Genomics 111 (2019) 1641–1650

Contents lists available at ScienceDirect

Genomics

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

Original Article

ParStream-seq: An improved method of handling next generation sequence data

Sudip Mondal[a], Ranjan Kumar Maji[b], Zhumur Ghosh[b], Sunirmal Khatua[a,*]

[a] Department of Computer Science and Engineering, University of Calcutta, Kolkata, India
[b] Bioinformatics Center, Bose Institute, Kolkata, India

ARTICLE INFO

Keywords:
Biological big data
Alignment
Streaming
Parallel computing
HDFS
NGS

ABSTRACT

The exponential growth of next generation sequencing (NGS) data has put forward the challenge for its storage as well as its efficient and faster analysis. Storing the entire amount of data for a particular experiment and its alignment to the reference genome is an essential step for any quantitative analysis of NGS data. Here, we introduce streaming access technique '*ParStream-seq*' that splits the bulk sequence data, accessed from a remote repository into short manageable packets followed by executing their alignment process in parallel in each of the compute core. The optimal packet size with fixed number of reads is determined in the stream that maximizes system utilization. Result shows a reduction in the execution time and improvement in the memory footprint. Overall, this streaming access technique provides means to overcome the hurdle of storing the entire volume of sequence data corresponding to a particular experiment, prior to its analysis.

1. Introduction

The amount of biological sequence data in public repositories has doubled every 6–8 months (on an average) from 2007 to 2016 [1]. It is estimated that by 2025, the storage of human genome alone will require 2–40 exabytes [2], which requires the special technique to handle such huge data. Next-generation sequencing (NGS) technologies have deeply changed our mode of analyzing biological data as it delivers an overwhelming amount of data with greater affordability. Hence, it is changing the biological landscape and is flooding the databases with massive amounts of raw sequence data. NGS data analysis involves intensive computation, in-memory computing, vectorization, bulk data transfer, CPU frequency scaling [3].

Sequence alignment is the most fundamental step for any NGS data analysis. Sequence alignment tools, such as BWA [4], SOAP [5], *Bowtie2* [6], and Maq [7] were developed to efficiently align short DNA/RNA sequences. *Bowtie* is an ultrafast and memory-efficient tool for aligning sequence reads and is more sensitive than BWA. Sequence alignment tools based on parallel execution are ParAlign [8], merAligner [9], Novoalign [10], CUDA ClustalW [11] and mrsFAST-Ultra [12] ParAlign supports Parallel processing capabilities in the form of the single instruction, multiple data (SIMD) technology. MerAligner relies on a high performance distributed hash table. CUDA ClustalW is a GPU version of ClustalW [13] and mrsFAST-Ultra is a short read mapper that optimizes cache usage to get higher performance using multi-threading.

Several sequence aligners such as CloudBurst [14], CloudAligner [15], BlastReduce [16], SparkBWA [17] have been developed using Hadoop MapReduce [18] big data technology [19]. CloudBurst and CloudAligner are effective for short reads. Crossbow, a cloud computing tool [20] identifies Single Nucleotide Polymorphisms (SNPs) from short read sequencing data using bowtie for alignment and hadoop cluster for computing. The aforementioned tools load the full volume of data at a time to process, which in turn increases the memory footprint. They demand extra storage as well as computing prowess for analysis.

The active development of alignment algorithms creates bottleneck even with the rapidly increasing throughput of sequencing machines [21]. Data streaming is an ideal procedure for processing large amount of data in near real time with minimum configuration [22,23]. This can be approached using a fast and efficient streaming access alignment pipeline [24] through its implementation in a distributed architecture. Hence the requirement for high level computation using machines with high system configuration for sequence data analysis can be mitigated. This motivated us to design and implement *ParStream-seq*, the sequence streaming tool for efficient access of NGS data.

* Corresponding author.
E-mail address: skhatuacomp@caluniv.ac.in (S. Khatua).

https://doi.org/10.1016/j.ygeno.2018.11.014
Received 31 August 2018; Received in revised form 11 November 2018; Accepted 12 November 2018
Available online 15 November 2018

Table 1
Input dataset.

Accession ID	Run ID	Sequencer	Read length	Total number of reads
ERX2269478	ERR2214619	Illumina MiSeq	85 bp	119,677,407
SRX026839	SRR094773	Illumina genome analyzer II	42 bp	24,818,985
SRX3459079	SRR6363052	NextSeq 500	~24 bp	9,901,573

2. Materials and methods

2.1. Input dataset

We have taken three datasets from SRA ("https://www.ncbi.nlm.nih.gov/sra") (Table 1) having read of different lengths. For the reference sequence, we considered human chromosomal sequence (hg19) downloaded from NCBI for chromosomes 1(248.9 Mb), 2 (242.1 Mb), 3 (198.2 Mb) and 4 (190.2 Mb) as they have comparatively larger chromosomal size than the rest.

In order to optimize our protocol on query sequence data of different read lengths (i.e 85, 42 & ~24 bp), we extracted the minimum number of reads (i.e ~9.9 million reads) contained in each of the aforementioned sequence datasets.

2.2. Computational resources used

All experiments were executed on the standalone system with 8-core and 12-core CPUs, each with 2.6 GHz Intel *Xeon* (n series) processor, 16 GB of internal memory, JDK (Java SE Development Kit)1.8 and Hadoop 2.7.2.

2.3. Building index and alignment step

In order to reduce the execution time for index building and alignment which is an essential step for NGS data analysis, each of these two steps were executed in parallel. In this study, we have used popular short read aligner *bowtie2* [25] for sequence alignment and JDK1.8 for parallel execution. The entire procedure described here is provided in Fig. 1.

2.3.1. Building the index file

Java threads were used as wrapper [26] to build the reference indices. *Bowtie2* [8] indices were built for (a) a single *fasta* file containing the reference for all chromosomal sequences (b) multiple *fasta* files for each chromosome, in equal splits of 2, 4 and 8. The reference sequence was split so that the *bowtie2* [25] index could be built for each of the splits in parallel using Java threads. All the index (*.bt2*) files for the reference splits were built (*bowtie2-build*).

2.3.2. Alignment

Java threads were used as a wrapper to call *bowtie2*. It is also used to take control over the *bowtie2* threads to take maximum advantage of the parallel execution. All the index (*.bt2*) files for the reference splits (generated in the previous step) are used as the reference index in this alignment step.

Finally, the alignment process was executed for:

Case a: A single query sequence file against a single reference file comprising of all the chromosomes.
Case b: The entire query sequence of different read lengths (i.e 85, 42 and ~24 bp) against references sequence (i.e *2-splits*, *4-splits* and *8-splits*). Alignment with the query sequence file was performed by varying the *bowtie2* [8] threads (p = 2, 4, 8) as well.

2.4. Sequence access through streaming using ParStream-seq

Streaming is an ordered split of a large input sequence data. Thus in order to access the query sequence as a stream, it is important to split the query sequence file such that each read remains intact during the process of splitting. We have used micro-batch processing technique [27] that splits incoming sequence data into batches. This is based either on 'arrival time' or 'packet size' [28,29]. We have split the input query sequence data based on manageable 'packet size' so as to execute the alignment in parallel in order to minimize computation, memory usage, and storage. Based on this, we have developed a stream access pipeline named "*ParStream-seq*". HDFS adds to the advantage for access, storage, and processing of the huge sequence data on distributed framework [30]. Hence, we have setup Hadoop cluster so as to store the alignment results through the use of HDFS. *ParStream-seq,* accesses

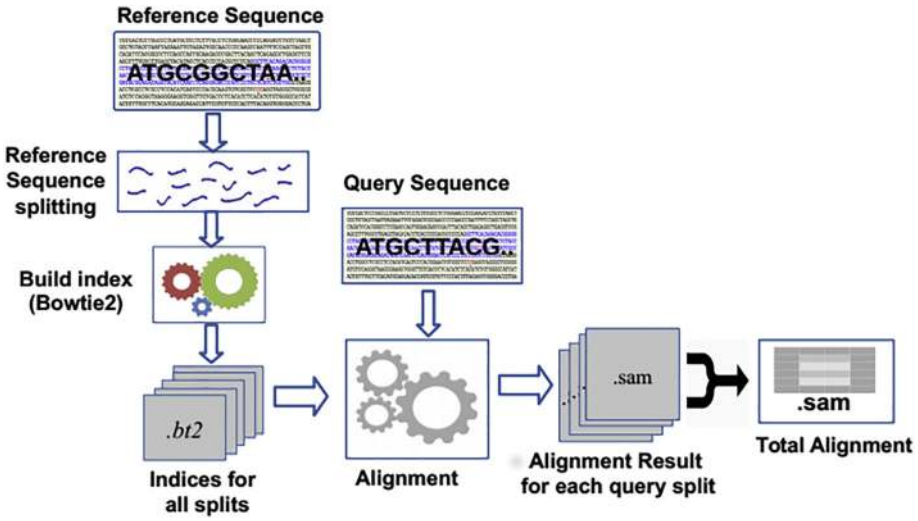

Fig. 1. Workflow for the execution of the alignment process using *bowtie2* and Java threads during index building for the reference sequence and alignment of query sequence against the split reference indices as build.

sequence data, optimizes data packet size and is compatible for storing the resultant data into HDFS for subsequent analysis. Finally, we merge the results to a single (.bam or .sam) file at the end of the entire experiment. Hence, while dealing with multiple-mapped reads, which is important while interpreting reads that map to repeat regions of the genome, the finally merged single file (.bam or .sam) contains all the information intact.

2.4.1. Pseudocode for ParStream-seq

The pseudocode of the overall steps for *ParStream-seq* is described below:

Algorithm 1: *ParStream-seq*

Input: *R: reference sequence;*
Qsrc: Query sequence source (local/remote/NCBI etc.);
S <- (s1, s2, ...sn): Set of query sequence from Qsrc;

Prerequisite: ***OptStreamSeq(****S,R)****:*** *calculate optimal data packet size(S) and reference sequence splits(R)*

1: **Procedure** *ParStream-Seq(Qsrc,R)*
2: n, p_size ← ***OptStreamSeq(****S,R)*** *# n: optimal number of reference split,* based on number of CPU cores, *p_size: optimal query packet size*

```
3: R ← split_reference(n)        # split reference sequence and assign to R vector
3: ref ← build_index(R)          # run parallel in each thread; ref is a vector
4:     while(p_size * i ≤ Qsize):        # where Qsize is the query sequence size
5:         S[i] ← stream(Qsrc, p_size)   # Si is the packet streams pulled by java
                                         stream()
6:         Sq[i] ← Collectors.toList(S[i])    # store each query packet
7:         Res[i] ← align(ref[i], Sq[i])      #alignment score
8:         Cache(Res[i])                      #store the alignment score
9:     End while
10:    for each i: GetAll(Res[i])         #store all processed results into HDFS
```

2.4.1.1. OptStream-seq() evaluating optimal data packet size and sequence splits. Evaluating the optimal packet size enables efficient thread handling and reduces execution time. The packet size is determined by the number of sequence reads in each data packet *(s1, s2, ...sn)*. The optimal packet volume is specific to the system configuration. We compute the optimal packet size by the method OptStream-seq() as described below:

We start with 64 reads in each packet. We then compared alignment result of each query packet separately by *bowtie2* and *ParStream-seq*. The steps are repeated with increasing packet size, i.e. incrementing the constituent read counts in multiples of 2. The execution time is compared by varying the size of each packet using *bowtie2* and *ParStream-seq*.

In order to determine the optimal sequence split and the optimal packet size, we define the variables for *OptStreamSeq*() (as shown in Fig. 2):

2.4.1.2. Computing OptStreamSeq. First, we consider the intersection of two lines L_m and L_b in 2-dimensional space, with line L_m being defined by two distinct points having coordinates (R_{mi}, T_{mi}) and (R_{mj}, T_{mj}), and line L_b being defined by two distinct coordinates $(R_{bi}, T_{bi})$and (R_{bj}, T_{bj}); i and j represent labels for two positions (along the X-axis) shown in Fig. 2.

The intersection point P_k having coordinates $(R_k, T_k)$of line L_m and L_b can be calculated using determinants for each of sequence splits, i.e. 2,4,8 etc.

The optimal packet size is therefore calculated by using:

$$p\_size = R_n \quad \text{such that } T_n = \min(T_k) \text{ where } k = 2, 4, 8$$

2.4.2. Preprocessing data coming from different sources for implementing ParStream-seq

The source of the query sequence for streaming access could be (a) local storage, (b) remote location provided by NCBI or (c) other remote locations provided by third party vendors (as shown in Fig. 3). Different preprocessing steps (given below) are needed for each of these datasets. Subsequently, ParStream-seq code is executed to access and align the pre-processed data.

Case a: In order to access sequence data from local storage, we have used *JDK 1.8 Stream API (java.util.stream.Stream)* [31] to process large query sequence data as a stream using two built-in methods; *stream()* [32] and *parallelStream()* [32] for obtaining sequential or parallel streams. The Stream method doesn't store data, it operates on the source data i.e. local storage. We have used the method *Collectors.toList()* for collecting the stream elements into a *list* instance and store it in *fasta* file format (.fa) so that it can be used for alignment process.

Case b: For accessing sequence data from remote data storage provided by NCBI, we used NCBI's SRA Toolkit [33] to access sequence data as a stream (as shown in Fig. 2). The toolkit allows to stream data from the NCBI servers for direct analysis. We disable local file caching of the total sequence file using *vdb-config* command. We then embedded *fastq-dump* (with -x and -z parameter) called by java *stream()*, returns a specific volume of data and apply *Collectors.toList()* method so as to fetch the output data (in fasta format) into batches. These data packets are then subsequently used for the alignment process.

Case c: For accessing data from remote locations provided by third party vendors, our pipeline enables handshaking via *ftp* (such as .gz compressed fasta and fastq file) and performs streaming access on demand. We have used *java.net.URL* [34] class to access data using *URL* via ftp. The Java *GZIPInputStream* [35] class (*java.util.zip.GZIPInputStream*) is used to decompress, access GZIP compressed files and store the stream data in packets.

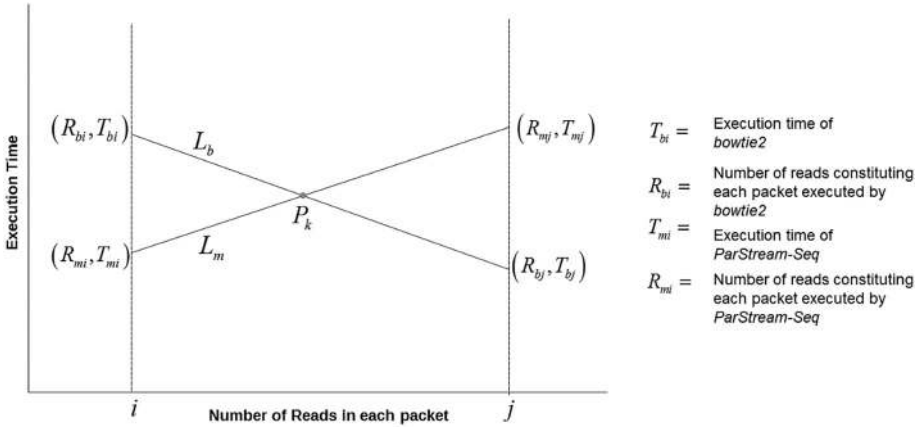

Fig. 2. Representation of the Variables for *OptStreamSeq()* method.

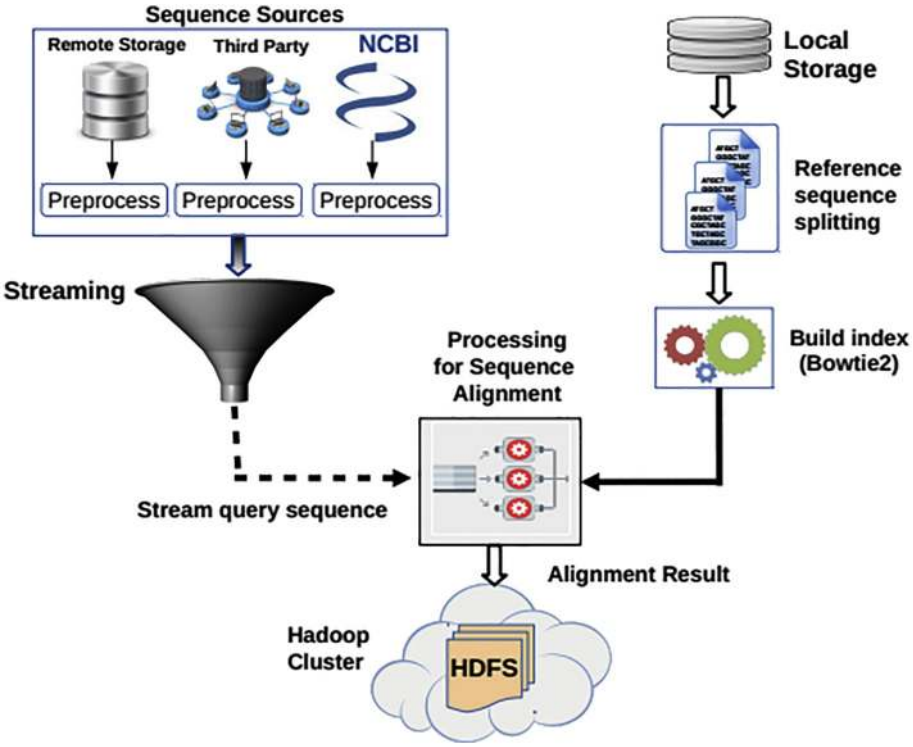

Fig. 3. Schematic diagram showing the workflow of *ParStream-seq*.

3. Results and discussion

3.1. Parallel execution of bowtie2

In our experiment, we considered chromosome numbers 1, 2, 3 and 4 to be denoted as rf1, rf2, rf3, and rf4 respectively, and the time for building indices for each reference file as tr1, tr2, tr3, and tr4 respectively. Fig. 4 shows the variation in the execution time of *bowtie2* for 3 query sequence files (containing ~9.9 million reads as described in Section 2.1) corresponding to read lengths ~24 bp, 42 bp, and 85 bp.

We have obtained five set of results which are as follows:

The first set of result (marked as (a) in Fig. 4), gives the total execution time to build the index for each chromosomal reference files (rf1, rf2, rf3, and rf4) executed sequentially one after another, followed by alignment of the 3 query sequence files against the separate index files.

The second set of result (marked as (b) in Fig. 4), refers to the execution time to build the index for a single reference file containing concatenated chromosomal reference sequences (i.e. Rf = rf1 + rf2 + rf3 + rf4), followed by the alignment of the 3 query sequence files against the single index file.

In the last three sets of the result shown in Fig. 4 (marked as (c), (d) and (e)), we have incorporated java *Thread()* method to allow parallel execution for building index which was not implemented in *bowtie2*. Thereafter, parallel execution of the alignment of the query files corresponding to ~24 bp, 42 bp, 85 bp read lengths was performed using each of the bowtie threads $p = 2$, $p = 4$ and $p = 8$. The execution time for alignment of each of the query files by varying the bowtie threads was almost the same. We found ~50%, ~49%, and ~47% improvement respectively for ~24 bp, 42 bp, 85 bp read lengths compared to the result obtained (a) for each of the bowtie threads (shown in (c), (d) and (e)).

Thus, java *Thread()* provides an advantage over *bowtie2* by parallelly executing the step for building indices (as shown in Fig. 1).

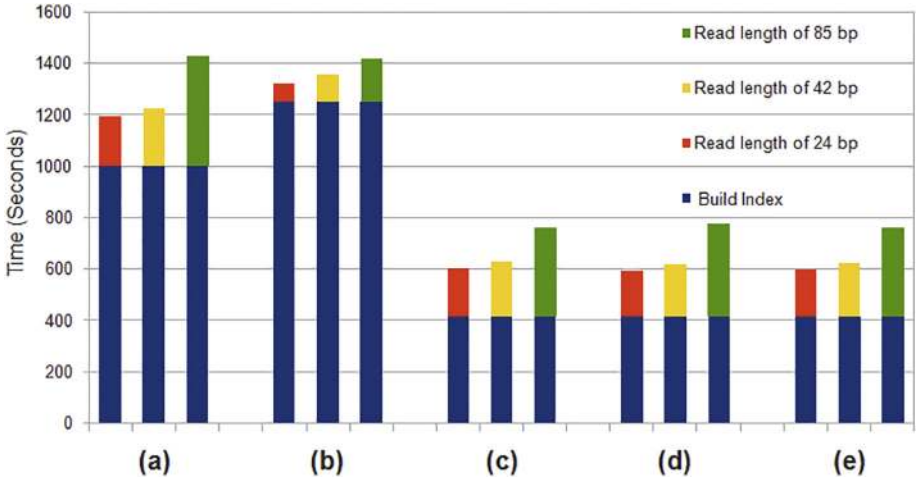

Fig. 4. Comparison of execution time for *bowtie2* and java *thread()* on query sequence file of different read lengths (~24 bp, 42 bp,85 bp). The lower region (blue color) of each bar denotes the execution time for building indices, while the top region denotes the time for alignment. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

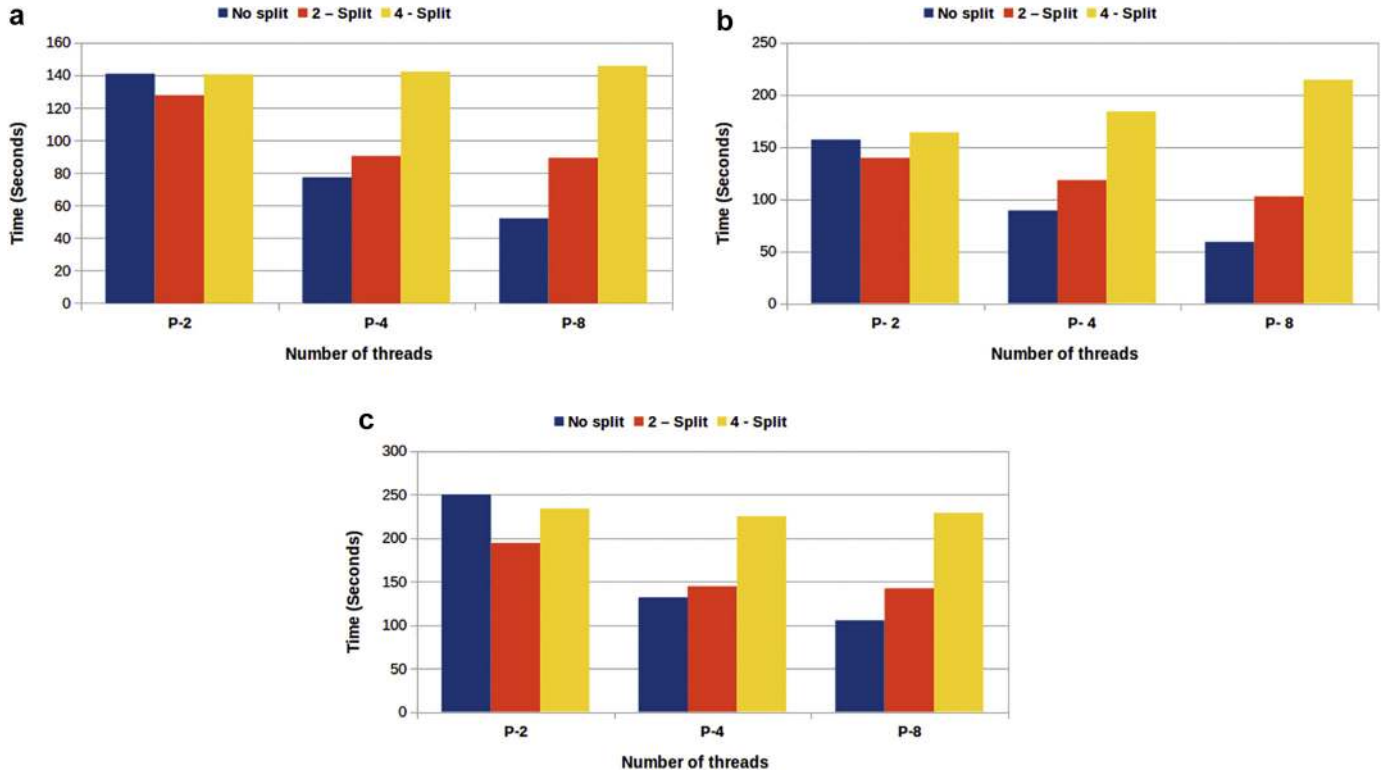

Fig. 5. Comparison of execution time with Reference sequence split against query sequence of fixed reads count ~9.9 million reads of read lengths (a) ~24 bases (b) 42 bases (c) 85 bases. Each of the grouped bars represents three versions of *bowtie2* threads execution (p = 2, 4, 8) using *2-split* and *4-split* reference files.

This gain in performance for building index corresponding to the reference file will be useful in a situation where a user needs to work with multispecies at a time because in such cases, there will be a need to build the indices separately for all the species simultaneously.

3.2. Parallel execution of bowtie2 by splitting the reference sequence

We have observed that parallel execution of *bowtie2* (as shown in (c), (d) and (e) of Fig. 4) reduces building and alignment time. We now split a single reference file, build the reference index for each split as well as perform the alignment of the query sequence file against each of the splits in parallel. Chromosome 1 has been selected as the reference file here, as it is the largest (248.9 Mb) among the 4 chromosomes. We then split the reference sequence file corresponding to chromosome 1 making *2- split* and *4-split*.

Fig. 5 shows the execution time for alignment of the 3 query sequence files (corresponding to ~24 bp, 42 bp and 85 bp read length), each of size ~9.9 million reads with the reference sequence file ('*no split*', '*2-split*' and '*4-split*' of the reference sequence file). Here we have considered only the alignment time but not the index building time as we have already calculated it previously as shown in Fig. 4 of Section 3.1.

Bowtie2 (p = 2, 4 and 8) threads have been used for aligning against '*no* split' reference sequence file. Further, Java *thread()* has been used for aligning against '*2-split*' and '*4-split*' reference sequence files. We observed that for all the 3 query files, the alignment execution time using *2-split* reference sequence file(using Java *thread()*) have better performance over *4-split* (using Java *thread()*) and '*no split*' reference sequence (using *bowtie2* thread (p = 2)).On the contrary, '*no split*' reference sequence using *bowtie2* thread (p = 4 and p = 8) performs better over *2-split* and *4-split* of reference sequence file(using Java *thread()*).

This is due to CPU thrashing caused by the over-usage of threads than the number of threads available in our system configuration. The use of limited reference sequence splits optimizes usage of system threads, whereas over-splitting causes thrashing. Thus, it is important to set limits to reference sequence splits based on system configuration. This gave us the clue that we can further optimize the consumption of system resource if we can process the query sequence file after splitting it into discrete packets [where the packet size is measured by the constituent number of reads] instead of processing the whole query file at a time.

3.3. Parallel execution of bowtie2 with query sequence streaming

We have observed in the previous section that the alignment process with reference sequence splits executed in parallel using java threads and *bowtie2* threads are beneficial to reduce the execution time. Based on this, we have used streaming access of query sequence packets along with usage of reference splits (of 2,4,8) for alignment execution in parallel.

3.3.1. Optimizing query sequence packet size

As java threads operate optimally on files of small size with lesser memory footprint [36,37], there is an exponential rise in execution time when the packet size contain reads exceeding the optimal number. In order to calculate the optimal read count in each packet stream, we use our method Opt*Stream-seq()* as described in Section 2.3.1.

In our experiment, the optimal read count and reference split have been calculated using method OptStream-seq(). Table 2 shows the optimal read counts obtained for the 3 query sequence files corresponding to ~24 bp, 42 bp and 85 bp read length using 8 core and 12 core machines. The detailed calculation of optimal read count and reference split using *OptStream-seq()* has been shown in Supplementary file 1.

Evaluation of the optimal data packet size and sequence split is required for the query sequence files of different read lengths. If the read length does not vary for multiple query sequence files, then this has to be calculated once.

Table 2
Optimal number of read counts and execution time corresponding to query sequences of ~24 bp, 42 bp and 85 bp.

Query sequence		Reference sequence splits	Query sequence	
Run IDs	Bases		8 - Core (Read count, execution time)	12 - Core (Read count, execution time)
ERR2214619	85 bp	2 split	5119, 0.42	5266, 0.59
		4 split	7246, 0.39	4778, 0.50
		8 split	6945, 0.49	3909, 0.50
SRR094773	42 bp	2 split	2047, 0.20	7372, 0.59
		4 split	2559, 0.22	6330, 0.50
		8 split	4351, 0.33	4608, 0.50
SRR6363052	~24 bp	2 split	10,532, 0.36	9216, 0.59
		4 split	8647, 0.32	7509, 0.51
		8 split	8714, 0.40	5558, 0.48

3.3.2. Execution time

In the previous section, we have calculated the optimal number of reads in each query sequence packet and an optimal number of reference sequence splits using *OptStream-seq()* (described in Section 2.3.1) and access sequence data from different sources. In this experiment, we have used two different system configuration (discussed in Section 2.2) to execute our method *ParStream-seq*. We reduced the execution time up to ~32% as shown in Fig. 6 in all the three samples of different read lengths of query sequence using *ParStream-seq* over *bowtie2* threads (p = 2, 4 and 8).

Hence we opt for streaming the query sequence with specific number of reads (as obtained from Table 2) to get a better performance with respect to execution time.

3.3.3. Memory utilization

Parallel execution with different length of query sequence packets and reference split indices reduces the system memory requirement as each thread needs the small memory to execute and as a whole, it consumes less for processing. We have executed our experiment in two different machine configurations (as mentioned in Section 2.2) and have averaged the memory utilization on executing all the three query sequence files (~24 bp, 42 bp, 85 bp) along with their optimal reference splits. We observed that memory footprint gets reduced by ~54% using

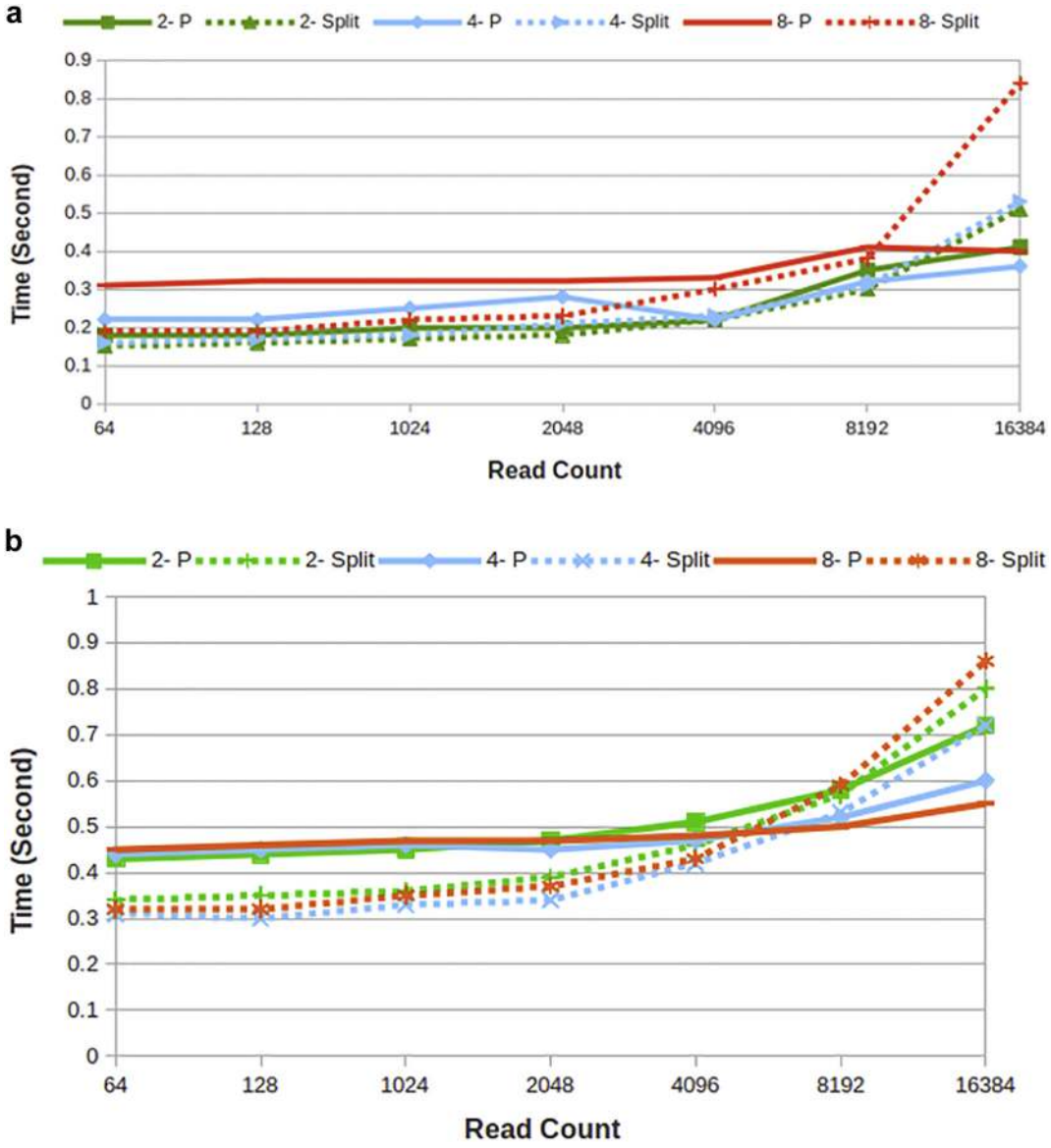

Fig. 6. Variation in execution time by batch streaming of the query with different number of reads in each batch for different read lengths (a) ~24 bases with 8 core (b) ~24 bases with 12 core (c) 42 bases with 8 core (d) 42 bases with 12 core (e) 85 bases with 8 core (f) 85 bases with 12 core system configuration.

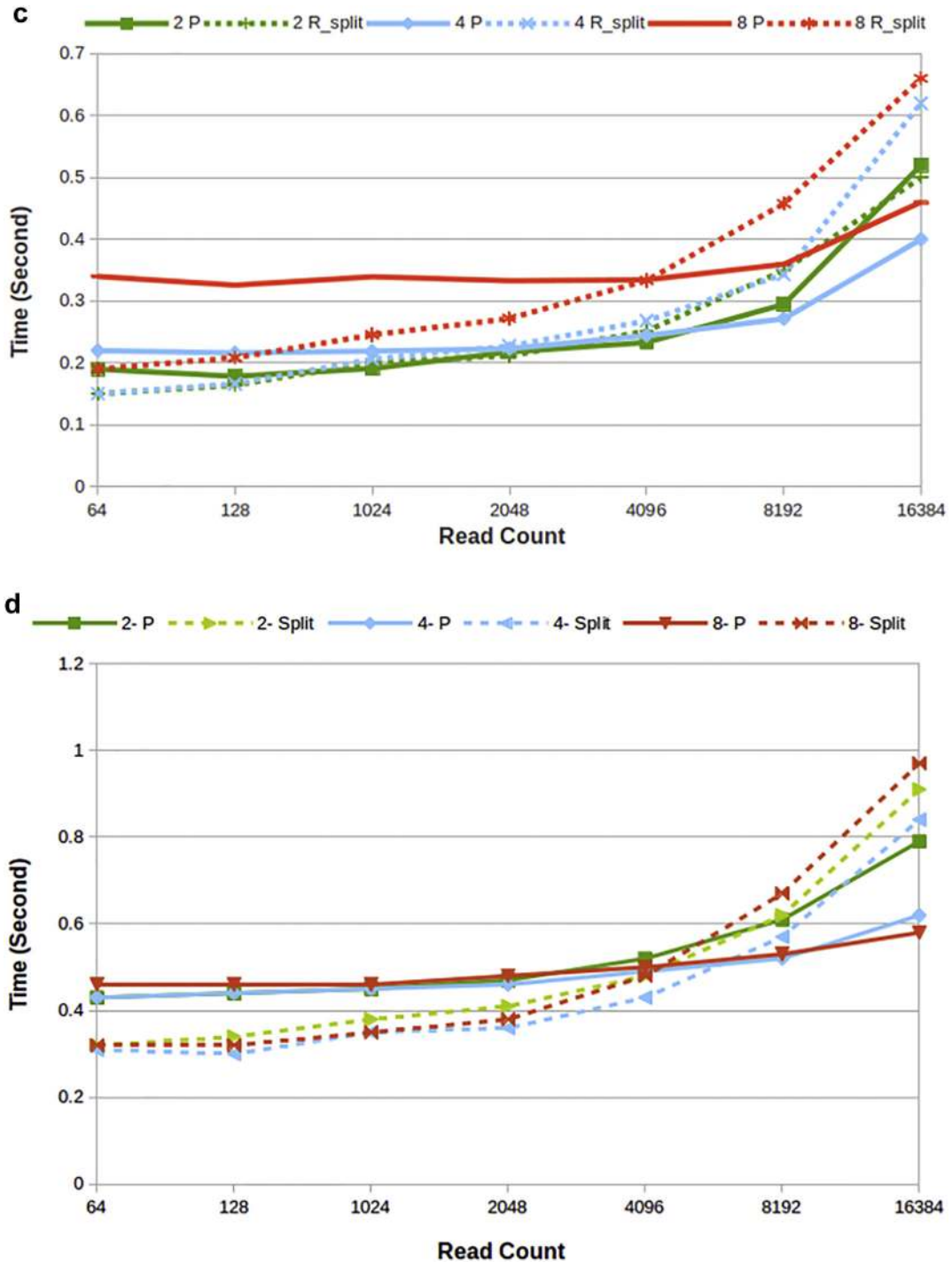

Fig. 6. (*continued*)

ParStream-seq (using *2-split*, *4-split* and *8-split* reference files) compared to the *bowtie2* (using *no split* reference file) as shown in Fig. 7 (*for 8 core and 12 core* machine*)*.

4. Conclusion

In this paper, we have introduced a streaming access technique *ParStream-seq* which can provide means to overcome the difficulty of storing the entire volume of sequence data corresponding to a particular experiment prior to analysis. This streaming access protocol enables retrieval of biological sequence data from local storage, remote location provided by NCBI and remote location provided by other third-party vendors for alignment. Here, we have successfully performed our analysis on three samples (~9.9 M reads per samples) containing sequences of varying read lengths (85, 42, and ~24) bp. We have added (i) Java threads for parallel execution and have (ii) computed optimal number of reads in each streamed packet for (iii) optimal number of sequence splits for parallelism. Accessing query sequences as a stream followed by sequence alignment using parallel execution and storage in a distributed framework (HDFS) reduces the execution time as well as optimizes memory utilization. Our algorithm also works perfectly for query sequences of various read lengths. As the number of CPU threads increases, computation time decreases.

As a whole, the key contribution of *ParStream-seq* is to access large

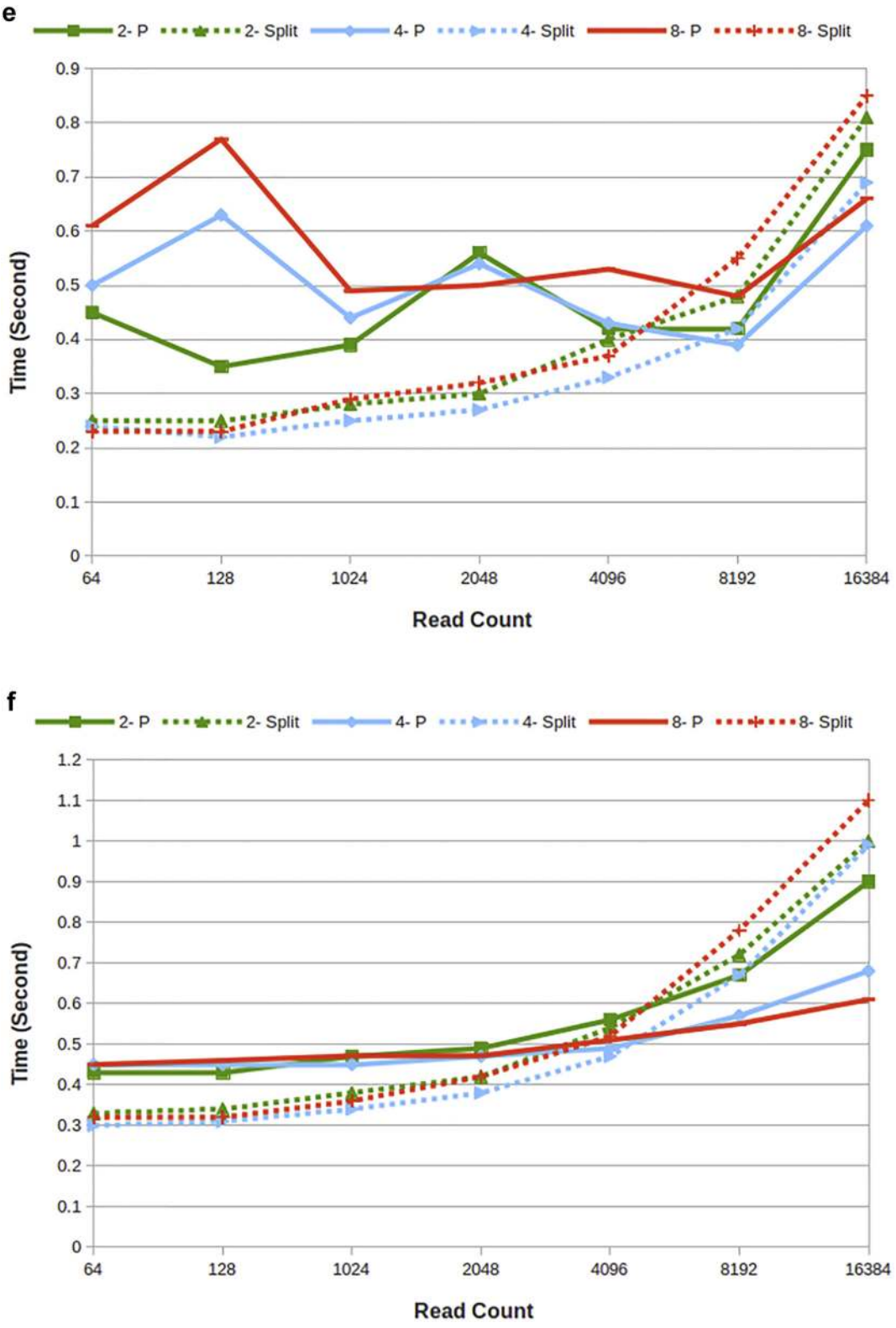

Fig. 6. (*continued*)

volume of query data (Big data) in a streaming mode and perform the alignment process in parallel. To evaluate the performance of ParStream-seq we have compared its results with the most popular aligners *bowtie2*. We have shown that *bowtie2* index building and alignment execution takes more time compared to that of *ParStream-seq*. Further, availability of tools that can access large volume of query data efficiently in the form of discrete packets (which will include calculation of packet size and subsequent reduction in the alignment execution time) will facilitate the performance evaluation of *ParStream-seq* in a more robust manner.

In future, we need to modify the existing algorithms (for downstream analysis of sequence data) such that they become compatible with *ParStream-seq* and can accept streamed input sequence generated by *ParStream-seq*. Overall, we hope that this new technique will resolve

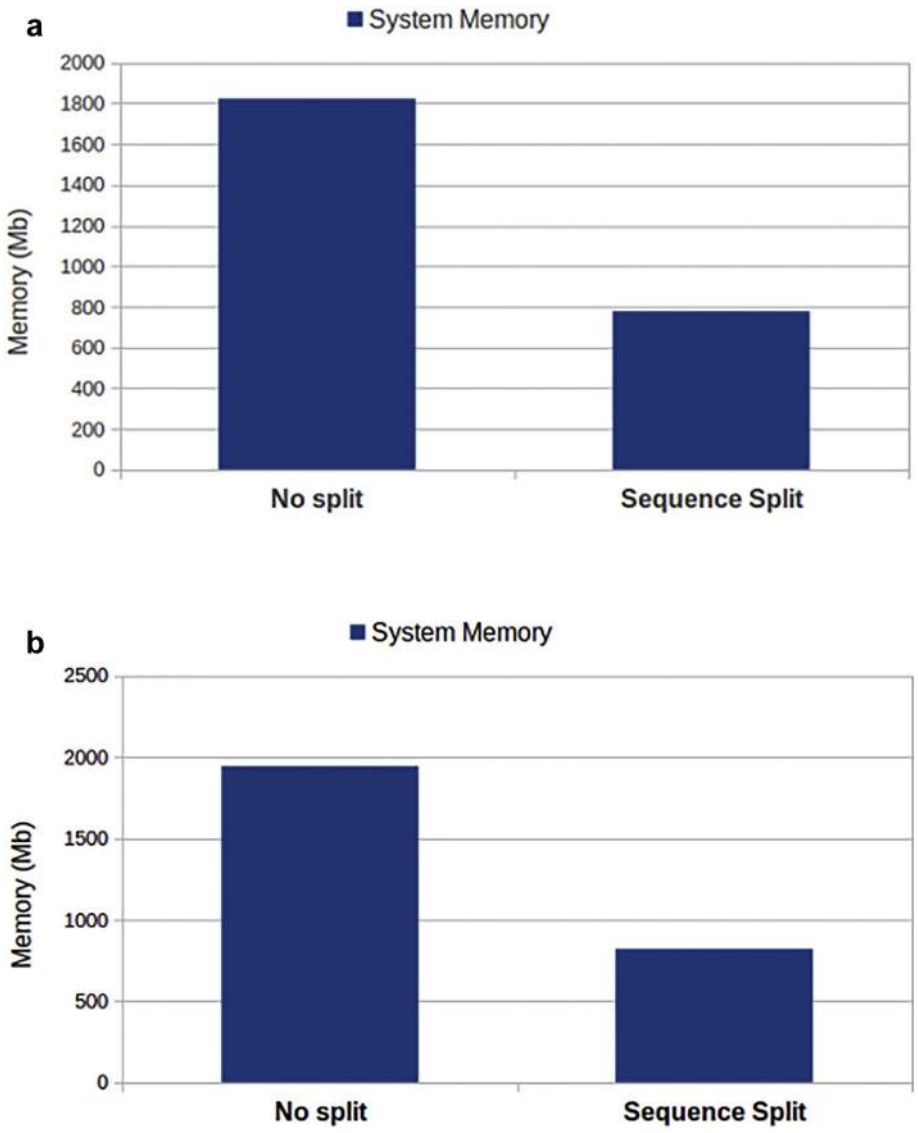

Fig. 7. Comparison of memory usage in the alignment step for *bowtie2* execution with *ParStream-seq* method in (a) 8 core (b) 12 core configuration.

the issue of handling and analysis of biological big data to a great extent.

Acknowledgements

This work has been supported by University Grants Commission (RGNFD) (NO.F./2014-15/RGNF-2014-15D-GEN-WES-65522) and Visvesvaraya PhD Scheme for Electronics and IT, Ministry of Electronics & Information Technology (MeitY) (MLA/MUM/ga/10(37)C), Govt. of India.

Availability

The code is available at the GitHub link https://github.com/sudipmondalcse/ParStream-seq

Appendix A. Supplementary data

Supplementary data to this article can be found online at https://doi.org/10.1016/j.ygeno.2018.11.014.

References

[1] L. Barone, J. Williams, D. Micklos, Unmet needs for analyzing biological big data: a survey of 704 NSF principal investigators, PLoS Comput. Biol. 13 (10) (2017) e1005755.
[2] Z.D. Stephens, et al., Big data: Astronomical or genomical? PLoS Biol. 13 (7) (2015) e1002195.
[3] N. Kathiresan, et al., Accelerating next generation sequencing data analysis with system level optimizations, Sci. Rep. 7 (1) (2017) 9058.
[4] H. Li, R. Durbin, Fast and accurate short read alignment with Burrows-Wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.
[5] R. Li, et al., SOAP: short oligonucleotide alignment program, Bioinformatics 24 (5) (2008) 713–714.
[6] B. Langmead, S.L. Salzberg, Fast gapped-read alignment with Bowtie 2, Nat. Methods 9 (4) (2012) 357–359.
[7] H. Li, J. Ruan, R. Durbin, Mapping short DNA sequencing reads and calling variants using mapping quality scores, Genome Res. 18 (11) (2008) 1851–1858.
[8] T. Rognes, ParAlign: a parallel sequence alignment algorithm for rapid and sensitive database searches, Nucl. Acids Res. 29 (7) (2001) 1647–1652.
[9] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, K. Yelick, meraligner: A fully parallel sequence aligner, Parallel and Distributed Processing Symposium (IPDPS), 2015, pp. 561–570.
[10] C. Hercus, Z. Albertyn, Novoalign, Novocraft Technologies, Selangor, 2015.
[11] C.L. Hung, et al., CUDA ClustalW: an efficient parallel algorithm for progressive multiple sequence alignment on multi-GPUs, Comput. Biol. Chem. 58 (2015) 62–68.
[12] F. Hach, et al., mrsFAST-Ultra: a compact, SNP-aware mapper for high performance sequencing applications, Nucleic Acids Res. 42(Web Server issue) (2014) W494–W500.
[13] J.D. Thompson, T.J. Gibson, D.G. Higgins, Multiple sequence alignment using

ClustalW and ClustalX, Curr. Protoc. Bioinformatics 1 (2003) 2–3.
[14] M.C. Schatz, CloudBurst: highly sensitive read mapping with MapReduce, Bioinformatics 25 (11) (2009) 1363–1369.
[15] T. Nguyen, W. Shi, D. Ruden, CloudAligner: a fast and full-featured MapReduce based tool for sequence mapping, BMC Res. Notes 4 (2011) 171.
[16] M.C. Schatz, BlastReduce: High Performance Short Read Mapping with MapReduce, University of Maryland, 2008.
[17] J.M. Abuin, et al., SparkBWA: speeding up the alignment of high-throughput DNA sequencing data, PLoS One 11 (5) (2016) e0155461.
[18] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[19] P. Zikopoulos, C. Eaton, Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data, McGraw-Hill Osborne Media, 2011.
[20] B. Langmead, et al., Searching for SNPs with cloud computing, Genome Biol. 10 (11) (2009) R134.
[21] H. Li, N. Homer, A survey of sequence alignment algorithms for next-generation sequencing, Brief. Bioinform. 11 (5) (2010) 473–483.
[22] C.C. Law, W.J. Schroeder, K.M. Martin, J. Temkin, A multi-threaded streaming pipeline architecture for large structured data sets, Visualization'99. Proc. IEEE, 1999, pp. 225–232.
[23] A. Kumar, M. Sung, J.J. Xu, J. Wang, Data streaming algorithms for efficient and accurate estimation of flow size distribution, In ACM SIGMETRICS Perform. Evaluat. Rev. 32 (1) (2004) 177–188.
[24] W. Liu, B. Schmidt, G. Voss, W. Muller-Wittig, Streaming algorithms for biological sequence alignment on GPUs, IEEE Trans. Parallel Distributed Systems, 18 2007.
[25] W.B. Langdon, Performance of genetic programming optimised Bowtie2 on genome comparison and analytic testing (GCAT) benchmarks, BioData Min. 8 (1) (2015) 1.
[26] E. González-Agulla, E. Otero-Muras, C. García-Mateo, J.L. Alba-Castro, A multi-platform Java wrapper for the BioAPI framework, Comp. Std. Interf. 31 (1) (2009) 186–191.
[27] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R.J. Fernández-Moctezuma, R. Lax, ... S. Whittle, The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, Proc. VLDB Endow. 8 (12) (2015) 1792–1803.
[28] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 2002, pp. 1–16.
[29] H. Wang, W. Fan, P.S. Yu, J. Han, Mining concept-drifting data streams using ensemble classifiers, Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, 2003, pp. 226–235.
[30] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, Mass storage systems and technologies (MSST), IEEE 26th symposium, 2010, pp. 1–10.
[31] Y. Chan, A. Wellings, I. Gray, N. Audsley, On the locality of java 8 streams in real-time big data applications, Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems, ACM, 2014, p. 20.
[32] Y. Chan, A. Wellings, I. Gray, N. Audsley, A distributed stream library for Java 8, IEEE Trans. Big Data 3 (3) (2017) 262–275.
[33] R. Leinonen, H. Sugawara, M. Shumway, The sequence read archive, Nucleic Acids Res. 39 (Database issue) (2011) D19–D21.
[34] M. Höschele, A. Zeller, Mining input grammars from dynamic taints, Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 720–725.
[35] J. Sharkey, Coding for life–battery life, that is, Google IO Developer Conference, 2009 2009.
[36] R.C. Weisner, How Memory Allocation Affects Performance in Multithreaded Programs, System News for Sun Users (2012).
[37] F. Pop, J. Kołodziej, B. Di Martino, Resource Management for Big Data Platforms: Algorithms, Modelling, and High-Performance Computing Techniques, Springer, 2016.