




2017

A Hybrid MPI-OpenMP Strategy to Speedup the Compression of Big Next-Generation Sequencing Datasets

Sandino Vargas-Perez
WMU

Fahad Saeed
Western Michigan University, fahadsaeed11@gmail.com

Follow this and additional works at: https://scholarworks.wmich.edu/pcds_reports

 Part of the Bioinformatics Commons, and the Computational Engineering Commons

WMU ScholarWorks Citation

Vargas-Perez, Sandino and Saeed, Fahad, "A Hybrid MPI-OpenMP Strategy to Speedup the Compression of Big Next-Generation Sequencing Datasets" (2017). *Parallel Computing and Data Science Lab Technical Reports*. 7.

https://scholarworks.wmich.edu/pcds_reports/7

This Technical Report is brought to you for free and open access by the Computer Science at ScholarWorks at WMU. It has been accepted for inclusion in Parallel Computing and Data Science Lab Technical Reports by an authorized administrator of ScholarWorks at WMU. For more information, please contact wmu-scholarworks@wmich.edu.



A Hybrid MPI-OpenMP Strategy to Speedup the Compression of Big Next-Generation Sequencing Datasets

Sandino Vargas-Pérez¹ and Fahad Saeed^{*,1,2}

¹*Department of Computer Science, Western Michigan University*

²*Department of Electrical and Computer Engineering, Western Michigan University*

Abstract

DNA sequencing has moved into the realm of Big Data due to the rapid development of high-throughput, low cost Next-Generation Sequencing (NGS) technologies. Sequential data compression solutions that once were sufficient to efficiently store and distribute this information are now falling behind. In this paper we introduce *phyNGSC*, a hybrid MPI-OpenMP strategy to speedup the compression of big NGS data by combining the features of both distributed and shared memory architectures. Our algorithm balances work-load among processes and threads, alleviates memory latency by exploiting locality, and accelerates I/O by reducing excessive read/write operations and inter-node message exchange. To make the algorithm scalable, we introduce a novel timestamp-based file structure that allows us to write the compressed data in a distributed and non-deterministic fashion while retaining the capability of reconstructing the dataset with its original order. Our experimental results show that *phyNGSC* achieved compression times for big NGS datasets that were 45% to 98% faster than NGS-specific sequential compressors with throughputs of up to 3GB/s. Our theoretical analysis and experimental results suggest strong scalability with some datasets yielding super-linear speedups and constant efficiency. We were able to compress 1 terabyte of data in under 8 minutes compared to more than 5 hours taken by NGS-specific compression algorithms running sequentially. Compared to other parallel solutions, *phyNGSC* achieved up to 6x speedups while maintaining a higher compression ratio. The code for this implementation is available at <https://github.com/pcdslab/PHYNGSC>.

1 Introduction

Sequencing DNA has become a very fast and low cost process [1] that is pushing Next-Generation Sequencing (NGS) datasets into the realm of Big Data. Highly specialized compression techniques for NGS have been implemented to tackle the task of loosely preserving the key elements of these datasets. Such techniques reduce the enormous amount of resources required to store or transmit these files until further analysis is required.

*Corresponding author e-mail: fahad.saeed@wmich.edu

Table 1: Comparing compression time of genomic data for sequential algorithms versus a parallel algorithm.

| Dataset Size | Sequential | | | Parallel |
|--------------|-----------------|-------------|-----------------|-----------------|
| | NGS-Specialized | | General Purpose | NGS-Specialized |
| | <i>DSRC</i> | <i>Quip</i> | <i>GZip</i> | <i>DSRC 2</i> |
| 10 MB | 0.20secs. | 0.35secs. | 0.87secs | 0.38secs. |
| 100 MB | 2.08 secs. | 2.42secs. | 11.51secs. | 0.47 secs. |
| 1 GB | 26.36secs. | 27.31secs. | 91.10secs. | 1.99secs. |
| 10 GB | 5.56mins. | 4.51mins. | 23mins. | 30.04secs. |
| 100 GB | 35.20mins. | 42.12mins. | 2hrs. | 5.39mins. |
| 1 TB | 5.66hrs. | DNF | DNF | 1.1hrs. |

In order to exploit the characteristics of genomic data, specialized compression solutions have been proposed. Algorithms such as Quip [2], G-SQZ [3], DSRC [4], KungFQ [5], SeqDB [6], LW-FQZip [7], LFQC [8], LEON [9], SCALCE [10] and SOLiDzipper [11] take into account the nature of DNA sequences and use different data compression techniques to achieve good ratios. But these algorithms process the data sequentially and do not fully utilize the processing powers of the newest computing resources, such as GPUs or CPU’s multi-core technologies, to speed up the compression of an increasing amount of genomic data.

Although several general-purpose parallel compression tools are available [12] [13] [14], algorithms specialized in NGS datasets are scarce. A parallel algorithm specific to NGS data, DSRC 2 [15], is a multi-threaded parallel version of DSRC with faster compression times, variable throughput, and comparable compression ratios to existing solutions. Table 1 compares three algorithms running sequentially versus DSRC 2’s multithreading parallel strategy. One can see the difference between specialized sequential compression algorithms (DSRC and QUIP) and the general purpose GZip compressor, which also runs serial code, versus DSRC 2 specialized parallel methods.

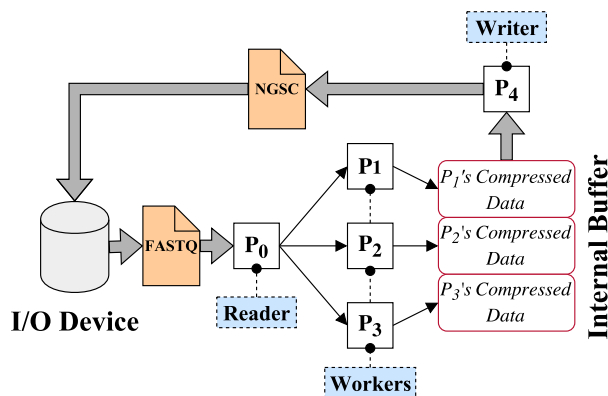
In this paper we present *phyNGSC*, a hybrid strategy between MPI and OpenMP to accelerate the compression of big NGS datasets by combining the best features of distributed and shared memory architectures to balance the load of work among processes, to alleviate memory latency by exploiting locality, and to accelerate I/O by reducing excessive read/write operations and inter-node message exchange. Our algorithm introduces a novel timestamp-based approach which allows concurrent writing of compressed data in a non-deterministic order and thereby allows us to exploit a high amount of parallelism.

In the following sections we analyze the proposed design of *phyNGSC* and evaluate the performance obtained. We discuss the results of running the algorithm for big NGS datasets with distinct combinations that vary in terms of number of processes and threads per process. We then compare our solution with other models that use different approaches to achieve parallel compression.

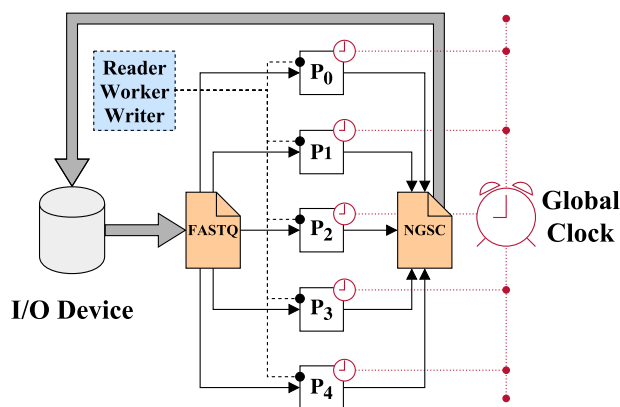
1.1 Our Contribution

One of the problems for parallel applications is concurrent reading and writing [16]. The effects of concurrent memory writing have been extensively studied [17]. However, the specific issues encountered when performing I/O operations for big data are relatively new in the field.

A very common way of writing to a file in parallel is to employ dedicated processes to read and write and use the rest of the processes for computations in a master/slave approach as shown in Fig. 1a. In these parallel compression schemes, the use of a shared memory buffer is required in order for the writer to fetch the data. This guarantees a writing order since only one process is accessing the output file and the data to be written is stored in the shared memory buffer in an identifiable order. However, using this read/write scheme limits the scalability of the parallel genomic compression algorithms because the process in charge of writing has to wait for all of the workers to finish processing the data before it can write to the output file.



(a) Reading and writing scheme commonly used in traditional parallel algorithms specific to compression of NGS datasets.



(b) Reading and writing method used in the proposed hybrid strategy for compression of NGS data in parallel using our **TOC-W** method.

Figure 1: Comparison of the method commonly used to address read and write in parallel algorithms and our proposed method.

In order for a parallel compression algorithm to be more scalable, the compressed form of the data should be writable as soon as its processing is finished. In other words, the compressed form of the data should be writable in a non-deterministic form (from any of the processes) while retaining the properties that would allow reconstruction of the original data. Our proposed *phyNGSC* hybrid parallel strategy, allows us to write data (in a shared file space) from different processes as soon as processing completes. In order to overcome the non-determinism and guarantee an order in the way of writing, we integrated the concept of time-stamping which we call Timestamp-based Ordering for Concurrent Writing or **TOC-W**. Our method uses a global clock and gets a snapshot of the time of completion for the writing operation to the output file as shown in Fig. 1b. This makes the proposed parallel strategy highly scalable since it removes the bottleneck of ordered-concurrent writing in a shared file space. We discuss the methodology in detail in Section 3.

2 Background Information

NGS datasets are commonly structured using the FASTQ format [18]. As shown in Fig. 2, FASTQ is composed of multiple records stored in plain text. Each record contains four lines:

1. *Title line*: Starts with the ASCII character '@' followed by a description of the record.
2. *DNA line*: Contains the DNA sequence read.
3. The third line is represented by the ASCII character '+' which indicates the end of the DNA sequence read, followed by optional repetition of the title line.
4. *Quality score line*: Represents the quality value of the DNA sequence read. It uses ASCII characters from '!' to '~' (33 to 126 in decimal). Each of the ASCII characters will represent a probability that the corresponding nucleotide in the DNA line is incorrect.

```

@ERR026198.1 IL20_5242:5:1:5965:929#1/2
ATCACGTTGATCGGAAGAGCGTCGTGTAGGGAAAG
+
IIIIIIIIIBBBBBBBBBB?B?B?BB?B5?BB?BB?
@ERR026198.2 IL20_5242:5:1:7092:928#1/2
ATCACGTTAAACTGAAGAAAGCGGTAGTGTCTTGG
+
IIIIIIIIIBBBBBBBBBBBB@BBB?BB?B?BBBBBB
@ERR026198.3 IL20_5242:5:1:7799:902#1/2
ATCACGTTGACACTGTAAATAGGACTGACAAGGTC
+
IIIIIIII@BBBBBBBBBBBBBBBBBBBBBBB?B?B
```

Figure 2: FASTQ file structure.

As a proof-of-concept, we will utilize some methods developed for DSRC [4] to underline the compression portion of our hybrid parallel strategy, since it exhibits superior performance for sequential solutions [19].

2.1 Related Research

Hybrid parallel solutions utilizing MPI and OpenMP in high performance computing architectures have been successfully applied to various problems [20] [21] [22]. In [22] the authors described a hybrid framework for interleaving I/O with data compression in order to improve I/O throughput and reduced big scientific data. However, none of these works are specific to genomic datasets which may require domain-specific knowledge about Omics related data.

Another parallel strategy, based on domain decomposition of the NGS data, was introduced by the authors in [23]. In our preliminary work, we implemented a solution that offered modest compression ratio, acceleration, and scalability for up to 100GB of genomic data. The scheme only included a distributed memory model approach and limited optimizations for I/O operations. Also, the compressed form of the data was written in an orderly format i.e., process p_i had to wait until process p_{i-1} finished its writing process before it could start. However, as reported in our paper, super-linear speedups were observed and high compression throughputs were registered.

3 Proposed Hybrid Strategy For Parallel Compression

For our hybrid parallel implementation we considered an SMP system where a FASTQ file of size N bytes is partitioned equally among p homogeneous processes, creating a working region (hereafter referred to as wr) of size $\frac{N}{p}$ for each $p_i, \forall i \in \mathbb{N} : i$ from 0 to $p - 1$. FASTQ records vary in length, hence p_i cannot know if a complete record (containing the 4 lines explained in the previous section) is present at the start and/or at the end of its wr . For this reason we use a small overlap between them to guarantee that each p_i 's wr contains complete records. As sketched in Fig. 3, each process identifies the start of the first complete record at the beginning of its wr by checking the characters until an '@' is encountered and it is confirmed that it belongs to a title line¹. p_i will continue fetching records until it reaches the end of its wr . If it does not have a complete record after reading the last byte, it will continue through the overlap portion and stop when a full record is found. The overlap is three times bigger than the maximum length that a short read can be for common FASTQ files². This will ensure that p_i will finish reading its last record at byte b since $p_{(i+1)}$ started reading its first complete record at byte $b + 1$.

Once the wr is correctly defined for each process, they will fetch chunks of R records out of the FASTQ file and create a threaded region where t number of threads will analyze and compress the title, DNA, and quality score portion of each fetched record, as shown in lines 7 to 15 of the pseudo-code in Fig. 4. Threaded Region 1 is a "loop worksharing", which means that each thread $t_h, \forall h \in \mathbb{N} : h$ from 0 to $t - 1$, will work collectively in solving the k tasks in the threaded region in a many-to-one relationship i.e., all threads will solve one task at a time, until all tasks are done. Processes then will prepare temporary memory buffers (line 17) to store the compression data from threads.

Threaded Region 2 (lines 19-25) will work on the compression of the records. This is a

¹The use of '@' sign to identify the start of the title line is ambiguous since the character can also appear in the quality score line.

²Illumina NextSeq Series Specifications.

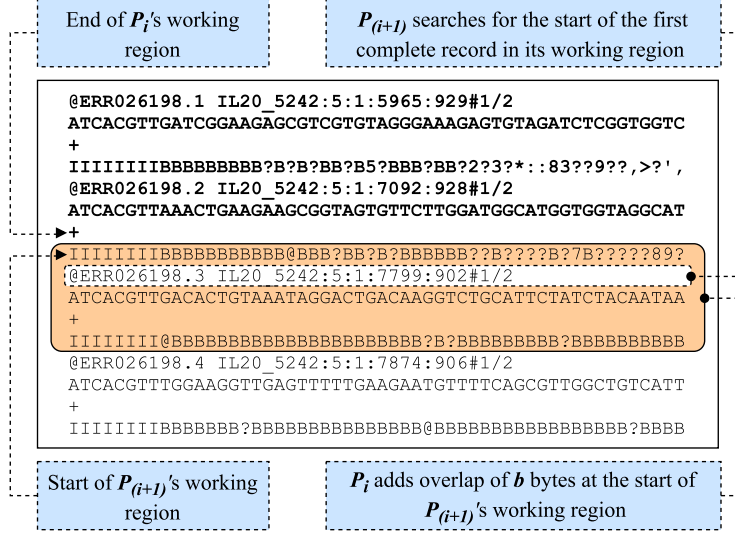


Figure 3: Partition of a FASTQ file among processes into working regions of size $\frac{N}{p}$ bytes each. p_i 's working region ends with an incomplete record and p_{i+1} 's working region starts in the middle of another record. p_i keeps reading beyond its working region until a full record is fetched. p_{i+1} finds the first complete record at the beginning of its working region.

“sections worksharing” region in which a thread t_h will be assigned the completion of one of the k tasks in a one-to-one relationship. In this type of threaded region if the number of threads is greater than the number of tasks, $t - k$ threads will remain idle. Otherwise, a thread t_h will be assigned to an unsolved task until all tasks are complete.

The compressed records will form a subblock which will be copied to a writing buffer wb of size W (lines 27-41). When wb is full, p_i will create a block of compressed data and write it using a shared file pointer to the output NGSC (Next-Generation Sequencing Compressed) file. Immediately after p_i is done writing, a timestamp will be taken and stored in a time of completion toc list to be used later on for the **TOC-W** method (lines 44-48).

When all processes are done fetching and compressing records from their wr , they will gather all the local toc lists into the root process, generally p_0 (line 50). The algorithm will terminate when p_0 builds the **TOC-W**, creates the footer, and writes it to the NGSC file (lines 52-56).

Require: p processes ($\forall p \geq 2$), t threads per p , *FASTQ* file of size N
Ensure: *NGSC* file of size M ($0 < M < N$) and all p exit correctly

```

1: procedure PHYNGSC( $p, t, FASTQ$ )
2:   Initiate MPI environment with  $p$  processes
3:    $p_i$  defines its working region  $wr$  of  $N/p$  bytes
4:   while  $wr$  contains records do
5:      $p_i$  fetches  $R$  records from FASTQ file
6:
7:     Begin Threaded Region 1:  $t$  threads and  $k$  tasks
8:     OMP Loop Worksharing
9:     for all records fetched do
10:       $k_0$ : ALL threads locate records' offset
11:       $k_1$ : ALL threads analyze records' title
12:       $k_2$ : ALL threads analyze records' DNA
13:       $k_3$ : ALL threads analyze records' Quality
14:     end for
15:     End Threaded Region 1
16:
17:      $p_i$  prepares environment for compression
18:
19:     Begin Threaded Region 2:  $t$  threads and  $k$  tasks
20:     OMP Sections Worksharing
21:      $k_0$ : ONE thread compresses EACH record's title
22:      $k_1$ : ONE thread compresses EACH record's DNA
23:      $k_2$ : ONE thread compresses EACH record's Quality
24:      $k_3$ : ONE thread compresses subblock's header
25:     End Threaded Region 2
26:
27:      $p_i$  check available space in writing buffer  $wb$  of size  $W$ 
28:
29:     if  $wb$  is full then
30:        $p_i$  packs subblocks in  $wb$  into a block
31:        $p_i$  writes  $wb$  to NGSC file
32:        $p_i$  adds a timestamp to toc list
33:     end if
34:
35:     Begin Threaded Region 3:  $t$  threads and  $k$  tasks
36:     OMP Sections Worksharing
37:      $k_0$ : ONE thread copies compressed title to  $wb$ 
38:      $k_1$ : ONE thread copies compressed DNA to  $wb$ 
39:      $k_2$ : ONE thread copies compressed Quality to  $wb$ 
40:      $k_3$ : ONE thread copies compressed header to  $wb$ 
41:     End Threaded Region 3
42:   end while
43:
44:   if  $wb$  is not empty then
45:      $p_i$  packs subblocks in  $wb$  into a block
46:      $p_i$  writes  $wb$  to NGSC file
47:      $p_i$  adds a timestamp to toc list
48:   end if
49:
50:    $p_i$  calls mpi_gather(  $p_i$ 's toc list,  $p_0$  gathers )
51:
52:   if  $p_i$ 's rank = 0 then
53:      $p_0$  gathers toc lists
54:      $p_0$  builds TOC-W
55:      $p_0$  writes TOC-W as footer in NGSC file
56:   end if
57:
58:    $p_i$  calls mpi_finalize()
59: end procedure

```

Figure 4: Pseudo-code for *phyNGSC* compression using p processes and t threads per process and a *FASTQ* file.

3.1 TOC-W: Timestamp-based Ordering for Concurrent Writing

Dependencies of a parallel algorithm on previous stages is detrimental to the scalability of a high performance system. Previous attempts to formulate a parallel algorithm for big datasets have been affected by these dependencies. In order for an algorithm to be scalable for big data and/or increasing number of processes, it is required that the algorithm write its results as soon as they are available. However, this is not possible with trivial techniques.

The use of shared file pointers, a feature introduced in MPI-2 [24] and the capabilities of parallel file systems, guarantee that when a process p_i wants to write to the output file it will do so in a non-deterministic fashion i.e., the process that updates the shared file pointer first will write first. This will provide faster writing phases since processes do not have to wait for each other to call a collective write function. Several algorithms have been evaluated in [25] using shared file pointers, showing an increase in writing performance while avoiding dependencies that the file systems have on file locks. Although one can implement the shared file pointers without the file system support as shown in [26]. However, non-determinism poses a considerable challenge at the time of decompression because the algorithm will not know which process wrote a particular block of compressed data and figuring this out will take a significant amount of time and substantial synchronization efforts.

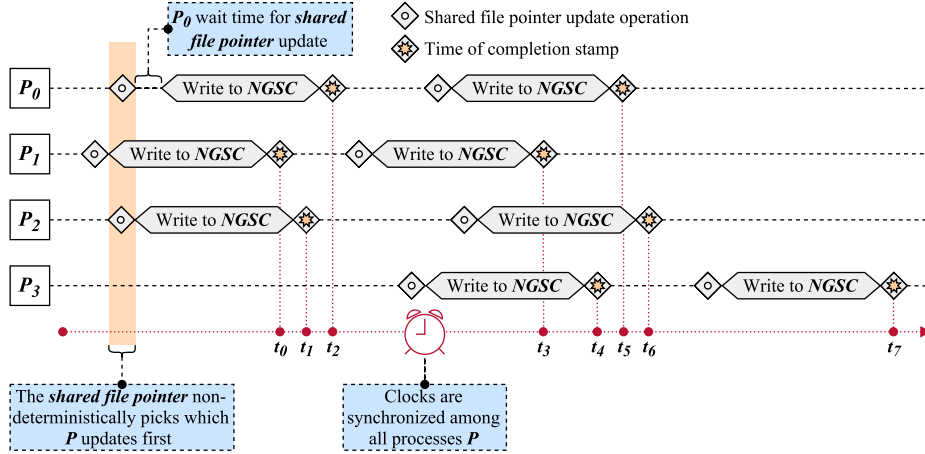
To solve this problem we propose **TOC-W**, a method based on timestamps that guarantees ordering of the data block for concurrent writing on distributed applications. The MVAPICH2 implementation that we use in our strategy guarantees a global, synchronized time for all MPI processes executing a program. Although a clock is local to the node where the process that calls the timer resides, a boolean value “true” in `MPI_WTIME_IS_GLOBAL` will imply that all nodes participating in the parallel execution of the program have the same synchronized time in their clocks. This is what we call a *global clock*. The synchronization of the clocks does not produce a significant computation nor communication overhead since the clocks are synchronized once when `MPI_Init` is called, for the rest of the application’s execution time. The effects of time drift among synchronized clocks and techniques used to keep clocks in sync are studied in [27] and [28] which are taken into account by the MPI implementation use for our proposed strategy. The timestamps for the *toc* are obtained with an MPI call to `MPI_Wtime` by each p_i , which returns a number of seconds (not clock ticks) that represent elapsed wall-clock time, with a precision of 1 millionth of a second³.

In Fig. 5a we can see that a NGSC file is built by each p_i writing blocks of equal lengths in arbitrary order. After a p_i finishes writing a block it takes a timestamp indicating the time of completion or *toc* of that writing operation. The *toc* is the ordering element of the method; it is used to create an ascending sorted list of the timestamps of all processes as in Fig. 5b. This list is then used to assign each process to the corresponding offsets of the blocks it owns within the NGSC file. This information is stored in the footer of the file.

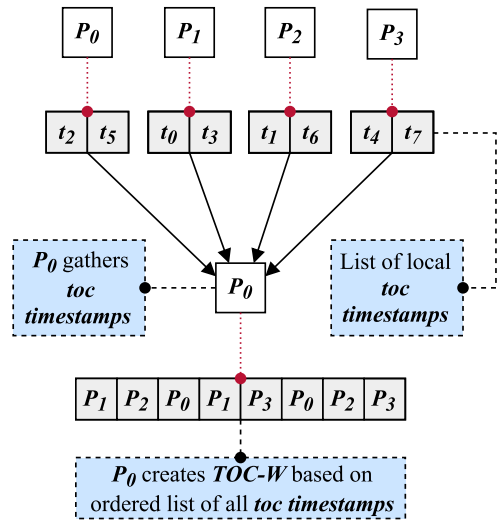
In order to ensure correct timestamps and ordering, the following 2 conditions need to be satisfied:

1. *A global clock*: This will guarantee that each p_i will have synchronized clocks and time will be global for all processes. The MVAPICH2 implementation of MPI provides the boolean attribute `MPI_WTIME_IS_GLOBAL` that indicates whether time is synchronized

³For processors Intel EM64T Xeon E5 2.6 GHz, 8-core.



(a) Diagram showing concurrent writing by each p_i .



(b) TOC-W is build using list of timestamps.

Figure 5: Diagram showing a global, synchronized clock and equal amount of compressed data written to NGSC file for correct **TOC-W** in distributed applications.

across all processes.

2. *Equal amount of data to be written:* Each p_i needs to write an equal amount of compressed data to avoid incorrect ordering. Consider the scenario where p_i and p_j request the shared file pointer by calling `MPI File_write_shared` at the same time. If there are not hardware or network issues, each process will correctly take a timestamp that will indicate which wrote the data first. Network or memory congestion in distributed systems can cause incorrect ordering of the blocks of compressed data. In order to deal with this issue, we introduce a header for each block in the NGSC file marked with a unique signature identifying the process who wrote it. The combination of *both* timestamps and unique signature always allows correct ordering and identification of the data.

Fig. 5a shows the scenario when both conditions are satisfied: A synchronized clock

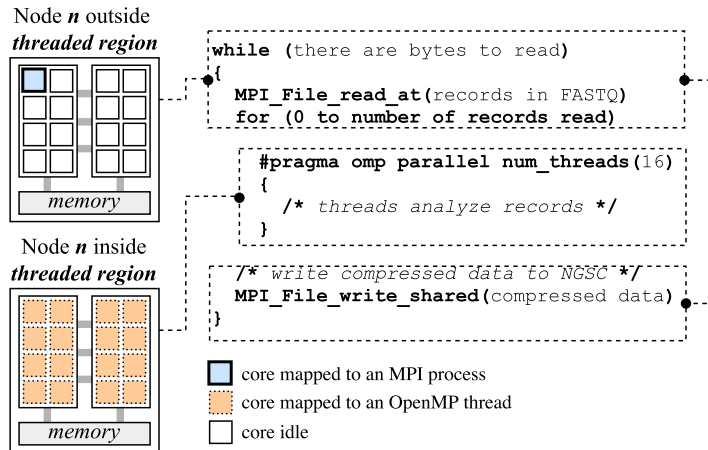


Figure 6: MPI+OpenMP hybrid approach using a *masteronly* model.

among processes and equal amount of data to be written. In the event that multiple processes request an update to the shared file pointer, the update operation will be serialized internally by MPI in a nondeterministic fashion, such that race conditions will be avoided [24]. The time taken by each process waiting to update the shared file pointer, although small, guarantees that at the end of the writing process there is a difference in the time of completion of each writing operation. This is illustrated with process p_0 and p_2 in Fig. 5a. The operation of checking the block’s signature and offsets using MPI requires minimal memory or time-resources [29]. The synchronization of MPI processes and timestamps required for our proposed strategy are discussed below.

3.2 MPI+OpenMP Hybrid Scheme

For our hybrid strategy we implemented the *masteronly* model described in [30]. In this model, an MPI process p_i is mapped to a single core in a typical multi-socket, multi-core SMP node n [31]. When an OpenMP parallel region⁴ is encountered, the number of threads requested will be mapped to the idle cores of node n . The process p_i will become the thread master of the threads inside this threaded region, such that if t threads are requested, $t - 1$ idle cores will be mapped to an OpenMP thread plus the process p_i (now functioning as a thread master). The model is shown in Fig. 6.

The *masteronly* model allows MPI communication (send/receive, read/write, broadcast, and so on) only outside of OpenMP parallel regions. This is advantageous for our design because we want to use OpenMP threads to accelerate computations only, since MPI is optimized for inter-node communication, especially for I/O. Synchronization is also reduced using this schema, thus minimum effort is required to guarantee thread safety.

3.3 NGSC File Structure

The proposed NGSC file is composed of many blocks of equal length W and a footer which contains information to be used for decompression such as the **TOC-W** and the working

⁴In this paper an OpenMP parallel region is referred to as threaded region.

region offsets in the FASTQ file. Each block contains a header with information about itself and a signature that identifies the process that wrote it. Blocks are formed by many subblocks such that the sum of bytes for all the subblocks in the particular block is equal to W . This implies that a block will have complete and incomplete subblocks.

A subblock will have a header and the compressed title, DNA and quality lines of the R records fetched from that process particular working region.

3.4 Analysis of Computation and Communication Costs

Sequential compression algorithms generally have a linear asymptotic growth of $O(N)$, where N is the number of bytes to be compressed. We will refer to N as the problem size given in bytes. In our hybrid parallel solution, each process p_i will process an approximately equal amount of bytes $w = \frac{N}{p}$, let's call w the work load and f the percentage of w that cannot be processed within a threaded region using t threads. With Amdahl's law we define T_t as the factor in which the processing of w increases, i.e., by adding t threads per p , the amount of work required to compressed N bytes will be in the order of $\mathcal{O}(\frac{N}{pT_t})$.

3.4.1 Communication Costs

As every parallel implementation incurs in some amount of communication overhead [32], we try to keep our overhead cost T_o small by reducing the communication rounds to only 3 phases; reading the FASTQ file, writing the compressed data to the NGSC file, and gathering timestamps for our **TOC-W** method (Fig. 7). We assume that the time spent by process p_i reading or writing to an I/O disk d does not depend on the distance between p_i and d , but on the amount of bytes to be read/written. The same assumption will be considered for inter-node communication between a process p_i in node n_i and p_j in node n_j .

Reading Phase Let us assume l is the size in bytes of a chunk of data that a process p_i will read out of its working region $wr = \frac{N}{p}$. Since each p_i will use an explicit offset to read non-collectively the bytes within its wr , the communication overhead will have a complexity of $\mathcal{O}(\frac{N}{lp})$.

Writing Phase Processes can write concurrently using a shared file pointer. When p_i initiates a writing operation of W bytes, it requests an update to the pointer, which reserves W bytes and points to the next available byte to be used by another process. Concurrent access to the shared file pointer is serialized in a non-deterministic way, but this degree of serialization is negligible [26] compared to collective ordered access which incurs in some synchronization overhead. With these assumptions we calculate the overhead incurred in the writing phase by defining r as the compression ratio of the strategy given by $r = \frac{N}{M}$, where M is the size of the compressed output file NGSC. W , as mentioned above, is the size of a compressed block in bytes that every process will write as they become available. Approximately $\frac{M}{W}$ blocks will be written by p processes concurrently using the shared file pointer, hence the complexity for this phase will be represented by $\mathcal{O}(\frac{N}{rWp})$.

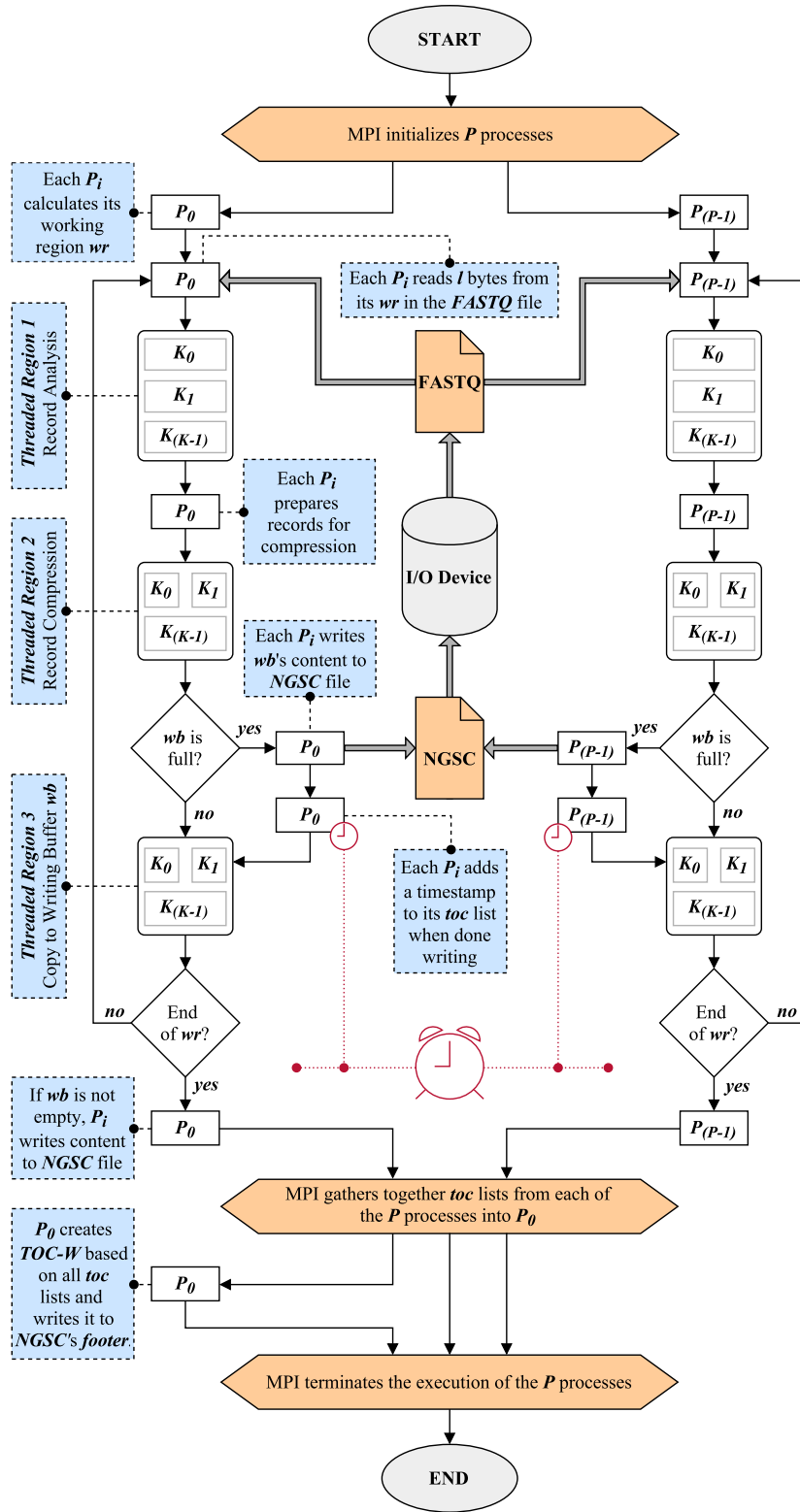


Figure 7: Sketch of the work flow of *phyNGSC*.

Timestamp Gathering Phase The gathering of timestamps is done once per program execution in $\mathcal{O}(\log(p + g))$, where g is the number of timestamps collected by each p_i . g will vary depending on the value of p , N and the amount of writing operations performed.

3.4.2 Time Complexity

The total time complexity of *phyNGSC*, given by $T_P = T_c + T_o$, where T_c represents the computational costs and T_o the overhead costs

$$T_c = \mathcal{O}\left(\frac{N}{pT_t}\right). \quad (1)$$

$$T_o = \mathcal{O}\left(\frac{N}{lp}\right) + \mathcal{O}\left(\frac{N}{rWp}\right) + \mathcal{O}\left(\log(p + g)\right). \quad (2)$$

$$T_P \approx \mathcal{O}\left(\frac{N}{p}\left(\frac{1}{T_t} + \frac{1}{l} + \frac{1}{rW}\right)\right) + \mathcal{O}\left(\log(p + g)\right). \quad (3)$$

3.4.3 Scalability

Calculating the speedup S with increasing number of processes, threads, and data size will tell us if our strategy has good scalability. We will use the definitions and assumptions of [33]: $S = \frac{T_S}{T_P}$, where T_S is the sequential execution time of DSRC, ‘our proof-of-concept’ algorithm. Substituting T_P with equation (3) we will have

$$S = \frac{N}{\frac{N}{p}\left(\frac{1}{T_t} + \frac{1}{l} + \frac{1}{rW}\right) + \log(p + g)}. \quad (4)$$

Let $C = \frac{1}{T_t} + \frac{1}{l} + \frac{1}{rW}$ and eliminate $\log(p + g)$ since its value is very small, then we can reduce equation (4) to

$$S = \frac{N}{\frac{NC}{p}} = \frac{p}{C}. \quad (5)$$

We know that the part $\frac{1}{l} + \frac{1}{rW}$ of C is a constant parameter, hence we can re-write equation (5) to have

$$S \approx pT_t. \quad (6)$$

4 Implementation Results

To test our implementation we used SDSC Gordon⁵, a high performance compute and I/O cluster of the Extreme Science and Engineering Discovery Environment (XSEDE). Gordon

⁵<https://portal.xsede.org/sdsc-gordon>

has 1024 compute nodes, each with two 8-core 2.6 GHz Intel EM64T Xeon E5 (Sandy Bridge) processors and 64 GB of DDR3-1333 memory. The network topology is a 4x4x4 3D torus with adjacent switches connected by three 4x QDR InfiniBand links (120 Gb/s). The cluster implements TORQUE resource manager and has Lustre file system. C++ was used to code our implementation using the OpenMP library to provide thread support and the MVAPICH2 implementation of MPI (version 1.9). The Intel compiler was used for compilation.

To run our hybrid model, the system maps each MPI process and each OpenMP thread to a node’s core (Fig. 6), so if 4 MPI processes are requested with 16 threads per process to be run on 4 nodes, then the system will have 1 core in each of the 4 nodes dedicated to MPI processes. Since each node in the SDSC Gordon cluster has 16 cores, the remaining 15 cores in each node will sleep until an OpenMP parallel region is created. At this moment the MPI processes will become the master thread and join the other 15 threads (mapped to the idle cores on the same node), for a total of 16 threads, to satisfy the amount requested. To keep the nodes at a good performance it is recommended that the value of *MPI processes per node* \times *number of threads per process* do not exceed the total number of cores per node.

The NGS datasets for the experiment were obtained from the 1000 Genomes Project⁶ and are shown in Table 2. In this table the column “Renamed” is used to identify the datasets with relation to their sizes (rounded to the closest power of 10). For clarity, the “Renamed” identification will be used throughout the rest of this paper instead of the original datasets’ IDs.

Table 2: Datasets used for the test. The fastq file SRR622457 was append three times to itself.

| IDs | Organism | Platform | Records | Size(GB) | Renamed |
|-----------|--------------|----------|---------------|----------|---------|
| ERR005195 | Homo sapiens | ILLUMINA | 76,167 | 0.0093 | 10MB |
| ERR229788 | Homo sapiens | ILLUMINA | 889,789 | 0.113 | 100MB |
| ERR009075 | Homo sapiens | ILLUMINA | 8,261,260 | 1.01 | 1GB |
| ERR792488 | Homo sapiens | — | 90,441,151 | 15 | 10GB |
| SRR622457 | Homo sapiens | ILLUMINA | 478,941,257 | 107 | 100GB |
| SRR622457 | Homo sapiens | ILLUMINA | 1,436,823,773 | 1,106 | 1TB |

4.1 Performance Evaluation

We executed the experiments using each of the datasets of Table 2 as input for *phyNGSC*. The algorithm ran with a set of 2, 4, 8, 16, 32 and 64⁷ processes, each mapped to a core in an individual node. Each set of processes used from 1 to 8 threads. All the experiments were repeated several times and the average execution time, max physical memory, and resulting NGSC file size were collected. DSRC version 1 (sequential) and version 2 (multi-threaded) as well as MPIBZIP2 were also tested for comparison with our algorithm.

As can be seen in Fig. 8, the average compression time, measured as the wallclock time elapsed executing code between MPI initialization and finalization, decreases sharply as more

⁶<http://www.1000genomes.org>

⁷SDSC Gordon’s policy only allows a maximum of 64 nodes to be scheduled for use

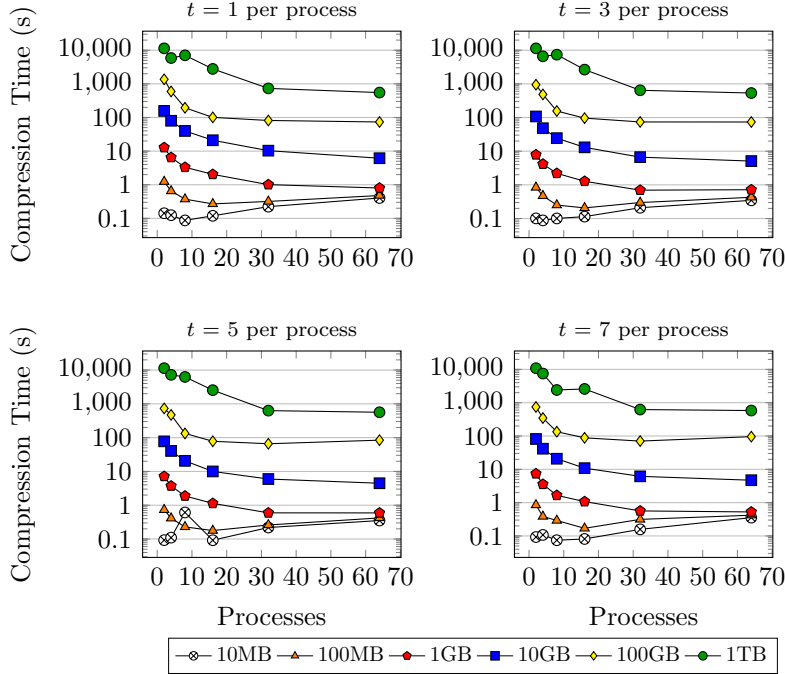


Figure 8: Compression time in seconds for *phyNGSC* with increasing number of processes and different dataset sizes using 1, 3, 5 and 7 threads per process.

processes are added with different datasets. The compression time also improved as more threads were added. These results are in accordance with our theoretical analysis, i.e., the asymptotic growth of the parallel compression time remained close to that represented by T_P in equation (3). Overall the implementation demonstrated a better performance when the number of threads were kept between 3 and 7. Fig. 8 also shows that the compression time starts deteriorating as more processes are added to compressed datasets 10MB and 100MB. Dataset 10MB is smaller than the size of l (3.4.1), set to be 8 megabytes, which causes the communication overhead of the reading phase to be greater than the computations done while compressing. Our strategy was designed to take advantage of coarse-grained task and data parallelism which is very efficient when working with large amounts of data [34].

Table 3: Percentage decrease in compression time for *phyNGSC* compared to DSRC’s sequential compression time.

| Datasets | Number of Processes | | | | | |
|----------|---------------------|--------|--------|--------|--------|--------|
| | 2 | 4 | 8 | 16 | 32 | 64 |
| 10MB | 53.56% | 46.22% | 62.73% | 59.35% | 21.08% | 77.62% |
| 100MB | 59.13% | 81.49% | 86.06% | 91.85% | 84.90% | 79.81% |
| 1GB | 73.37% | 86.49% | 93.70% | 96.21% | 97.88% | 97.99% |
| 10GB | 76.73% | 87.96% | 94.10% | 97.01% | 98.24% | 98.59% |
| 100GB | 68.74% | 83.71% | 93.77% | 96.52% | 97.73% | 96.61% |
| 1TB | 46.70% | 78.06% | 88.60% | 87.53% | 96.94% | 97.38% |

Furthermore, Table 3 shows the percentage decrease in compression time obtained by our solution. A more than 90% decrease in compression time was obtained by using 32 and 64 processes to compress all of the bigger datasets. Also a reduction in time of more than 75% was obtained when we used 4 processes to compress datasets 100MB to 1TB. These results suggest that by doubling the amount of processes and threads the compression time changed by a decreasing percentage of approximately $\frac{T}{p} - \frac{1}{pT}$, in accordance with our theoretical analysis of the computation cost T_c .

Table 4 shows the fastest times obtained by *phyNGSC* compressing different dataset sizes. Using 64 processes our hybrid model compressed 1 terabyte of data in 8.91 minutes, whereas the sequential DSRC took 5.66 hours.

Table 4: Total average of the best time taken by *phyNGSC* to compress different datasets.

| Datasets | DSRC | DSRC 2 | Number of Processes | | | | | |
|----------|---------|---------|---------------------|---------|---------|---------|---------|--------|
| | | | 2 | 4 | 8 | 16 | 32 | 64 |
| 10MB | 0.20s. | 0.25s. | 0.09s. | 0.11s. | 0.07s. | 0.08s. | 0.16s. | 0.36s. |
| 100MB | 2.08s. | 0.36s. | 0.85s. | 0.39s. | 0.29s. | 0.17s. | 0.31s. | 0.42s. |
| 1GB | 26.36s. | 1.99s. | 7.02s. | 3.56s. | 1.66s. | 1.00s. | 0.56s. | 0.53s. |
| 10GB | 5.56m. | 30.04s. | 1.29m. | 40.19s. | 19.69s. | 9.99s. | 5.88s. | 4.71s. |
| 100GB | 35.37m. | 5.39m. | 11.06m. | 5.76m. | 2.20m. | 1.23m. | 0.80m. | 0.51m. |
| 1TB | 5.66h. | 55.63m. | 3.02h. | 1.24h. | 38.69m. | 42.34m. | 10.37m. | 8.91m. |

Fig. 9 shows *phyNGSC* speedup ratio. It shows that super-linear speedups were obtained when compressing the datasets using up to 32 processes and 1, 3, 5, and 7 threads per process, respectively. This behavior is due to the fact that adding threads in the same node allows faster access to a global memory. In addition, the use of their accumulated cache allows the amount of data processed at a given time to fit into the new cache size, reducing the latencies incurred in memory access. This cache effect is studied in [35].

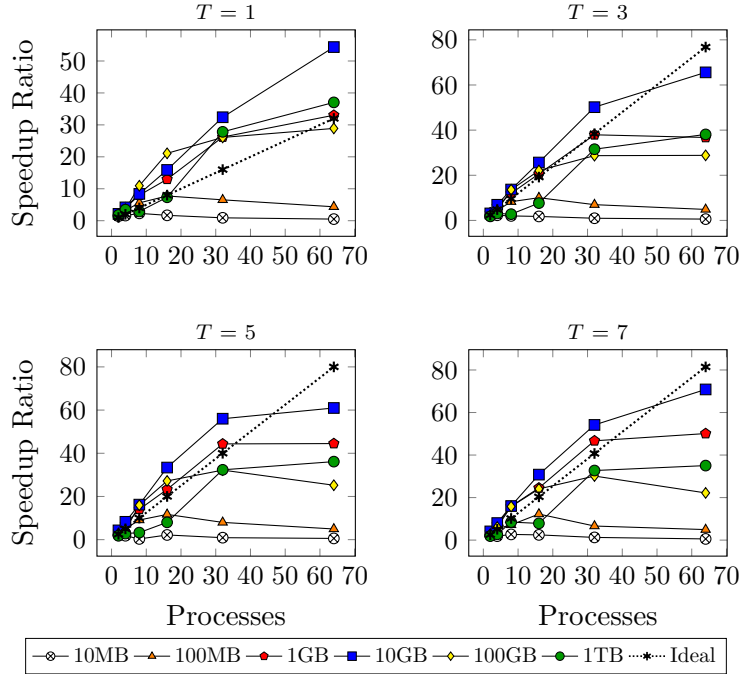


Figure 9: Speedup ratio S for *phyNGSC* with increasing number of processes and different dataset sizes using 1, 3, 5 and 7 threads per process.

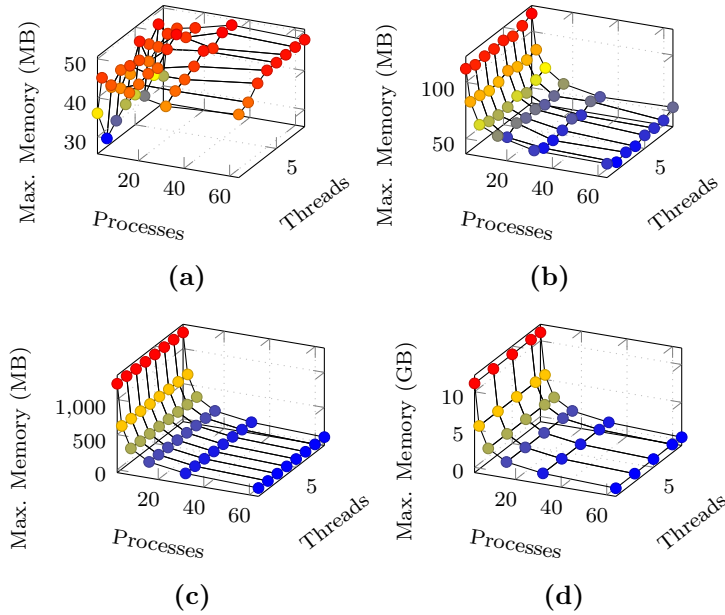


Figure 10: Maximum memory usage for *phyNGSC* with increasing number of processes and different amount of threads per process. (a) 1GB. (b) 10GB. (c) 100GB. (d) 1TB.

Fig. 10 presents the maximum memory consumed by our implementation. We can observe that for datasets 10GB or bigger the memory usage decreases as more processes are added. For smaller datasets, such as those of 1GB, the memory consumption remained lower than

Table 5: Total number of timestamps collected by *phyNGSC* when compressing datasets of different sizes and increasing number of processes.

| Datasets | Number of Processes | | | | | |
|----------|---------------------|--------|--------|--------|--------|--------|
| | 2 | 4 | 8 | 16 | 32 | 64 |
| 10MB | 2 | 4 | 8 | 16 | 32 | 64 |
| 100MB | 2 | 4 | 8 | 16 | 32 | 64 |
| 1GB | 18 | 20 | 22 | 28 | 32 | 64 |
| 10GB | 252 | 254 | 254 | 258 | 262 | 266 |
| 100GB | 2,225 | 2,227 | 2,229 | 2,234 | 2,244 | 2,256 |
| 1TB | 20,825 | 20,826 | 20,830 | 20,832 | 20,842 | 20,860 |

Table 6: Compression throughput (GB/s) of *phyNGSC* for different dataset sizes and increasing number of processes using 7 threads per process.

| Datasets | Number of Processes | | | | | |
|----------|---------------------|------|------|------|------|------|
| | 2 | 4 | 8 | 16 | 32 | 64 |
| 10MB | 0.10 | 0.08 | 0.12 | 0.11 | 0.06 | 0.03 |
| 100MB | 0.12 | 0.27 | 0.36 | 0.62 | 0.34 | 0.25 |
| 1GB | 0.14 | 0.28 | 0.60 | 0.99 | 1.77 | 1.87 |
| 10GB | 0.18 | 0.35 | 0.72 | 1.42 | 2.41 | 3.00 |
| 100GB | 0.16 | 0.31 | 0.81 | 1.44 | 2.21 | 1.48 |
| 1TB | 0.09 | 0.23 | 0.43 | 0.40 | 1.62 | 1.88 |

50 Megabytes per process. Table 5 contains the total number of timestamps collected by processes. Timestamps can also represent a writing operation to the NGSC file and we can see in this table that for smaller datasets, more writing operations are required when using high number of processes, since each p_i will write blocks of size less than W . On the other hand, with bigger datasets processes can pack and write full blocks and reduce writing operations efficiently.

We evaluated the compression throughput of *phyNGSC* to have a measure of the overall compression performance. Throughput will be defined as the amount of data being processed per second, expressed in gigabytes per second (GB/s). Table 6 shows the compression throughput results. Among all the datasets 10GB has the maximum throughput of 3GB/s when compressed with 64 processes, which correlates with its super-linear speedup. 32 to 64 process were used to achieve the best compression throughput for datasets 100GB and 1TB of 2.21GB/s - 1.48GB/s and 1.62GB/s - 1.88GB/s, respectively.

4.2 Comparison with Existing Parallel Compression Algorithms

We mentioned in Section 1 the existence of several parallel compression algorithms, such as pigz, PBZIP2, MPIBZIP2, and DSRC 2 (multi-threaded). We selected DSRC 2 and MPIBZIP2 to compare against our implementation. Fig. 11 shows the compression time for DSRC 2 using 32 threads, *phyNGSC* and MPIBZIP2 using 2, 4, 8, 16, 32 and 64 pro-

cesses each. We can see how *phyNGSC* outperforms DSRC 2 after increasing the number of processes to more than 8. Our implementation did better than MPIBZIP2 using the same amount of processes. MPIBZIP2 did not finish the compression of 1TB using 2 processes after it exceeded a 24 hour execution time limit. Fig. 12 shows the compression ratio algorithms. Our implementation yielded compression ratios that remained within the range of 4.3 and 7.9.

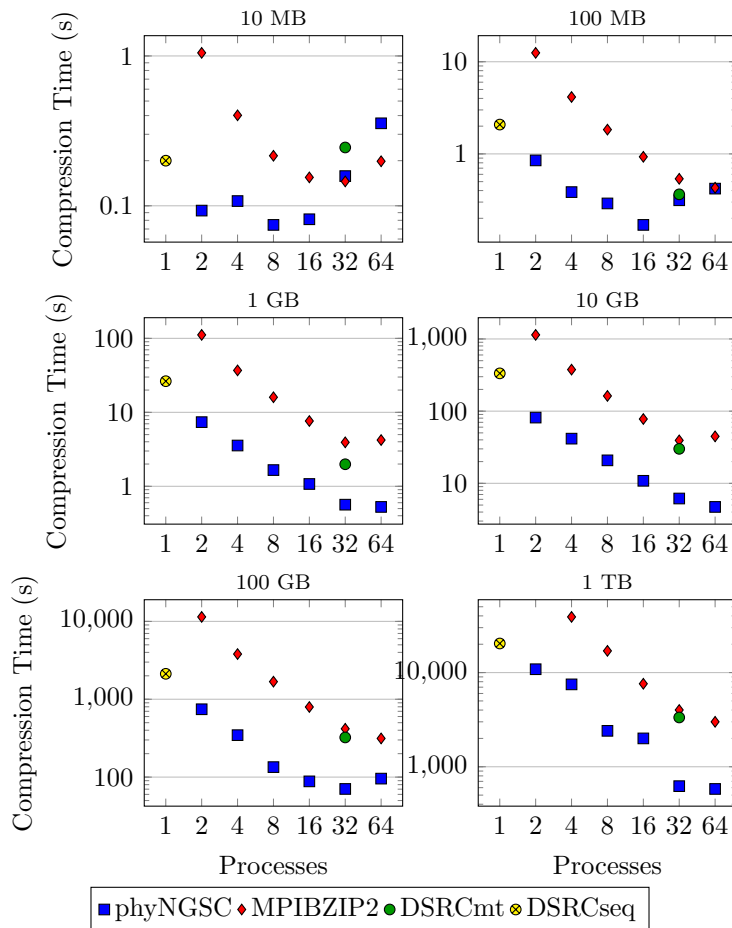


Figure 11: Compression time for different NGS datasets for DSRC sequential, DSRC multi-threaded, and *phyNGSC* and MPIBZIP2 with 2, 4, 8, 16, and 32 processes.

5 Conclusion and Discussion

In this paper we presented a hybrid parallel strategy that utilizes distributed-memory and shared-memory architectures for the compression of big NGS datasets. In order to exploit extreme forms of parallelism we introduce the concept of Timestamp-based Ordering for Concurrent Writing (**TOC-W**) of the compressed form of the data. **TOC-W** presented efficient methods to write the compressed form of the data in a non-deterministic manner while keeping the amount of information required to reconstruct it back to its original state to the minimum. This led to the development of a highly scalable parallel algorithm optimized

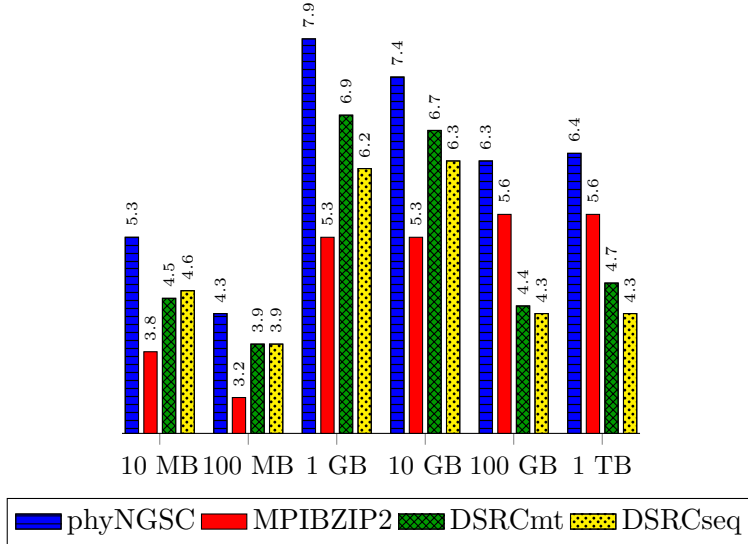


Figure 12: Compression Ratio (y:1) of the resulting NGSC file for DSRC sequential, DSRC multi-threaded, *phyNGSC* and MPIBZIP2. A higher ratio means a better compression.

for Symmetric Multiprocessing (SMP) clusters, which exhibited a tremendous reduction in compression time and faster speedups for most of the datasets when compared to traditional methods of dividing the data and compressing it on individual machines. To the best of our knowledge this is the first attempt to exploit HPC architectures for non-deterministic ordered compression of NGS data.

Theoretical analysis of our hybrid strategy suggested $O(\frac{1}{pT_t})$ reduction in the compression time and this was verified by our experimental results. However, we noticed that Amdahl’s law [36] limited the theoretical linear speedups after adding 64 processes. As was observed with 1TB dataset (Table 4), 32 processes yielded a compression time of approximately 10 minutes and doubling the amount of processes, only produced a 20% reduction in the compression time. Due to the coarse-grained design of our parallel hybrid strategy the smaller datasets (10MB and 100MB) did not benefit as much when more processes were added.

A number of research problems remain open and the techniques presented in this paper suggest new lines of inquiry that can be pursued. For the immediate future, we plan to develop a high-performance decompression strategy by utilizing the novel data structure presented in this paper.

Acknowledgments

This work was supported in part by grant NSF CRII CCF-1464268. The authors would like to acknowledge XSEDE startup grant CCR150017 and the help of staff at the SDSC Gordon computer cluster.

References

- [1] M. A. Quail, I. Kozarewa, F. Smith, A. Scally, P. J. Stephens, R. Durbin, H. Swerdlow, and D. J. Turner, “A large genome center’s improvements to the illumina sequencing system,” *Nature methods*, vol. 5, no. 12, pp. 1005–1010, 2008.
- [2] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze, “Compression of next-generation sequencing reads aided by highly efficient de novo assembly,” *Nucleic acids research*, p. gks754, 2012.
- [3] W. Tembe, J. Lowey, and E. Suh, “G-sqz: compact encoding of genomic sequence and quality data,” *Bioinformatics*, vol. 26, no. 17, pp. 2192–2194, 2010.
- [4] S. Deorowicz and S. Grabowski, “Compression of dna sequence reads in fastq format,” *Bioinformatics*, vol. 27, no. 6, pp. 860–862, 2011.
- [5] E. Grassi, F. Di Gregorio, and I. Molineris, “Kungfq: A simple and powerful approach to compress fastq files,” *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 9, no. 6, pp. 1837–1842, Nov. 2012.
- [6] M. Howison, “High-throughput compression of fastq data with seqdb,” *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 10, no. 1, pp. 213–218, Jan. 2013.
- [7] Y. Zhang, L. Li, Y. Yang, X. Yang, S. He, and Z. Zhu, “Light-weight reference-based compression of fastq data,” *BMC bioinformatics*, vol. 16, no. 1, p. 1, 2015.
- [8] M. Nicolae, S. Pathak, and S. Rajasekaran, “Lfqc: a lossless compression algorithm for fastq files,” *Bioinformatics*, p. btv384, 2015.
- [9] C. Lemaitre, D. Lavenier, E. Drezen, T. Dayris, R. Uricaru, G. Rizk *et al.*, “Reference-free compression of high throughput sequencing data with a probabilistic de bruijn graph,” *BMC Bioinformatics*, vol. 16, pp. 288–288, 2015.
- [10] F. Hach, I. Numanagić, C. Alkan, and S. C. Sahinalp, “Scalce: boosting sequence compression algorithms using locally consistent encoding,” *Bioinformatics*, vol. 28, no. 23, pp. 3051–3057, 2012.
- [11] Y. J. Jeon, S. H. Park, S. M. Ahn, and H. J. Hwang, “Solidzipper: A high speed encoding method for the next-generation sequencing data,” *Evolutionary bioinformatics online*, vol. 7, p. 1, 2011.
- [12] M. Adler. pigz: A parallel implementation of gzip for modern multi-processor, multi-core machines. HTML. [Online]. Available: <http://zlib.net/pigz/>
- [13] J. Gilchrist. Parallel bzip2 (pbzip2) data compression software. HTML. [Online]. Available: <http://compression.ca/pbzip2/>
- [14] ——. Parallel mpi bzip2 (mpibzip2) data compression software. HTML. [Online]. Available: <http://compression.ca/mpibzip2/>

- [15] L. Roguski and S. Deorowicz, “Dsrc 2industry-oriented compression of fastq files,” *Bioinformatics*, vol. 30, no. 15, pp. 2213–2215, 2014.
- [16] L. Lamport, “Concurrent reading and writing,” *Commun. ACM*, vol. 20, no. 11, pp. 806–811, Nov. 1977.
- [17] F. E. Fich, P. L. Ragde, and A. Wigderson, “Relations between concurrent-write models of parallel computation,” in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’84. New York, NY, USA: ACM, 1984, pp. 179–189.
- [18] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, “The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants,” *Nucleic acids research*, vol. 38, no. 6, pp. 1767–1771, 2010.
- [19] H. Na, H. Luo, W. Ting, and B. Wang, “Multi-task parallel algorithm for dsrc,” *Procedia Computer Science*, vol. 31, pp. 1133–1139, 2014.
- [20] T. V. T. Duy, K. Yamazaki, K. Ikegami, and S. Oyanagi, “Hybrid mpi-openmp paradigm on SMP clusters: MPEG-2 encoder and n-body simulation,” *CoRR*, vol. abs/1211.2292, 2012. [Online]. Available: <http://arxiv.org/abs/1211.2292>
- [21] X. Wang and V. Jandhyala, “Enhanced hybrid mpi-openmp parallel electromagnetic simulations based on low-rank compressions,” in *Electromagnetic Compatibility, 2008. EMC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–5.
- [22] E. R. Schendel, S. V. Pendse, J. Jenkins, D. A. Boyuka II, Z. Gong, S. Lakshminarasimhan, Q. Liu, H. Kolla, J. Chen, S. Klasky *et al.*, “Isobar hybrid compression-i/o interleaving for large-scale parallel i/o optimization,” in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 61–72.
- [23] S. Vargas-Pérez and F. Saeed, “A parallel algorithm for compression of big next-generation sequencing datasets,” in *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 03*. IEEE Computer Society, 2015, pp. 196–201. [Online]. Available: <http://dx.doi.org/10.1109/Trustcom.2015.632>
- [24] M. P. I. Forum, *MPI: A Message-Passing Interface Standard*, v2.1 ed., PDF, 2008. [Online]. Available: <http://www.mpi-forum.org/docs/mpi-2.1/mpi21-report.pdf>
- [25] K. Kulkarni and E. Gabriel, “Evaluating algorithms for shared file pointer operations in mpi i/o,” in *Computational Science–ICCS 2009*. Springer, 2009, pp. 280–289.
- [26] R. Latham, R. Ross, R. Thakur, and B. Toonen, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 12th European PVM/MPI Users’ Group Meeting Sorrento, Italy, September 18-21, 2005. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ch. Implementing MPI-IO Shared File Pointers Without File System Support, pp. 84–93.

- [27] S. Hunold and A. Carpen-Amarie, “On the impact of synchronizing clocks and processes on benchmarking mpi collectives,” in *Proceedings of the 22Nd European MPI Users’ Group Meeting*, ser. EuroMPI ’15. New York, NY, USA: ACM, 2015, pp. 8:1–8:10. [Online]. Available: <http://doi.acm.org/10.1145/2802658.2802662>
- [28] A. Lastovetsky, V. Rychkov, and M. O’Flynn, *MPIBlib: Benchmarking MPI Communications for Parallel Computing on Homogeneous and Heterogeneous Clusters*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 227–238. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87475-1_32
- [29] A. Chan, W. Gropp, and E. Lusk, “An efficient format for nearly constant-time access to arbitrary time intervals in large trace files,” *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, 2008.
- [30] R. Rabenseifner and G. Wellein, “Communication and optimization aspects of parallel programming models on hybrid architectures,” *International Journal of High Performance Computing Applications*, vol. 17, no. 1, pp. 49–62, 2003.
- [31] R. Rabenseifner, G. Hager, and G. Jost, “Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes,” in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. IEEE, 2009, pp. 427–436.
- [32] B. Barney. (2010) Introduction to parallel computing. HTML. Lawrence Livermore National Laboratory. [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp
- [33] A. Y. Grama, A. Gupta, and V. Kumar, “Isoefficiency: Measuring the scalability of parallel algorithms and architectures,” *IEEE concurrency*, vol. 1, no. 3, pp. 12–21, 1993.
- [34] M. I. Gordon, W. Thies, and S. Amarasinghe, “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs,” *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 151–162, Oct. 2006. [Online]. Available: <http://doi.acm.org/10.1145/1168919.1168877>
- [35] J. Benzi and M. Damodaran, “Parallel three dimensional direct simulation monte carlo for simulating micro flows,” in *Parallel Computational Fluid Dynamics 2007*. Springer, 2009, pp. 91–98.
- [36] M. D. Hill and M. R. Marty, “Amdahl’s law in the multicore era,” *Computer*, no. 7, pp. 33–38, 2008.