 13. It can be observed that the best performance is achieved when a small number of groups are used (2 or 3). Beyond these values, adding more groups worsens performance. These results seem coherent if we take into account the number of NUMA nodes in the system (2 in the case of Sandman and 1 in the case of Sandman Phi). Having more replicas helps distribute traffic between different memory banks and prevents a single bank from becoming congested. However, adding more replicas increases the overhead and reduces the performance of the cache because there is more competition between different groups of threads without increasing the transference capacity of each memory bank. Threads that belong to each group also take advantage of those cache entries that contain the index positions which are read in the first stages of the alignment. By increasing the number of thread groups, the usage of index entries in the cache memory gets worse and there is an increasing number of cache misses that does not compensate for the reduction of memory congestion.

The performance improvements observed in these experiments are based on the idea that each group of threads will make accesses to the memory node of its NUMA domain, thus taking advantage of the greatest available bandwidth. Using the NUMATOP tool, we obtained measurements on the distribution of memory accesses between the different memory nodes. Figs. 14 and 15 show

the behavior of the original version of BWA and our DPR strategy when they were executed in Sandman, a system that has 2 NUMA nodes formed by a processor and a memory bank. The two figures show, on the one hand, the proportion of local memory accesses (LMA) and the speedup of the application compared to the sequential execution. As can be seen in Fig. 14, BWA exhibits a proportion of local memory access greater than 80% if it is executed in a single processor (12 threads).

When increasing the parallelism, the additional threads are executed in the second processor and all of them will generate remote accesses of memory because they have to access the node where the reference index is located. The LMA index decreases to just over 40%. As a result, remote accesses have a negative effect on the execution time of the application because latency is higher for remote accesses and they increase congestion in the node where the index is located. This negative impact on the execution time explains that when doubling from 12 to 24 threads the speed-up only improves by a factor of just over 2.

On the contrary, the use of instances (as shown in Fig. 15) in DPR always maintains a proportion of local accesses above 80% although the number of instances used is increased. There is no negative impact on memory access times nor does congestion occur. Anyway, as we saw earlier, the best speedup (factor 12, approximately) is obtained when the number of instances is low (2 or 4, which means one or two instances executed in each NUMA node). If the number of instances is too high (6 or more) the groups of threads compete and interfere with the use of the cache memory without improving the access times to the main memory.

7.2.3. Performance evaluation of MDPR

This last set of experiments analyzes the performance of the MDPR strategy, evaluating the three data distribution policies described in Section 5.2.

MDPR-BWA allows the execution of BWA aligner in the symmetric mode and it can also be used on distributed memory systems. The best execution time should be achieved if reads are distributed so that all the threads finish at the same time. The distribution of reads can be done in different ways taking into account that there is a trade-off between the overhead incurred and the load balance that can be achieved. In our case, we have evaluated a static mechanism that requires some benchmarking executions beforehand but does not incur any overhead at run time. We have also tested two dynamic methods (even distribution and round-robin), which incur in more overhead in execution time but do not require any prior benchmarking.

In these experiments, we used the same datasets used in previous experiments (SRR+YH) and three different heterogeneous systems. The first one (referred as S1) is the one equipped with an Intel Xeon and Xeon Phi (Sandman and Sandman-Xeon Phi used in previous sections). The second one (referred as S2) is made of an Intel-based server and an AMD-based server. Furthermore, we also tested MDPR in a large system (S3) that comprises 6 nodes (made of Intel Xeon and AMD Opteron processors), 17 sockets, 25 NUMA domains, capable of running 252 threads.

According to the results obtained in the evaluation of DPR with different number of replicas the following configurations of MDPR were tested in S1: 1+1(2), 2+1(3), 2+2(4), 2+3(5) and 4+2(6). The first value corresponds to the number of replicas in the Xeon part and the second value corresponds to the number of replicas in the Xeon Phi. System S2 has 4+4 sockets and 4+8 NUMA domains, while system S3 has a total of 17 sockets and 25 NUMA domains. Thus, the configurations evaluated in S2 are 1+1(2), 2+2(4), 4+4(8) and 4+8(12); in S3 the configurations were 6, 9, 17 and 25. All available hardware threads were used in all cases of MDPR experiments (i.e., Intel Xeon and Xeon Phi could use 24 threads plus 240 threads, the S2 system could support 64 threads each, and S3 could provide 252 threads).

As shown in Figs. 16–18, the static distribution of MDPR achieves the best execution times in all cases. The even distribution is the worst strategy since it only makes an initial distribution of the data without subsequent corrections according the relative performance of the nodes. As an intermediate strategy, the round-robin method obtains results close to the static one without the need for preliminary setup computations. On the other hand, if we analyze the number of replicas used, it is observed that the best results are obtained when the number of replicas is adjusted to the number of NUMA domains available in the system. This behavior has already been observed in the DPR experiments on the manycore system and is confirmed in the execution of MDPR in systems S2 and S3.

The results also show that, although the even distribution is the one that achieves worst results in all scenarios, this phenomenon is more significant when applied in heterogeneous systems where the differences in performance between its components is very significant. In our case, this is observed in the S1 system, where the Xeon Phi part is significantly slower than the Xeon part (as seen in previous experiments). Therefore, when the same amount of data is distributed to each instance, the execution in the instances running in the slowest component will determine the overall execution time. As the degree of heterogeneity decreases and the number of instances increases, the performance differences of both dynamic data distribution strategies are close to the performance of the static strategy. If we take into account the cost incurred by the static strategy to compute the relative performance of each computing node, the use of the dynamoc round-robin strategy constitutes a good trade-off between the reduction in execution times and the overheads added at run-time.

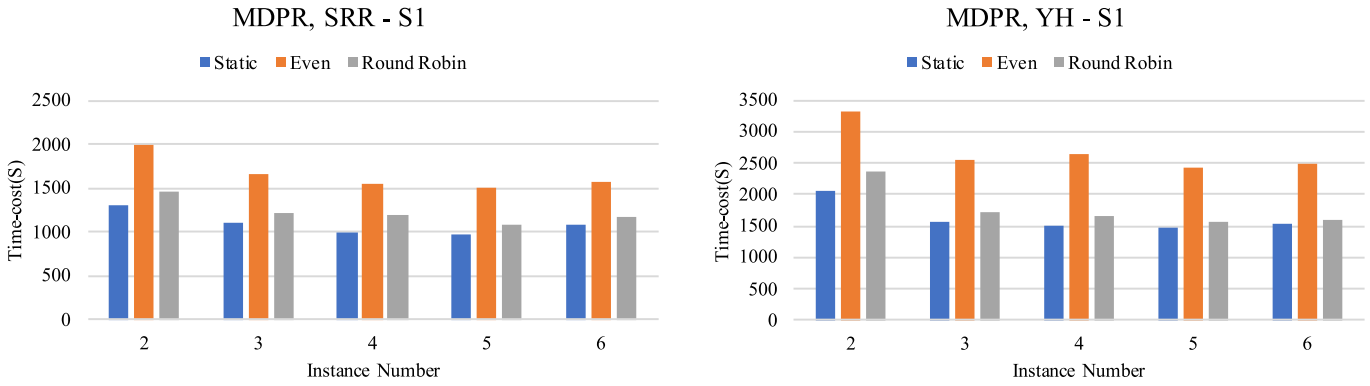

Fig. 16. MDPR comparison on Xeon and Xeon Phi(S1).

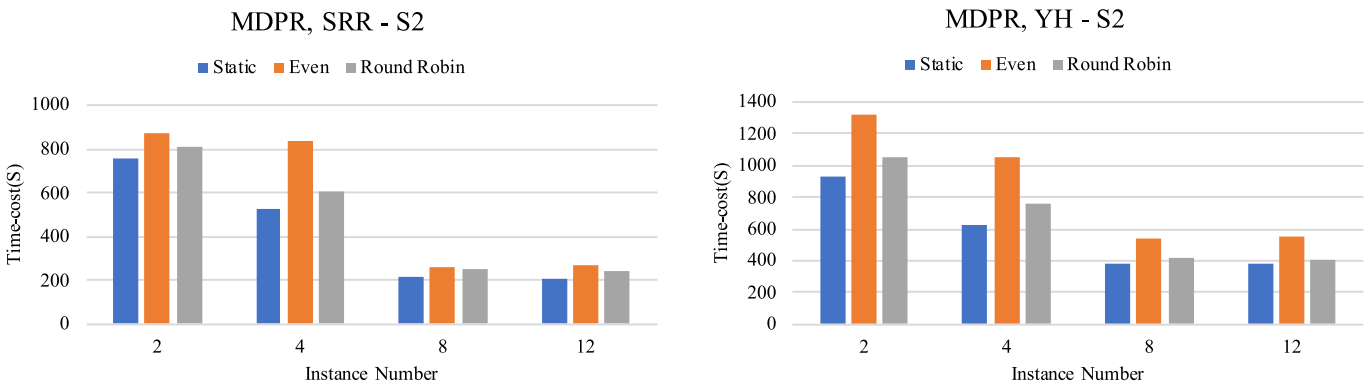

Fig. 17. MDPR comparison on Intel and AMD NUMA nodes(S2).

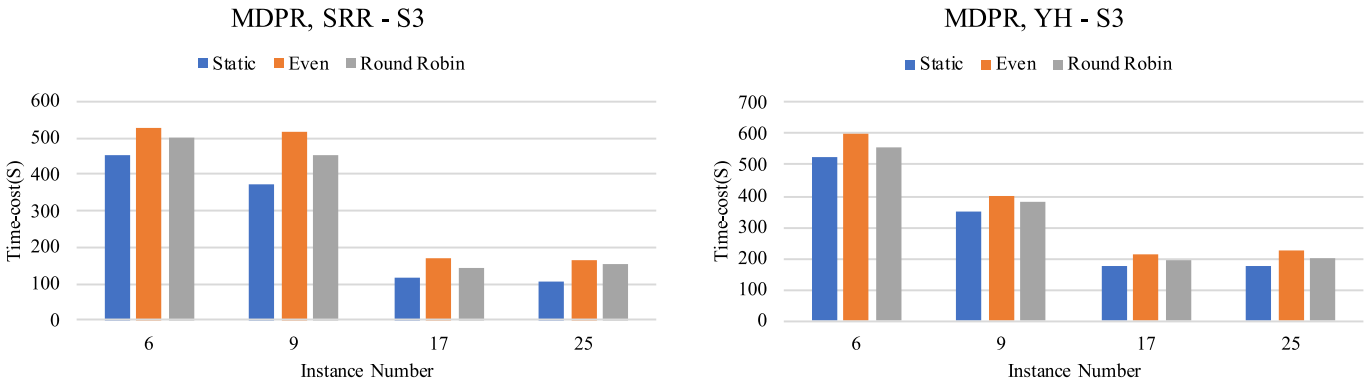

Fig. 18. MDPR comparison on Intel and AMD NUMA nodes(S3).

8. Discussion and conclusion

In this paper, we have evaluated different data parallelism strategies to execute BWA aligner in heterogeneous architectures made of multicore and/or manycore nodes. BWA has scalability limitations when large numbers of threads are used. The use of a shared reference genome generates problems of memory contention that explain such scalability issues.

We have analized several strategies in order to reduce the negative impact incurred by BWA due to memory contention. The basic idea that inspires our strategies is to divide the application into a series of groups (or instances) that will cooperate in the tasks of read alignment; each group is formed by a certain number of threads that will process each chunk of data in which the original dataset is divided. In addition, the application can use data replication. In our case, this serves to replicate the reference genome in each group or instance. The strategy that incorporates all these elements is the one that we referred as MDPR and is the one that has obtained the best results, in general. MDPR can be used in any heterogeneous systems based on manycore and/or multicore nodes.

From our experiments, we have observed that the best MDPR results are obtained when the number of instances is equal to the number of NUMA domains of the underlying architecture. In this way, the structure of the application is being adjusted to the characteristics of the different nodes that build the whole system: there are as many groups as there are NUMA domains and each group executes with as many threads as cores the processor of that domain has.

In our study we have also analyzed three strategies to perform the distribution of data among the different groups that are created in MDPR. The best execution times were obtained by the static strategy that initially distributes the entire data set taking into account the performance of each node of the system, which were previously evaluated through a benchmarking process. However, we believe that the use of a dynamic strategy that distributes data chunks in a round-robind fashion is the best practical alternative because the loss of performance is not significant and its use does not require the need to perform any previous benchmarking of nodes.

First, we evaluate different versions of BWA that run on manycore systems in native mode, offload mode and symmetric mode. Second, we analyze different versions of BWA that using different strategies to parallelize the distribution of short reads or to replicate the reference genome.

Tables 4–7 summarize the best results achieved in all our experiments by each BWA-based aligner presented in this paper when they were executed with the maximum number of resources. Each entry in the tables shows the aligner that was used, the system on which it was executed, the dataset, the average execution time that was obtained, the corresponding speed-up, the number of groups or instances used and the total number of threads. The speed up is obtained by comparison with the original implementation of BWA (BWA-Pthread) using one thread with SRR dataset (first entry in Table 4).

Tables 4 and 5 show the results obtained by different BWA versions executed using a native o a offload method in the Xeon part or in the Xeon-Phi part, respectively, of our S1 system. In both cases we have also compared an official BWA implementation (referred as BWA-ALN-Xeon-Phi) from Intel Corporation [39]. BWA-ALN-Xeon-Phi was tested in native mode (Tables 4 and 5) and it was also tested in symmetric mode in S1 cluster, as shown in Table 6.

Table 4
BWA performance summary on Intel Xeon.

Aligner	System	Dataset	Time-cost(s)	Speed-up	Group num	Thread num
BWA-Pthread	S1-Xeon	SRR	18,658	1X	1	1
BWA-ALN-Xeon-Phi	S1-Xeon	SRR	1873	10X	1	24
		YH	2985	6.3X	1	24
DP	S1-Xeon	SRR	1580	11.8X	2	24
		YH	2591	7.2X	2	24
DR	S1-Xeon	SRR	1740	10.7X	2	24
		YH	2784	6.7X	2	24
DPR	S1-Xeon	SRR	1392	13.4X	3	24
		YH	1777	10.5X	3	24

Table 5
BWA Performance Summary on Intel Xeon Phi.

Aligner	System	Dataset	Time-cost(s)	Speed-up	Group num	Thread num
BWA-ALN-Xeon-Phi	S1-Xeon Phi	SRR	2895	6.4X	1	240
		YH	4623	4X	1	240
mBWA	S1-Xeon Phi	SRR	2427	7.7X	1	240
		YH	3817	4.9X	1	240
DP	S1-Xeon Phi	SRR	2630	7.1X	3	240
		YH	3395	5.5X	3	240
DR	S1-Xeon Phi	SRR	3347	5.6X	2	240
		YH	4118	4.5X	2	240
DPR	S1-Xeon Phi	SRR	2555	7.3X	2	240
		YH	2955	6.3X	3	240

Table 6
BWA performance summary on S1.

Aligner	System	Dataset	Time-cost(s)	Speed-up	Group num	Thread num
BWA-ALN-Xeon-Phi	S1	SRR	1156	16.1X	2(1+1)	24+240
		YH	1733	10.8X	2(1+1)	24+240
sBWA	S1	SRR	1301	14.3X	2(1+1)	24+240
		YH	2047	9.1X	2(1+1)	24+240
MDPR-static	S1	SRR	966	19.3X	5(2+3)	24+240
		YH	1477	12.6X	5(2+3)	24+240
MDPR-even	S1	SRR	1501	12.4X	5(2+3)	24+240
		YH	2443	7.6X	5(2+3)	24+240
MDPR-roundrobin	S1	SRR	1086	17.2X	5(2+3)	24+240
		YH	1584	11.8X	5(2+3)	24+240

Table 7
BWA performance summary on S2 and S3.

Aligner	System	Dataset	Time-cost(s)	Speed-up	Group num	Thread num
MDPR-static	S2	SRR	204	91.5X	12(4+8)	64+64
		YH	385	48.5X	12(4+8)	64+64
	S3	SRR	109	171.2X	25	252
		YH	175	106.6X	25	252
MDPR-even	S2	SRR	260	71.8X	8(4+4)	64+64
		YH	535	34.9X	8(4+4)	64+64
	S3	SRR	167	111.7X	25	252
		YH	212	88X	17	252
MDPR-roundrobin	S2	SRR	245	76.2X	12(4+8)	64+64
		YH	411	45.4X	12(4+8)	64+64
	S3	SRR	145	128.7X	17	252
		YH	195	95.7X	17	252

As seen in Tables 4 and 5, our strategy DPR that use data replication and data parallelization obtain better execution times in general. DPR achieved a speed-up of 13.4 and 10.5 for SRR and YH, respectively, in the Xeon case, and a speed-up of 7.3 and 6.3 in the Xeon-Phi case. Only mBWA achieved a slightly better result in the case of SRR dataset with a seep-up of 7.7.

Table 6 shows the best results obtained by the strategies running in symmetric mode on the entire S1 system. In this case, two versions of MDPR (with static distribution and with round-robin) obtain better results than the other two strategies that do not use data replication (BWA-ALN-Xeon-Phi) and sBWA. The best strategy is MDPR-static with speed-ups of 19.3 and 12.6, respectively, while MDPR-roundrobin achieves values of 17.2 and 11.8. It is worth noting that although our MDPR approach allows a significant improvement of performance in manycore systems based on Intel Xeon Phi, the characteristics of this accelerator (limited I/O capacity and main memory interconnected by a ring bus) limit its potential performance in comparison with multicore systems with NUMA architecture, if we take into account the total number of cores used. Speed-up values obtained at the S1 system are significantly lower that those obtained at S2 and S3 systems (see Table 7), but the total number of cores in S1 was greater than that used in S2 and S3.

Finally, Table 7 shows the results of the different MDPR variants when running on larger systems formed by several heterogeneous

multicore nodes. MDPR with static distribution achieves the best results, followed by the round-robin version.

From all the results obtained, we can conclude that the MDPR strategy is a very attractive alternative to reduce the execution time of BWA in the distributed systems, either by using a static distribution or a dynamic round-robin pattern.

Our study also reveals an idea that can be applied to other problems in the field of bioinformatics. Although in this field there is a great growth in the quantity and quality of existing data sets, in practice, the resolution of problems such as sequence alignment are limited mainly by the size of reference genomes (usually between 3GB and 30GB) which are the structure that should remain in main memory all the time. Data files with sequences are read gradually in the form of chunks and, therefore, they do not suppose any significant limitation in terms of memory requirements in current systems. Consequently, the design of parallel applications by using an architecture based on instances that contain replicas of certain data structures constitutes a solution with low algorithmic complexity that can provide significant improvements in systems whose memory architecture is of the NUMA type, as shown by MDPR in the case of BWA.

Acknowledgements

This work is supported by the Spanish Government Project TIN201784553C21R] and China Scholarship Council [201406890007].

References

[1] C.F. Baes, M.A. Dolezal, J.E. Koltes, B. Bapst, E. Fritz-Waters, S. Jansen, C. Flury, H. Signer-Hasler, C. Stricker, R. Fernando, et al., Evaluation of variant identification methods for whole genome sequencing data in dairy cattle, BMC Genomics 15 (1) (2014) 948.
[2] I. Buck, Gpubench: evaluating GPU performance for numerical and scientific application, in: Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^2'04), 2004.
[3] L. Stein, Genome annotation: from sequence to biology, Nat. Rev. Genet. 2 (7) (2001) 493.
[4] J.G. Reid, A. Carroll, N. Veeraraghavan, M. Dahdouli, A. Sundquist, A. English, M. Bainbridge, S. White, W. Salerno, C. Buhay, et al., Launching genomics into the cloud: deployment of mercury, a next generation sequence analysis pipeline, BMC Bioinformatics 15 (1) (2014) 30.
[5] S. Pabinger, A. Dander, M. Fischer, R. Snajder, M. Sperk, M. Efremova, B. Krabichler, M.R. Speicher, J. Zschocke, Z. Trajanoski, A survey of tools for variant analysis of next-generation genome sequencing data, Brief. Bioinformatics 15 (2) (2014) 256–278.
[6] Y. Cui, X. Liao, X. Zhu, B. Wang, S. Peng, mBWA: a massively parallel sequence reads aligner, in: 8th International Conference on Practical Applications of Computational Biology & Bioinformatics (PACBB 2014), Springer, 2014, pp. 113–120.
[7] H. Li, R. Durbin, Fast and accurate short read alignment with burrows–wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.
[8] S. Chen, M.A. Senar, Accelerating BWA aligner using multistage data parallelization on multicore and manycore architectures, Procedia Comput. Sci. 80 (2016) 2438–2442.
[9] S. Chen, M.A. Senar, Improving performance of genomic aligners on intel xeon phi-based architectures, in: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2018, pp. 570–578.
[10] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, D.J. Lipman, Gapped blast and psi-blast: a new generation of protein database search programs, Nucleic Acids Res. 25 (17) (1997) 3389–3402.
[11] B. Langmead, C. Trapnell, M. Pop, S.L. Salzberg, Ultrafast and memory-efficient alignment of short dna sequences to the human genome, Genome Biol. 10 (3) (2009) R25.
[12] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, J. Wang, Soap2: an improved ultrafast tool for short read alignment, Bioinformatics 25 (15) (2009) 1966–1967.
[13] H. Li, R. Durbin, Fast and accurate short read alignment with burrows–wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.
[14] J. Zhang, H. Lin, P. Balaji, W.-c. Feng, Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures, in: Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on, IEEE, 2013, pp. 377–384.
[15] N. Kathiresan, M.R. Temanni, R. Al-Ali, Performance improvement of BWA MEM algorithm using data-parallel with concurrent parallelization, in: Parallel, Distributed and Grid Computing (PDGC), 2014 International Conference on, IEEE, 2014, pp. 406–411.
[16] E. de Araujo Macedo, A. Boukerche, Hybrid MPI/OpenMP strategy for biological multiple sequence alignment with DIALIGN-TX in heterogeneous multicore clusters, in: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, IEEE, 2011, pp. 418–425.
[17] S.L. Olivier, A.K. Porterfield, K.B. Wheeler, J.F. Prins, Scheduling task parallelism on multi-socket multicore systems, in: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, ACM, 2011, pp. 49–56.
[18] J. Lenis, M.A. Senar, A performance comparison of data and memory allocation strategies for sequence aligners on NUMA architectures, Cluster Comput. 20 (3) (2017) 1909–1924.
[19] Y.-T. Chen, J. Cong, J. Lei, P. Wei, A novel high-throughput acceleration engine for read alignment, in: Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on, IEEE, 2015, pp. 199–202.
[20] E.J. Houtgast, V.-M. Sima, K. Bertels, Z. Al-Ars, Gpu-accelerated BWA-MEM genomic mapping algorithm using adaptive load balancing, in: International Conference on Architecture of Computing Systems, Springer, 2016, pp. 130–142.
[21] E.J. Houtgast, V.-M. Sima, K. Bertels, Z. Al-Ars, An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm, in: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on, IEEE, 2015, pp. 221–227.
[22] H.-C. Ng, S. Liu, W. Luk, Reconfigurable acceleration of genetic sequence alignment: a survey of two decades of efforts, in: Field Programmable Logic and Applications (FPL), 2017 27th International Conference on, IEEE, 2017, pp. 1–8.
[23] H. Li, N. Homer, A survey of sequence alignment algorithms for next-generation sequencing, Brief. Bioinformatics 11 (5) (2010) 473–483.
[24] E.J. Houtgast, V.-M. Sima, G. Marchiori, K. Bertels, Z. Al-Ars, Power-efficiency analysis of accelerated BWA-MEM implementations on heterogeneous computing platforms, in: ReConFigurable Computing and FPGAs (ReConFig), 2016 International Conference on, IEEE, 2016a, pp. 1–8.
[25] E.J. Houtgast, V.-M. Sima, K. Bertels, Z. Al-Ars, GPU-accelerated BWA-MEM genomic mapping algorithm using adaptive load balancing, in: International Conference on Architecture of Computing Systems, Springer, 2016b, pp. 130–142.
[26] J.M. Abuín, J.C. Pichel, T.F. Pena, J. Amigo, BigBWA: approaching the burrows–wheeler aligner to big data technologies, Bioinformatics 31 (24) (2015) 4003–4005.
[27] J.M. Abuín, J.C. Pichel, T.F. Pena, J. Amigo, Sparkbwa: speeding up the alignment of high-throughput dna sequencing data, PLoS ONE 11 (5) (2016) e0155461.
[28] S.-H. Chan, J. Cheung, E. Wu, H. Wang, C.-M. Liu, X. Zhu, S. Peng, R. Luo, T.-W. Lam, MICA: a fast short-read aligner that takes full advantage of intel many integrated core architecture (mic), arXiv:1402.4876 (2014).
[29] X. Tian, H. Saito, S.V. Preis, E.N. Garcia, S.S. Kozhukhov, M. Masten, A.G. Cherkasov, N. Panchenko, Practical SIMD vectorization techniques for intel® xeon phi coprocessors, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, IEEE, 2013, pp. 1149–1158.
[30] S. Memeti, S. Pllana, Accelerating dna sequence analysis using intel (r) xeon phi (tm), in: Trustcom/BigDataSE/ISPA, 2015 IEEE, volume 3, IEEE, 2015, pp. 222–227.
[31] L. Wang, Y. Chan, X. Duan, H. Lan, X. Meng, W. Liu, XSW: accelerating biological database search on xeon phi, in: Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, IEEE, 2014, pp. 950–957.
[32] L. Ping, Speeding up large-scale next generation sequencing data analysis with pBWA, J. Appl. Bioinf. Comput. Biol. (2012).
[33] C. Herzeel, T.J. Ashby, P. Costanza, W. De Meuter, Resolving load balancing issues in BWA on NUMA multicore architectures, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2013, pp. 227–236.
[34] T. Cramer, D. Schmidl, M. Klemm, D. an Mey, Openmp programming on intel r xeon phi tm coprocessors: an early performance comparison, in: Proc. Many Core Appl. Res. Community (MARC) Symp, 2012, pp. 38–44.
[35] J. Reinders, An overview of programming for intel xeon processors and intel xeon phi coprocessors, Intel Corporation, Santa Clara, 2012.
[36] Y. Cui, X. Liao, X. Zhu, B. Wang, S. Peng, B-Mic: an ultrafast three-level parallel sequence aligner using mic, Interdiscip. Sci. 8 (1) (2016) 28–34.
[37] .G.P. Consortium, et al., A global reference for human genetic variation, Nature 526 (7571) (2015) 68.
[38] J. Wang, Y. Li, R. Luo, B. Liu, Y. Xie, Z. Li, X. Fang, H. Zheng, J. Qin, B. Yang, et al., Updated genome assembly of yh: the first diploid genome sequence of a han chinese individual (version 2, 07/2012), GigaScience Database (2012).
[39] C.I. You, Liang (Intel); Congdon, Building and optimizing BWA-ALN 0.5.10 for intel xeon phi coprocessors, URL https://software.intel.com/en-us/articles/recipe-building-and-optimizing-bwa-aln-0510-for-intel-xeon-phi-coprocessors (2014).