parSRA: A Framework for the Parallel Execution of Short Read Aligners on Compute Clusters

Jorge González-Domínguez*

Computer Architecture Group, University of A Coruña, Spain

Christian Hundt, Bertil Schmidt

Parallel and Distributed Architectures Group, Johannes Gutenberg University Mainz, Germany

Abstract

The growth of next generation sequencing datasets poses as a challenge to the alignment of reads to reference genomes in terms of both accuracy and speed. In this work we present *parSRA*, a parallel framework to accelerate the execution of existing short read aligners on distributed-memory systems. *parSRA* can be used to parallelize a variety of short read alignment tools installed in the system without any modification to their source code. We show that our framework provides good scalability on a compute cluster for accelerating the popular *BWA-MEM* and *Bowtie2* aligners. On average, it is able to accelerate sequence alignments on 16 64-core nodes (in total, 1024 cores) with speedup of 10.48 compared to the original multithreaded tools running with 64 threads on one node. It is also faster and more scalable than the *pMap* and *BigBWA* frameworks. Source code of parSRA in C++ and UPC++ running on Linux systems with support for *FUSE* is freely available at
https://sourceforge.net/projects/parsra/.

Keywords: Short Read Alignment, High Performance Computing, Multicore Clusters, Bioinformatics, PGAS

*Principal corresponding author: Jorge González-Domínguez
Email addresses: jgonzalezd@udc.es (Jorge González-Domínguez), hundt@uni-mainz.de (Christian Hundt), bertil.schmidt@uni-mainz.de (Bertil Schmidt)

1. Introduction

Short Read Alignment (SRA) is a crucial step in many bioinformatics pipelines. It consists in mapping DNA fragments (called reads) onto a reference genome, in order to locate the genomic coordinates these fragments come from. The rapid progress of next generation sequencing (NGS) technologies has led to large-scale datasets containing hundreds of millions or even billions of reads, which makes the SRA step time consuming.

Although efficient *seed-and-extend* based algorithms that provide high-quality alignments have been developed, their associated runtimes are still high. Examples include *GASSST* [1], *Bowtie2* [2], *GEM* [3], *SeqAlto* [4], *BWA-MEM* [5] and *CUSHAW3* [6]. The main algorithmic idea applied by all these tools are based on the fact that significant alignments usually contain short exact matches (so called *seeds*). Typical short read aligners thus map a given read by first identifying such seeds on the given reference genome. This is usually accomplished by using a pre-computed index data structure that allows for fast retrieval of short exact matches between query and reference genome. Subsequently, these seeds are extended and refined under certain constraints, such as minimal percentage identity or length, in order to filter out irrelevant seeds. Finally, more sophisticated but also computationally more expensive approaches (*e.g.,* dynamic programming based alignment algorithms) are employed to obtain the final alignments from the seeds. It should be noted that the time needed for the alignment of each read can vary as it depends on the number of associated seeds.

Parallelization can be used to accelerate this procedure. Most existing SRA tools only provide shared memory parallelism based on multi-threading, which limits their execution to single compute nodes. In order to overcome this limitation, there exist parallel implementations of certain SRA tools that can be executed on multicore clusters and exploit the computing capabilities of several nodes (*e.g., pBWA* [7] and *merAligner* [8]). However, the accuracy of the results provided by these multinode implementations is limited to only one mapping approach and they do not offer portability for the underlying aligner. For instance, *pBWA* is limited to a particular version of the *BWA* aligner [9].

Research on SRA approaches is still evolving and new methods with better accuracy in some scenarios are continuously developed. Therefore, it is not advisable to limit parallel frameworks to work with only one type of alignment (e.g. *pBWA* only works with the outdated 0.5.9 version of the *BWA method*). *pMap* [10] is a parallel framework that allows for working with several existing and previously installed tools (*e.g., Bowtie2*, *BWA-MEM* or *CUSHAW3*). It splits the workload among nodes and uses the selected method to complete the alignment in parallel on multiple compute nodes within a cluster. However, as will be shown in the experimental evaluation, the scalability of *pMap* is low even for a moderate number of nodes. Recently, approaches based on the map-reduce paradigm have been presented for distributed execution of the BWA aligner [11–13]. However, their scalability is limited to a small number of compute nodes.

In this paper we present *parSRA*, a novel framework to parallelize SRA on multicore clusters which can work with different underlying methods and provides significantly better scalability than *pMap*. *parSRA* can use the most suitable alignment method for each situation, and even more accurate methods that can be developed in future. Moreover, *parSRA* is even more portable than *pMap* as its configuration file allows for the users to parallelize the execution of existing SRA tools without the need to modify the source code of *parSRA* or the aligner.

The rest of the paper is organized as follows. Section 2 reviews some related work. Our parallelization approach is described in Section 3. Experimental evaluations are presented in Section 4. Section 5 concludes the paper.

2. Related Work

The implementation of parallel tools for SRA that resort to accelerators to reduce their runtime has attracted extensive research interests. The most popular accelerators for SRA are GPUs, and some examples of GPU-based tools are *CUSHAW* [14], *CUSHAW2-GPU* [15], *BarraCUDA* [16], *SOAP3* [17], *SOAP3-db* [18] and *nvBowtie* [19]. Other examples include FPGA and Xeon Phi implementations such as [20] and [21].

So far, not much effort has been made to develop tools able to exploit the characteristics of compute clusters. For instance, there is no parallel SRA implementation using workflow systems such as Swift/T [22] or SciCumulus [23]. Three examples of map-reduce based SRA aligners are *BigBWA* [11], SEAL [12] and *SparkBWA* [13], which are limited to the BWA method [9]. Regarding the message-passing paradigm, *pBWA* [7] and *pMap* [10] use MPI to distribute the reads among the processes and align the assigned reads on each process. While *pBWA* is also limited to the *BWA* aligner, *pMap* is portable enough to be able to work with several different aligners. The current publicly available version of *pMap* provides support for some popular aligners. Moreover, the source code can be modified in the case that we want to work with a new aligner. Both *pBWA* and *pMap* suffer from two major problems that limit their scalability. First, the overhead of their initial file splitting is significant, especially when increasing the number of processes. Moreover, they apply a static distribution that assigns the same number of reads to each MPI process. As the time to align one read can vary, a simple static distribution cannot achieve good load balancing.

In this paper we describe *parSRA*, a novel framework to execute short read aligners on compute clusters. Our parallel implementation overcomes the scalability issues of *pMap* thanks to:

1. A fast splitting of the input reads using the *FUSE* kernel module [24].
2. Gathering of results into a unique output file using OS commands.
3. A balanced on-demand distribution of the reads based on the shared locks of UPC++ [25].

UPC++ is an extension of C++ for parallel computing which has evolved from Unified Parallel C (UPC) [26]. PGAS languages (such as UPC, Co-Array Fortran [27] or Titanium [28]) are often easier to use than their message passing counterparts [29, 30] and can also obtain better performance by using efficient one-sided communication [31–33]. UPC++ combines these advantages of the PGAS model with object oriented programming. Both UPC and UPC++ have recently been used for the parallelization of bioinformatics applications [8, 34, 35].

merAligner [8] is a parallel UPC-based sequence aligner for distributed-memory architectures which obtains good scalability on multicore clusters. Although this tool is also an aligner, it is focused on scenarios where the reference genome is very large and thus represented as a (distributed) collection of contigs. According to the results provided by the authors, the whole procedure (index construction and sequence mapping) is faster in *merAligner* than in *pMap*. However, the scalability of the alignment step on several nodes is lower than that of *pMap*. *merAligner* also optimizes the distribution of the genome in case that it is too large to fit in one node. However, the goal of our work is the parallelization of the type of aligners presented in the previous section, that work with genomes that fit in the memory of one node (which is typically the case for a human reference genome; the most common use case). There is no restriction related to the size of the dataset with the reads to align in *parSRA*. Furthermore, *merAligner* does not provide portability to existing aligners.

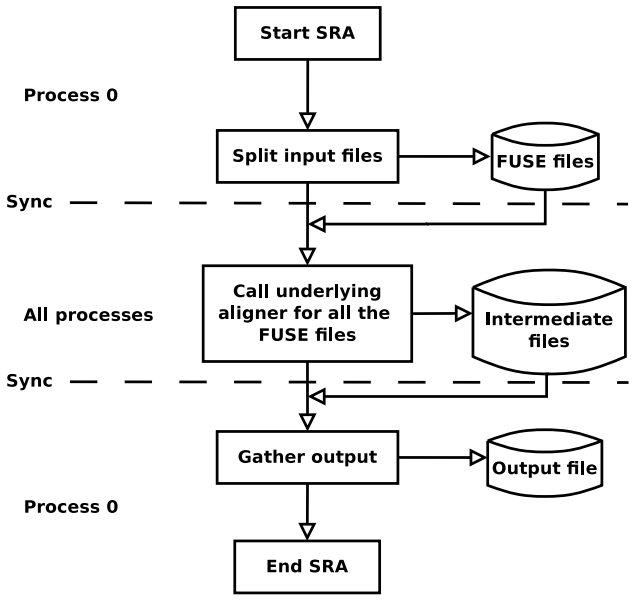

Figure 1: Workflow of *parSRA*. All processes work in parallel to align different reads while Process 0 splits the input file and gathers the output.

3. Implementation

The aim of *parSRA* is to accelerate the SRA while preserving the quality of the results provided by the underlying aligner. Therefore, we do not modify the source code of the aligners. Figure 1 shows the workflow of a *parSRA* execution. The procedure starts with one process splitting the files through *FUSE* as will be explained in Section 3.1. Once all the virtual files have been created, all processes simultaneously align the reads. Each process calls the underlying aligner (*e.g., BWA-MEM* or *Bowtie2*) several times with different *FUSE* files. The assignment of virtual files to processes is described in Section 3.2. Each process writes its results into a different intermediate file (*i.e.,* there are as many intermediate files as processes). Finally, once all reads have been aligned, the results are gathered into a unique output file. This is carried out again by only one process using the OS commands to concatenate files.

All the information to perform the alignment is indicated by the user in a configuration file. One important parameter that must be included in this configuration file is the number of blocks (virtual files) that will be generated from the original input file. Increasing the number of blocks generally improves load balance but might also lead to higher synchronization overhead. The configuration file also indicates the path to the input files folders, the command of the aligner executable, and the flags needed by this aligner. An explanation of all the configuration fields is included in the reference manual available with *parSRA*.

3.1. Splitting Input Files with FUSE

The procedure of *parSRA* starts with all UPC++ processes splitting the input file (or two input files in the case of paired-end alignment) using *FUSE*. The *FUSE* kernel module is shipped with every major Linux distribution and allows for the implementation of custom file systems in userspace without root permissions.

The *parSRA* tool provides a virtual file system that computes valid split points of *fasta* and *fastq* files using breadth-first search on a binary tree. Assuming a maximum number of $n = 2^k > 0$ splits for a file of $L > 0$ bytes, the input file is locally read from position $\frac{L}{2}$ until a valid delimiter is found at position $s \geq \frac{L}{2}$. Since NGS reads usually contain about a hundred base pairs, a delimiter is expected to be found in the direct neighbourhood. The resulting intervals $[0, s)$ and $[s, L)$ are then processed recursively for each of the remaining $k - 1$ levels of the binary tree in order to build an index.

As the input file is only read locally at the split positions the I/O is limited to merely $\mathcal{O}(n)$ accesses to the physical file system. Afterwards, n virtual files are exposed to the aligners that can be read in a lazy fashion *i.e.,* the content of the file is not accessed until the aligner reads it. Thus, using FUSE, we can completely eliminate the L write operations to the physical file system when performing a naïve splitting and further reduce the $2 \cdot L$ read operations (read input file + read split files) to merely $L+\mathcal{O}(n)$ accesses. Concluding, in contrast to a naïve splitting, approximately $2 \cdot L$ of the $3 \cdot L$ overall accesses to the

file system can be saved with our approach.

3.2. On-Demand Distribution of FUSE Files

Once the *FUSE* files with the blocks of reads are created, the processes align the reads of the different virtual files using an on-demand distribution. Each process i starts calling the external aligner using the virtual file i as input. Once the aligner has finished this work, the process looks for the next virtual file j that has not been computed yet by any process, and calls the external aligner using this file j as input. Note that every call to the aligner loads the reference genome index to the main memory of the corresponding node, but this time is almost negligible compared to the alignment runtime. The advantage of our on-demand distribution is that the workload adapts better to the characteristics of the input dataset than a static distribution as the one implemented by *pMap*. A process can compute more virtual files if the reads that they contain are aligned fast.

The implementation of the presented on-demand distribution using a traditional message passing approach like MPI would force us to broadcast the number of the next *FUSE* file to align every time that one process finishes one call to the underlying aligner, with the associated synchronization among all processes. Therefore, as previously mentioned, we have employed UPC++ to develop this on-demand distribution. The execution model of UPC++ is single program multiple data (SPMD). It takes advantage of C++ language features, such as templates, object-oriented design, operator overloading, and lambda functions (in C++ 11) to provide advanced PGAS features.

As all PGAS languages, UPC++ exposes a global shared address space to the user which is logically divided among processes, so each process is associated or presents affinity to a different part of the shared memory. Moreover, UPC++ also provides a private memory space per process for local computations, as shown in Figure 2. Therefore, each process has access to both its private memory

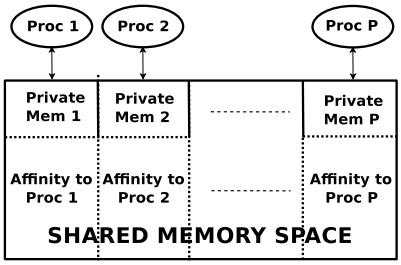

Figure 2: UPC++ memory model

and to the whole global memory space (even the parts that do not present affinity to it) with read/write functions. This memory specification combines the advantages of both shared and distributed programming models. On the one hand, the global shared memory space facilitates the development of parallel codes, allowing for all processes to directly read and write remote data without explicitly notifying the owner. On the other hand, performance can be increased by taking data affinity into account. Typically, accesses to remote data are more expensive than the accesses to local data (*i.e.,* accesses to private memory and to shared memory with affinity to the process).

A UPC++ shared variable is used to keep trace of the last file that has been computed. Concurrent accesses to this shared variable are synchronized with locks in order to avoid race conditions (*i.e.,* several processes working with the same virtual file). The basic concept of a lock is that only one process can own it any given time. Therefore, even if several processes try to access the lock only one will succeed. No other process can access that lock until the owning process unlocks it. One process closes the lock every time that it needs to know the value of the shared variable, *i.e.,* other processes will have to wait if they try to access it. The process that reads the value also increments it and opens the lock to make the variable accessible again to other processes.

Table 1: Characteristics of the Illumina datasets used in the tests.

Name	`SRR091634`	`SRR926245`
Read length	100	150
Number of reads	328,621,238	246,839,706
Type of alignment	Paired-end	Single-end

4. Experimental Results

16 nodes of the MOGON cluster, installed at the Johannes Gutenberg University-Mainz, are used for evaluating the scalability of *parSRA*. Each node contains four 16-core AMD Opteron 6272 processors (*i.e.,* 64 cores at 2.10 GHz within each node). A private L1 cache of 16 KB is available for each core, while the 2 MB L2 and 8 MB L3 caches are shared among two and eight cores, respectively. Nodes provide 256 GB of memory and are connected through a QDR InfiniBand network. UPC++ runs over GCC v4.8.1 and OpenMPI v1.6.5. We compare the scalability of *parSRA* and *pMap* aligning two Illumina short-read datasets (see Table 1) to the human genome `hg38` working with two popular aligners: *BWA-MEM* [5] (*BWA* v 0.7.11) and *Bowtie2* [2] (v 2.2.4). In order to illustrate different situations, one of the datasets is mapped with single-end alignment while the other with paired-end. Both datasets are publicly available and named after their accession numbers in the NCBI sequence read archive. Note that the current version of *pMap* only provides support for older versions of *BWA* and *Bowtie*. We have therefore adapted the *pMap* source code to work with the newer versions. We have also tested *BigBWA* [11] on the same system as an example of Hadoop-based SRA aligner.

Figure 3 shows the runtime (in minutes) for the alignment of both datasets with *parSRA*, *pMap* and *BigBWA*. As *BWA-MEM* and *Bowtie* already provide support for multithreaded parallelism, the experiments for *parSRA* and *pMap* are carried out with one process per node and 64 threads within each node. In fact, the time shown for one node is the time of the original multithreaded aligner, without including the overhead of *parSRA* or *pMap*. However, *BigBWA* could only use 27 cores per node due to high memory requirements of the Hadoop-system, which leads to higher per-node runtimes.

The results show that *parSRA* is faster than *pMap* in all cases, and the difference becomes more significant when increasing the number of nodes. Table 2 shows the speedups of both frameworks over the original multithreaded *BWA-MEM* and *Bowtie2* using up to 16 nodes. The scalability of *parSRA* is significantly higher than *pMap*. For instance, the average speedup of *parSRA* for 16 nodes (1,024 cores) is 10.48, which is 2.30 times higher than that of *pMap* (average speedup of 4.55 for 16 nodes). This trend indicates that speedups would become even higher when increasing the number of processes, *i.e.,* in systems with more nodes or less cores per node. Regarding *BigBWA*, we can assert that it has several drawbacks compared to *parSRA*. First, due to high memory requirements, *BigBWA* can only exploit 27 cores per node on our system (256 GB RAM). Moreover, the scalability of the Hadoop-based tool is limited. The main reasons are the expensive initial split of the input files (around 12 and 40 minutes for the single and paired datasets, respectively) and the reduce phase.

Figure 4 illustrates the breakdown of the runtime for the experiments with 16 nodes, in order to help us to understand the reasons why *parSRA* is faster than *pMap*. First, the initial file splitting in *pMap* is around 6 and 10 minutes for the `SRR926245` and `SRR091634` datasets, respectively. This time is almost constant for different number of processes, so its impact on the total time is more significant when the computational time is reduced. The time for splitting files through *FUSE* is almost negligible. Furthermore, gathering of the results into a unique output file is much faster in *parSRA*, thanks to using OS commands. In *pMap* this final step is performed in parallel by all processes, using the C `printf` function to print their results into a unique output file. However, this parallelization is inefficient due to two reasons: 1) `printf` is an ex-

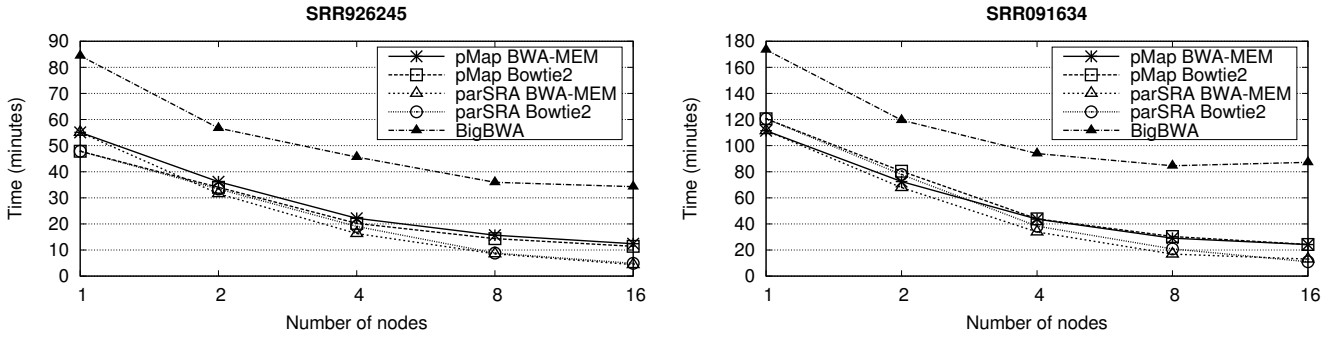

Figure 3: Runtime (minutes) of the alignment varying the number of nodes for *pMap* and *parSRA* using *BWA-MEM* and *Bowtie2* as underlying aligners, as well as *BigBWA*.

Table 2: Speedups of *parSRA*, *pMap* and *BigBWA*. Multithreaded *BWA-MEM* and *Bowtie2* aligners with 64 threads are used as baseline for *parSRA* and *pMap*. Speedups of *BigBWA* are obtained over the highest number of cores that can be employed on one node (27).

	Single-end SRR926245					Paired-end SRR091634				
Nodes ↓	*BWA-MEM*		*Bowtie2*		*BigBWA*	*BWA-MEM*		*Bowtie2*		*BigBWA*
	pMap	*parSRA*	*pMap*	*parSRA*		*pMap*	*parSRA*	*pMap*	*parSRA*	
2	1.52	1.74	1.41	1.43	1.49	1.54	1.64	1.50	1.55	1.45
4	2.49	3.38	2.37	2.51	1.85	2.55	3.29	2.75	3.15	1.85
8	3.51	6.43	3.34	5.42	2.35	3.84	6.60	3.97	5.79	2.05
16	4.43	12.57	4.19	9.81	2.46	4.61	8.56	4.97	10.98	1.99

pensive function; 2) the synchronization overhead needed to guarantee that only one process accesses the output file at a time. The main drawback of *parSRA* is that accessing the virtual files generated by *FUSE* has also an associated overhead. For instance, in contrast to *pMap*, the overhead of the virtual file system causes an increasing of the runtime of the alignment calls using *parSRA* (see Figure 4). Nevertheless, thanks to the adaptable workload distribution, the time difference between the alignment parts is smaller than the time difference when splitting the input and gathering the output.

In this section we have not included results for pBWA [7] as it is not portable (it must use version 0.5.9 of BWA as underlying aligner). Nevertheless, according to the experimental results presented in [7], this tool never obtains parallel efficiency higher than 24% even for a small cluster with 240 cores. Thus, we can deduce that its scalability is supposed to be significantly lower than the results presented in Table 2.

Finally, note that we have also tested that the output alignments provided by *parSRA* are the same as for the original tools. The only difference is that the order of the alignments in the output file can be different, which does not influence alignment quality.

5. Conclusion

NGS technologies have revolutionized biological research in recent years. The alignment of produced reads to a given reference genome is an important basic operation in many bioinformatics pipelines such as variant calling [36]. Even though several multi-threaded alignment tools (such as *BWA* [5] or *Bowtie2* [2]) have been developed their associated runtime for large-scale input data sets are still high.

In this paper we have addressed this runtime bottleneck

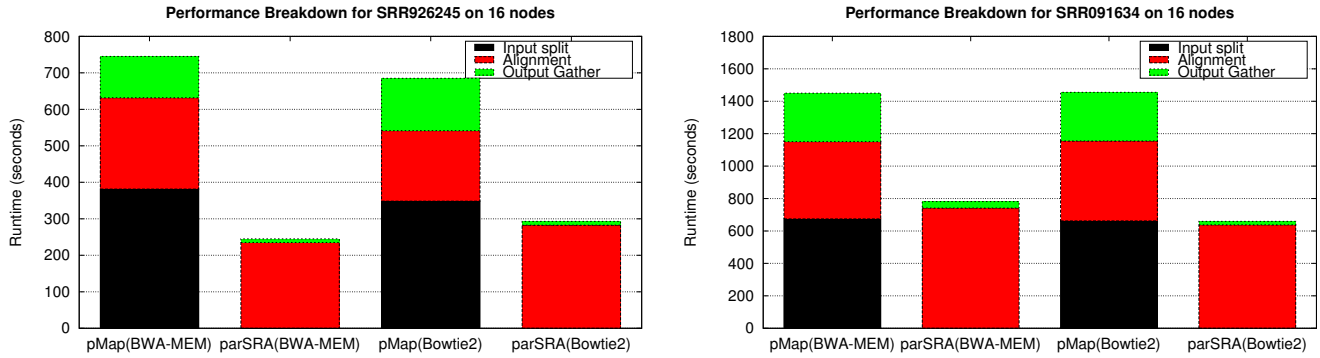

Figure 4: Breakdown of *pMap* and *parSRA* using *BWA-MEM* and *Bowtie2* as underlying aligners on 16 nodes.

by presenting *parSRA*, a framework for executing existing SRA tools on a compute cluster. *parSRA* gains efficiency by using (1) fast splitting of input reads using the FUSE kernel module; (2) balanced workload distribution based on shared lock in UPC++; and (3) gathering of results of results into a unique output file using OS commands.

Our performance evaluation using 16 compute nodes demonstrates speedups of 12.57 for executing *BWA-MEM* over *parSRA* for the single-end `SRR926246` dataset and 10.98 for executing *Bowtie2* for the paired-end `SRR091634` dataset. This is a factor of 2.8 and 2.2 times faster, respectively, compared to the *pMap* tool [10], which is based on a naive approach. *parSRA* is also faster and more scalable than the Hadoop-based approach *BigBWA*.

Source code of *parSRA* in C++ and UPC++ running on Linux systems with support for *FUSE* is available at https://sourceforge.net/projects/parsra/. The available version provides examples of configuration files for the aligners that have been tested: *Bowtie2* [2], *BWA-MEM* [5], *CUSHAW3* [6] and *SOAP2* [37].

Our current implementation of *parSRA* supports input files in the commonly used *fasta* or *fastq* formats. In order to save space input files are also often stored in compressed formats. Part of our future work is an extension of the first stage of *parSRA* to support direct reading of short read input files in a compressed format using *FUSE*. This would require an indexed compression format with a block structure which is supported by the *gzip* compliant *BGZF* (a valid *gzip* file) compression format part of the popular *SAMtools* [38].

Acknowledgments

This work was partly supported by the Ministry of Economy and Competitiviness of Spain and FEDER funds of the EU (Project TIN2013-42148-P).

References

[1] G. Rizk, D. Lavenier, GASSST: Global Alignment Short Sequence Search Tool, Bioinformatics 26 (20) (2010) 2534–2540.

[2] B. Langmead, S. L. Salzberg, Fast Gapped-Read Alignment with Bowtie2, Nature Methods 9 (4) (2012) 357–359.

[3] S. Marco-Sola, M. Sammeth, R. Guigó, P. Ribeca, The GEM Mapper: Fast, Accurate and Versatile Alignment by Filtration, Nature Methods 9 (2012) 1185–1188.

[4] J. C. Mu, H. Jiang, A. Kiani, M. Mohiyuddin, N. B. Asadi, W. H. Wong, Fast and Accurate Read Alignment for Resequencing, Bioinformatics 28 (18) (2012) 2366–2373.

[5] H. Li, Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM, arXiv:1303.3997 [q-bio.GN].

[6] Y. Liu, B. Popp, B. Schmidt, CUSHAW3: Sensitive and Accurate Base-Space and Color-Space Short-Read Alignment with Hybrid Seeding, PLOS ONE 9 (1).

[7] D. Peters, X. Luo, K. Qiu, P. Liang, Speeding Up Large-Scale Next Generation Sequencing Data Analysis with pBWA, Journal of Applied Bioinformatics & Computational Biology 1 (1).

[8] E. Georganas, A. Buluc, J. Chapman, L. Oliker, D. Rokhsar, K. Yelick, merAligner: A Fully Parallel Sequence Aligner, in: Proc. 29th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'15), Hyderabad, India, 2015.

[9] H. Li, R. Durbin, Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform, Bioinformatics 25 (14) (2009) 1754–1760.

[10] HPC Lab, pMap: Parallel Sequence Mapping Tool-http://bmi.osu.edu/hpc/software/pmap/pmap.html.

[11] J. M. Abuín, J. C. Pichel, T. F. Pena, J. Amigo, BigBWA: Approaching the BurrowsWheeler Aligner to Big Data Technologies, Bioinformatics 31 (24) (2015) 4003–4005.

[12] L. Pireddu, S. Leo, G. Zanetti, SEAL: a distributed short read mapping and duplicate removal tool, Bioinformatics 27 (15) (2011) 2159–2160.
URL http://dx.doi.org/10.1093/bioinformatics/btr325

[13] J. M. Abuín, J. C. Pichel, T. F. Pena, J. Amigo, SparkBWA: Speeding Up the Alignment of High-Throughput DNA Sequencing Data, PLOS ONE In press.

[14] Y. Chen, B. Schmidt, D. L. Maskell, CUSHAW: a CUDA Compatible Short Read Aligner to Large Genomes Based on the Burrows-Wheeler Transform, Bioinformatics 28 (14) (2012) 1830–1837.

[15] Y. Liu, B. Schmidt, CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing", IEEE Design & Test of Computers 31 (1) (2014) 31–39.

[16] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, B. Y. Lam, BarraCUDA - a Fast Short Read Sequence Aligner Using Graphics Processing Units, BMC Research Notes 5 (27).

[17] C. Liu, T. Wong, E. Wu, R. Luo, S. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, T. Lam, SOAP3: Ultra-Fast GPU-Based Parallel Alignment Tool for Short Reads, Bioinformatics 28 (6) (2012) 878–879.

[18] R. Luo, T. Wong, J. Zhu, C. Liu, X. Zhu, E. Wu, L. Lee, H. Lin, W. Zhu, D. W. Cheung, H. Ting, S. Yiu, S. Peng, C. Yu, Y. Li, R. Li, T. Lam, SOAP3-dp: Fast, Accurate and Sensitive GPU-Based Short Read Aligner, PLOS ONE 8 (5).

[19] NVIDIA CUDA Zone: nvBio, https://developer.nvidia.com/nvbio.

[20] Y. Chen, B. Schmidt, D. L. Maskell, A Hybrid Short Read Mapping Accelerator, BMC Bioinformatics 14 (67).

[21] R. Luo, J. Cheung, E. Wu, H. Wang, S.-H. Chan, W.-C. Law, G. He, C. Yu, C.-M. Liu, D. Zhou, Y. Li, R. Li, J. Wang, X. Zhu, S. Peng, T.-W. Lam, MICA: A Fast Short-Read Aligner that takes Full Advantage of Many Integrated Core Architecture (MIC), BMC Bioinformatics 16 (Suppl 7).

[22] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, M. Wilde, Swift: Fast, Reliable, Loosely Coupled Parallel Computation, in: Proc. 3rd IEEE World Congress on Services (SERVICES'07), Salt lake City, UT, USA, 2007.

[23] D. De Oliveira, E. Ogasawara, F. Baião, M. Mattoso, Scicumulus: A Lightweight Cloud Middleware to Explore Many Task Computing Paradigm in Scientific Workflows, in: Proc. 3rd IEEE Intl. Conf. on Cloud Computing (CLOUD'10), Miami, FL, USA, 2010.

[24] FUSE, Filesystem in Userspacehttp://fuse.sourceforge.net/.

[25] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, K. Yelick, UPC++: a PGAS Extension for C++, in: Proc. 28th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'14), Phoenix, AR, USA, 2014.

[26] UPC Consortium, UPC Language Specifications, v1.2, http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf.

[27] R. W. Numrich, J. Reid, Co-Array Fortran for Parallel Programming, ACM FORTRAN FORUM 17 (2) (1998) 1–31.

[28] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken, Titanium: A High-Performance Java Dialect, Concurrency: Practice and Experience 10 (11) (1998) 825–836.

[29] J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, D. A. Mallón, B. Wibecan, UPCBLAS: a Library for Parallel Matrix Computations in Unified Parallel C, Concurrency and Computation: Practice and Experience 24 (14) (2012) 1645–1667.

[30] J. González-Domínguez, O. A. Marques, M. J. Martín, G. L. Taboada, J. Touriño, Design and Performance Issues of Cholesky and LU Solvers Using UPCBLAS, in: Proc. 10th IEEE Intl. Symp. on Parallel and Distributed Processing with Applications (ISPA'12), Leganés, Spain, 2012, pp. 40–47.

[31] C. Bell, D. Bonachaea, R. Nishtala, K. Yelick, Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap, in: Proc. 20th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'06), Rhodes Island, Greece, 2006.

[32] R. Nishtala, P. Hargrove, D. Bonachea, K. Yelick, Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap, in: Proc. 23rd IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'09), Rome, Italy, 2009.

[33] R. Nishtala, Y. Zheng, P. Hargrove, K. Yelick, Tuning Collective Communication for Partitioned Global Address Space Programming Models, Parallel Computing 37 (9) (2011) 576–591.

[34] E. Georganas, A. Buluc, J. Chapman, L. Oliker, D. Rokhsar, K. Yelick, Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly, in: 26th ACM/IEEE Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC'14), New Orleans, LA, USA, 2014.

[35] J. C. Kässens, J. González-Domínguez, L. Wienbrandt, B. Schmidt, UPC++ for Bioinformatics: A Case Study Using Genome-Wide Association Studies, in: Proc. 15th IEEE Intl.

Conf. on Cluster Computing (Cluster'14), Madrid, Spain, 2014.

[36] G. A. V. der Auwera, M. O. Carneiro, C. Hartl, R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, E. Banks, K. V. Garimella, D. Altshuler, S. Gabriel, M. A. DePristo, From FastQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline, Current Protocols in Bioinformatics 11 (11).

[37] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansend, J. Wang, SOAP2: an Improved Ultrafast Tool for Short Read Alignment, Bioinformatics 25 (15) (2009) 1966–1967.

[38] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, The Sequence Alignment Map Format and SAMtools, Bioinformatics 25 (16) 2078–2079.

Jorge González-Domínguez received the B.Sc., M.Sc. and PhD degrees in Computer Science from the University of A Coruña, Spain, in 2008, 2010 and 2013, respectively. He is currently an assistant teacher in the Computer Architecture Group at the University of A Coruña, Spain. His main research interests are in the areas of high performance computing for bioinformatics and PGAS programming languages.

Christian Hundt has received his diploma in theoretical physics for the analysis of quantization maps on curved manifolds and a PhD degree in Computer Science for the efficient subsequence alignment of time series on CUDA-enabled accelerators at the University of Mainz, Germany, in 2010 and 2015. In his current position, as a postdoctoral researcher at the Parallel and Distributed Architectures group, he investigates the design and parallelization of algorithms in the field of bioinformatics.

Bertil Schmidt (M'04-SM'07) is tenured Full Professor and Chair for Parallel and Distributed Architectures at the University of Mainz, Germany. Prior to that he was a faculty member at Nanyang Technological University (Singapore) and at University of New South Wales (UNSW). His research group has designed a variety of algorithms and tools for Bioinformatics mainly focusing on the analysis of large-scale sequence and read datasets. For his research work, he has received a GPU Research Center award, GPU Education Center Award, CUDA Academic Partnership award, CUDA Professor Partnership award and Best Paper Awards at IEEE ASAP 2015 and IEEE ASAP 2009.