Computer Architecture Aware Optimisation of DNA Analysis Systems

Hasindu Gamaarachchi

A thesis in fulfillment of the requirements for the degree of

Doctor of Philosophy

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

July 2020

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **Gamaarachchi**

First name: **Hasindu** Other name/s:

Abreviation for degree as given in the University calendar: **PhD**

School: **School of Computer Science and Engineering** Faculty: **Faculty of Engineering**

Title: Computer Architecture Aware Optimisation of DNA Analysis Systems

Abstract

Information about the structure and the functioning of an organism is encoded in the molecule called Deoxyribonucleic Acid (DNA). The process that converts the massive amount of chemically encoded data to a computer readable form is called DNA sequencing. The time and the cost for this process were immense a decade ago. However, currently it takes less than 3 days at a cost of only $1000. This will be even more affordable in few years. The output from the sequencer is analysed on a computer to extract useful information. The extracted information is being extensively used for finding mutations for diseases, improving clinical diagnosis and finding effective treatments. This has given rise to personalised medicine which will significantly impact the future health care. Though DNA sequencing has significantly improved, the analysis techniques are still lagging. The analysis process is very compute intensive and is done on super computers. Super computers are very expensive and large and only top research facilities possess them. Further, uploading confidential information through the Internet causes privacy issues. I will be building an embedded system for DNA sequence analysis. Most suitable DNA analysis algorithms for an embedded system will be identified. Many suitable embedded processors will be connected in an appropriate topology. Analysis algorithms and data structures will be modified and optimised to suit the selected processors. The outcome will be a low cost, energy efficient, sufficiently fast and portable system which will enable analysis of a DNA within a doctor's office or a pathology lab itself.

Signature **Hasindu Gamaarachchi** Witness Date **01 March, 2020**

FOR OFFICE USE ONLY Date of completion of requirements for Award

Originality Statement

Hasindu Gamaarachchi
01 March, 2020

Copyright Statement

Hasindu Gamaarachchi
01 March, 2020

Authenticity Statement

Hasindu Gamaarachchi
01 March, 2020

Abstract

Information about the structure and the functioning of an organism is encoded in the molecule called Deoxyribonucleic Acid (DNA). The process that converts the massive amount of chemically encoded data to a computer readable form is called DNA sequencing. The time and the cost for this process were immense a decade ago. However, currently it takes less than 3 days at a cost of only $1000. This will be even more affordable in few years. The output from the sequencer is analysed on a computer to extract useful information. The extracted information is being extensively used for finding mutations for diseases, improving clinical diagnosis and finding effective treatments. This has given rise to personalised medicine which will significantly impact the future health care. Though DNA sequencing has significantly improved, the analysis techniques are still lagging. The analysis process is very compute intensive and is done on super computers. Super computers are very expensive and large and only top research facilities possess them. Further, uploading confidential information through the Internet causes privacy issues. I will be building an embedded system for DNA sequence analysis. Most suitable DNA analysis algorithms for an embedded system will be identified. Many suitable embedded processors will be connected in an appropriate topology. Analysis algorithms and data structures will be modified and optimised to suit the selected processors. The outcome will be a low cost, energy efficient, sufficiently fast and portable system which will enable analysis of a DNA within a doctor's office or a pathology lab itself.

Acknowledgement

Contents

List of Figures

List of Tables

Abbreviations

Chapter 1

Optimisation of Nanopore Sequence Analysis for Many-core CPUs

Nanopore sequencing is a third generation (the latest) genome sequencing technology. These modern advances in computational genomics are reshaping healthcare through life-saving applications in medicine and epidemiology, where quick turn-around time of results is critical. Nanopore sequence analysis software tools are inefficient in utilising the computing power offered by modern High Performance Computing systems equipped with many-core CPUs and RAID systems. In this chapter, we present a systematic experimental analysis to identify the potential bottlenecks, which reveals that the primary bottleneck is the thread-inefficient HDF5 library used to load nanopore data. We propose multiple optimisation strategies suitable for different practical scenarios to alleviate the bottleneck: 1) a new file format that offers upto $\sim$42$\times$ I/O performance improvement; and, 2) a multi-process based solution for the scenario when using a new file format is not possible, that offers upto $\sim$32$\times$ I/O performance improvement.

We demonstrate the efficacy of our optimisations by integrating them to the popular

Nanopolish toolkit. Our experiments using a representative nanopore dataset demonstrate that the proposed optimisations enable improved scaling of overall-performance with the number of threads (∼6.5× for 4 vs. 32 threads). Moreover, they also lead to overall-performance improvement (∼2× for 4 threads and ∼6.5× for 32 threads) and improved CPU utilisation (from 69% to 99% for 4 cores and from 22% to 85% for 32 threads) for a given number of threads, when compared to the original *Nanopolish*.

Refer to the Appendix ??

1.1 Introduction

Computational genomics has turned a new chapter in medical sciences and epidemiology [35, 40]. It enables promising applications such as accurate disease diagnosis, identifying genetic predisposition, and precision medicine [39]. *Genome sequencing* converts the genetic and biological information encoded in DNA molecules into computer readable data, which is typically hundreds or thousands of gigabytes in size. *Nanopore sequencing* is a leading third-generation (the latest) genome sequencing technology [30]. Computational genomics software tools analyse the huge amount of data generated by genome sequencing to extract meaningful information for the above applications.

Quick turn-around time of results in such applications is highly desirable. For instance, quick diagnostics can instantiate immediate treatments. Moreover, rapid results would enable faster tracking of disease spreading in epidemiological applications such as the ongoing Corona virus outbreak [9]. However, to analyse the enormous amount of data with high speed, genomic computation software tools demand massive computing time. Thus, scientists typically use High Performance Computing (HPC) systems to run these software tools [38].

A modern HPC system offers significant computational power through many-core CPUs that are to be exploited through parallelism. The major advantage in such systems when compared to an ordinary personal computer is the availability of number of cores in the

CPUs. Moreover, HPC systems have RAID storage composed of many disks for higher I/O throughput with the added benefit of reliability [4]. *Unfortunately, the existing software tools for nanopore sequencing are generally not capable of efficiently utilising the large number of cores available in many-core HPC systems, and thus fail to take maximal advantage of the available computing power.* Consequently, the overall execution time of the applications on an expensive HPC system may not improve significantly when compared to its execution on a less expensive workstation or a personal computer (refer to chapter **??**). *In this chapter, we present software optimisations in nanopore software tools to enable them to take maximal advantage of the computing power offered by modern many-core HPC systems.*

To demonstrate the problem mentioned above, we present a motivational example using *Nanopolish* [42], which is a popular state-of-the-art nanopore raw data analysis toolkit [33].

Motivational Example: We executed the *call-methylation* tool in *Nanopolish* toolkit on a representative dataset[1]. The experiment was performed on a high-end HPC system with 36 Intel Xeon cores[2] using different number of threads. The graph in Fig. 1.1a plots the execution time (y-axis) for *Nanopolish* against the number of threads (x-axis). We observe that when the tool is executed with four threads, the execution-time is nearly 10 hours. The execution-time does not improve significantly with increasing number of threads, and there is little improvement beyond 16 threads. Laptop results and an example on another server refer to supplementary material.

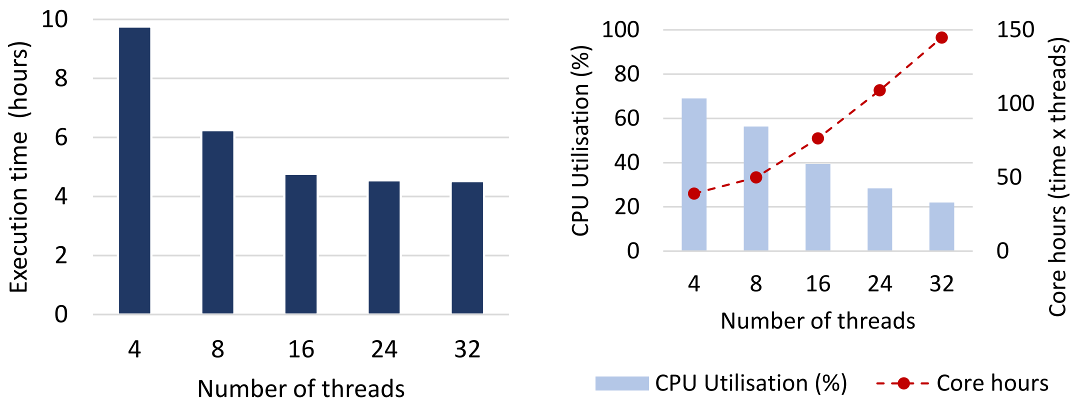
To analyse further, Fig. 1.1b plots the CPU utilisation[3] (left y-axis) and the core-hours[4] (right y-axis) for each case in the above experiment. The CPU utilisation, for execution with four threads, is less than ideal (69%). Moreover, as the number of threads increase, the CPU utilisation decreases significantly. Specifically, when executed with 32 threads,

[1]See the experimental setup under results for details of the dataset.

[2]See system S1 in Table 1.4 for the specification of the HPC system.

[3]CPU utilisation is calculated as in results.

[4]Core-hours is inspired by the common term *man-hour*. It is equal to the product of the number of hours and the number of cores/threads [5].



(a) Execution time

(b) CPU utilisation & core-hours

Figure 1.1: Variation of (a) execution time, (b) CPU utilisation and core-hours in original *Nanopolish* with the number of data processing threads.

the CPU utilisation is as low as 22%. We also observe that the core-hours (which should be constant with the number of threads in an ideal case) increase significantly, and hence depicting that employing greater number of threads is inefficient and not highly beneficial.

Thus, procuring an HPC system with a higher number of CPU cores might not be beneficial for achieving quick turn-around time of results for nanopore software tools, and there is a need for software optimisations in nanopore software tools to exploit the available resources in HPC systems. *To this end, in this chapter, we first present a systematic experimental analysis to identify the potential bottlenecks that hinder the efficient utilisation of CPU resources in nanopore software tools. Then we present multiple optimisations–suitable for different practical scenarios–to overcome these bottlenecks and enable performance improvements.* Our experiments using the state-of-the-art *Nanopolish* toolkit on HPC systems demonstrate that our proposed optimisations enable improved CPU utilisation and hence improved performance scaling with the number of threads (Fig. 1.9 and 1.11). Moreover, they also enable improved performance for a given number of threads with respect to the original *Nanopolish.* For example, for 32 threads, the CPU utilisation increases up to ∼85% (which was 22%), and a ∼6.5× speed up is achieved when compared to the original *Nanopolish.* We believe that such improved performance will facilitate fast diagnostics and rapid epidemic response.

Contributions: The key novel contributions of this chapter can be summarised as follows.

- We, for the first time, present a systematic analysis to identify the potential bottlenecks in nanopore software tools. The analysis reveals that the primary bottleneck is caused by a limitation in an underlying library (HDF5) that serialises disk accesses from multiple threads (Section 1.3).
- We propose an alternate file format (SLOW5) that alleviates the bottleneck by allowing random accesses from multiple parallel threads. The proposed file format is designed by exploiting the domain knowledge of nanopore sequencing (Section 1.4.1).
- In some scenarios, it may not be practically possible to use a new file format. Therefore, we present a second solution based on multi-processes. This solution alleviates the bottleneck without requiring any modification to the existing file format (Section 1.4.2).
- We demonstrate that the new multi-FAST5 file format–which is projected as a replacement of the existing FAST5 file format by the research community–also suffers from the same bottleneck, thus our proposed SLOW5 format is superior. Moreover, our multi-process based solution is also effective in alleviating the bottleneck in multi-FAST5 (Section 1.5.5).

chapter Organisation: Section 1.2 discusses the background and related work. Section 1.3 elaborates our analysis for identifying bottlenecks and its explanation. Our proposed optimisations and solutions are presented in Section 1.4. Section 1.5 presents our experimental setup and results. Finally, Section 1.6 is the discussion and the chapter is concluded in Section 1.7.

1.2 Background

1.2.1 I/O

Two types of I/O: synchronous I/O (blocking I/O) and asynchronous I/O (non-blocking I/O) are in the context of random accesses (opposed to sequential/streaming access) are discussed in subsections 1.2.1.1 and 1.2.1.2, respectively.

1.2.1.1 Synchronous I/O

Synchronous I/O is convenient to be programmed, and such programmed code are legible. Thus synchronous I/O is the most popular and predominantly used amongst typical programmers. Following is a simplified account of how random disk requests are served in a modern operating system.

Consider a single-threaded program that requests I/O using standard *read* or *write* system calls (buffered read/write API calls such as *fwrite*, *fread*, *fprintf*, *getline*, etc are eventually mapped to these system calls). These system calls are synchronous calls which return when the requested data is read from the disk.

In Fig. 1.2, the user-space thread is performing a synchronous I/O request. The operating system receives the system call and queues the disk request in its disk request queue. Momentarily, the user-space thread is put to sleep by the operating system, since a disk request is expected to take hundreds of thousands of CPU clock cycles. The operating system will schedule the disk request (assign to the disk controller) based on policies and priority levels imposed. The disk controller will perform the operation and the operating system will wake up the thread, once requested data reading is completed. If the disk system has a single disk, effectively one request can be served at a time[5]. If the disk system has K disks, up to K requests may be served simultaneously, depending on the

[5]as the discussion is about random accesses, disk request merge operations are infrequent

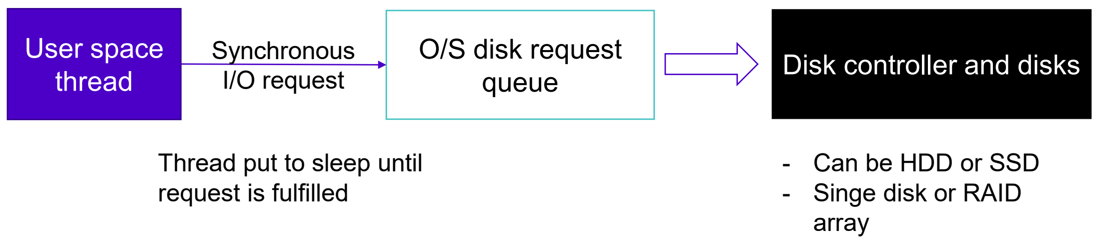

User space thread
Synchronous I/O request
O/S disk request queue
Disk controller and disks
Thread put to sleep until request is fulfilled
- Can be HDD or SSD
- Singe disk or RAID array

Figure 1.2: Elaboration of synchronous I/O

RAID level; i.e. K simultaneous parallel reads are possible on a RAID 0 system with K disks.

Consider a program with a single thread requesting I/O as shown in Fig. 1.2. Let t be the average disk request service time (from the time of the system call to when the thread is woken up). Let a single thread be requesting n synchronous disk reads sequentially. Despite the number of requests n, the total time spent on disk reading T is : $T = t \times n$.

Now consider a program with multiple threads requesting I/O (I/O threads) as shown in Fig. 1.3. One thread put to sleep due to an I/O request, does not limit other threads from requesting I/O . Thus, if we launch K I/O threads and if the disk controller can serve K requests in parallel, the total time for disk reading is T' : $T' = t \times \frac{n}{K}$

The scenario in Fig. 1.3 is achieved by programs where threads are having an independent code path - where each processing thread independently performs disk accesses on demand. However, in programs that perform data processing batch by batch, where one single thread reads a batch of data from the disk and assigns to multiple processing threads to be processed in parallel, it is the scenario in Fig. 1.2.

1.2.1.2 Asynchronous I/O

Asynchronous I/O is pertinent to highly responsive applications like web servers and database servers. In asynchronous I/O, the system call that requests I/O will return immediately. The user-space thread won't be put to sleep and this can continue submit

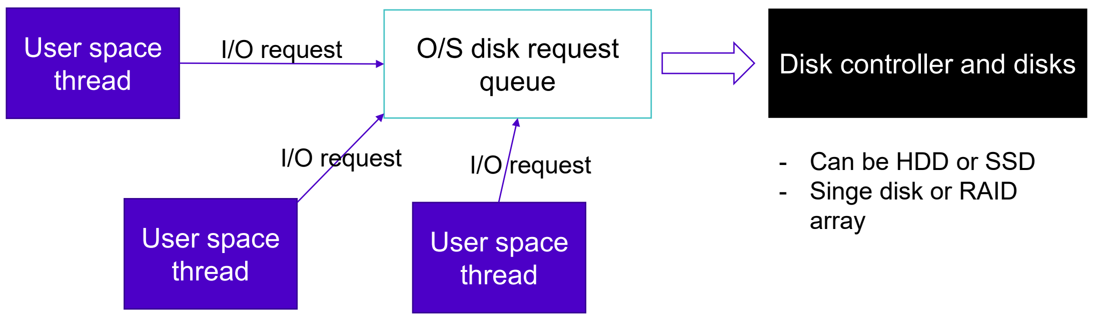


Figure 1.3: Elaboration of multi threaded synchronous I/O

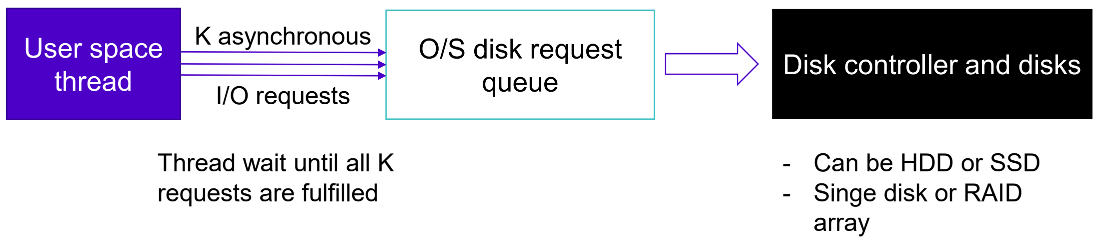

User space thread
K asynchronous
I/O requests
O/S disk request queue
Disk controller and disks
Thread wait until all K requests are fulfilled
- Can be HDD or SSD
- Singe disk or RAID array

Figure 1.4: Elaboration of asynchronous I/O

another I/O requests or or execute some other task while the disk request is being served.

Consider the asynchronous I/O example in Fig. 1.4 where a single thread submits multiple I/O requests to the operating system simultaneously. In Fig. 1.4, the single user space thread submits K I/O requests in parallel. Then the thread can either poll or wait for a notification from the operating system for I/O request completion. Assume we have n total disk requests to be performed. If the disk system can perform K accesses in parallel and if the time for a single disk accesses is t (K parallel accesses take t as well), the total time $T' = t \times \frac{n}{K}$. Note that the time is the same as for Fig. 1.3.

This type of asynchronous I/O is suitable when a program performs reading data and processing batch by batch where one thread performs I/O and then assigns multiple threads or to an accelerator card (eg: GPU to be processed). This is in contrast to independently processing threads we discussed under synchronous I/O above, as threads need to converge in this case.

Asynchronous I/O can be performed by: (1) native synchronous I/O system calls in the operating system or (2) a library that emulates asynchronous I/O through a thread pool that use synchronous I/O system calls in the operating system.

From the two methods above, (1) allows 'real' asynchronous I/O, but only if supported by the operating system. Early Linux kernels (before version 2.5) did not have native asynchronous I/O systems calls, however, they are available in modern Linux kernels starting from version 2.5 onwards [6]. Despite that, asynchronous I/O implementation in the Linux kernel has been a controversial topic amongst Linux developers [1], is complicated [6], have various drawbacks [6, 31] and does not support certain file systems such as NFS [7]. GNU C Library (Glibc) does not provide wrapper functions for asynchronous I/O system calls [45]. Instead the programmers must use low-level system calls which are not easy and non-portable (Architecture specific). Third party libraries such as *libaio* [32] which uses Linux native asynchronous I/O system calls have attempted to provide an abstract interface.

An example of the method (2) above is the current Portable Operating System Interface (POSIX) compatible asynchronous I/O (AIO) library provided by Glibc. POSIX AIO implementation in GlibC is provided in the user-space and uses multiple threads [24]. The developer of POSIX AIO have admitted that their approach is expensive and have scalability issues which are expected to be fixed in the future through a state-machine-based implementation of asynchronous I/O [24]. Further, POSIX AIO is not implemented in all systems (eg: Windows subsystem for Linux)

While the POSIX AIO is suitable for typical I/O loads, the programmers can also spawn multiple I/O threads per batch and assign the disk accesses amongst them. This is suitable if the batch size is big and the thread spawning time is small compared to the I/O time of the batch.

1.2.2 Nanopore raw data analysis

Genomics: DNA is a molecule composed of a long strand of millions of units called *nucleotide bases* (or simply called *bases*). Genome sequencers read a DNA strand in relatively smaller fragments (around 10,000 bases long in nanopore) and converts them into digitised data, termed as *reads* [30] in the domain of bioinformatics. In this chapter we refer to them explicitly as *genomic reads* to avoid confusion with disk reads.

Nanopore Sequence Analysis: Nanopore sequencing is a leading third-generation (the latest) sequencing technology [30]. Oxford Nanopore Technologies (ONT) is the company that produces nanopore sequencers. A nanopore sequencer is composed of an array of pico-ampere range current sensors that measure the ionic current disruptions when DNA fragments pass through nanometer scale protein pores [20, 30]. The raw sensor output for a *genomic read* is a time series current signal and is referred to as *raw signal* or *raw data*. ONT stores the raw signal and other metadata (e.g., sampling frequency) in a file format called FAST5 [34]. FAST5 is essentially a Hierarchical Data Formats 5 (HDF5) [17] file with a specific schema defined by ONT. The only existing library for accessing HDF5 format is the official library developed and maintained by the non-profit organisation HDF Group [36, 37].

***Nanopolish*:** Raw data analysis toolkits analyse the sequencer outputs using complex algorithms and extract meaningful information. *Nanopolish* is currently a popular state-of-the-art nanopore raw data analysis toolkit. *Nanopolish* is used in a number of genomic workflows such as methylation detection [42], variant detection [25], draft genome polishing [19, 29] and real-time molecular epidemiology for the ongoing Corona Virus outbreak [9]. *Nanopolish* is written in C/C++ and supports multi-threaded execution through OpenMP. It is an open-source toolkit with a large and complex codebase [41, 42].

Previous Work on Optimising *Nanopolish*: Nanopore sequence analysis is a relatively new field that only emerged in the last decade. Thus, optimisation efforts to improve performance of nanopore software tools are rare. In *Nanopolish*, calculation of log likelihood ratio is a predominantly used CPU intensive computation kernel [42]. To reduce

CPU time for log likelihood computation, *Nanopolish* authors have already employed a fast table-driven log-sum implementation established elsewhere in [10]. However, none of the existing works have focused on improving the overall performance of *Nanopolish* on HPC systems with many-cores and Redundant Array of Independent Disks (RAID). Our proposed optimisations are orthogonal to the methods discussed above.

Previous Work on Optimising Sequence Analysis: Several optimisation efforts exist for the second generation sequencing software (also known as *Next Generation Sequencing*) [2, 14, 15, 21–23]. However, software used for nanopore sequencing (third-generation) is distinct from the first and second generations [3]. Nanopore technology involves processing raw signal data, which is not the case for first and second generation.

In this chapter, we for this first time, identify the major causes behind the inefficient resource-utilisation by nanopore software tools and present multiple optimisations to alleviate those issues.

1.3 Identification and Explanation of bottleneck

The motivational example in Section 1.1 revealed that *Nanopolish* is unable to efficiently utilise multiple cores in the system. There can be two reasons for an application to be unable to utilise parallel resources. These are: 1) data processing bottleneck; and/or 2) I/O bottleneck. In this section, we identify and explain that the primary reason of the under-utilisation is I/O bottleneck.

1.3.1 Identification of the Bottleneck

We employed performance monitoring and profiling tools in the motivational example setup, to hypothesise the causes of inefficient resource-utilisation and performance.

Hypothesis-1: The performance of the software tool is bounded by file I/O. We observed through *htop* utility in Linux that majority of *Nanopolish* threads are in the 'D' state.

The 'D' state is defined as the 'state of the process for disk sleep (uninterruptible)' [11]. This leads to our first hypothesis that the software tool is bounded by file I/O. In fact, *Nanopolish* incurs a large number of random disk accesses when reading millions of FAST5 files (based on HDF5) in a nanopore dataset[6].

Hypothesis-2: The file I/O bottleneck is caused by the HDF5 library and not by the limitation of physical disks. We observed the disk usage statistics through the *iostat* utility to find that disk system is not fully utilised (i.e., the observed number of disk reads per second was around 100, while the particular disk system could handle more than 1000 IOPS). This implies that I/O bottleneck is not due to the limitation of physical disks to serve data fast enough to saturate the processor. To investigate further, we profiled *Nanopolish* with *Intel Vtune* under *concurrency profiling.* It reveals that the majority of the 'wait time' is due to a conditional variable (synchronisation primitive) in the underlying library called HDF5 library (Hierarchical Data Format 5–used to access raw nanopore data stored in FAST5 file). A closer look into the HDF5 library revealed that the thread-safe version of the HDF5 library serialises the calls for disk read requests [18]. Thus, we hypothesise that the reduced CPU utilisation is caused by the disk requests being serialised by the HDF5 library, consequently causing the bottleneck and limiting the utility of a multi-disk RAID system.

1.3.2 Verification of the Identified Bottleneck

To verify the above identified cause of the bottleneck, we performed a deeper analysis. For this, we first restructured *Nanopolish* such that wall-clock time spent on I/O operations and data processing can be separately measured to determine the time spent on individual components in the program.

We run the restructured *Nanopolish* with various number of threads (for FAST5 access

[6]A nanopore dataset of a single genome sample contains millions of *genomic reads* (fragments of DNA), and each *genomic read* is stored in a separate FAST5 file. Thus, accessing millions of such *genomic reads* incurs millions of random disk accesses (opposed to sequential/streaming access)

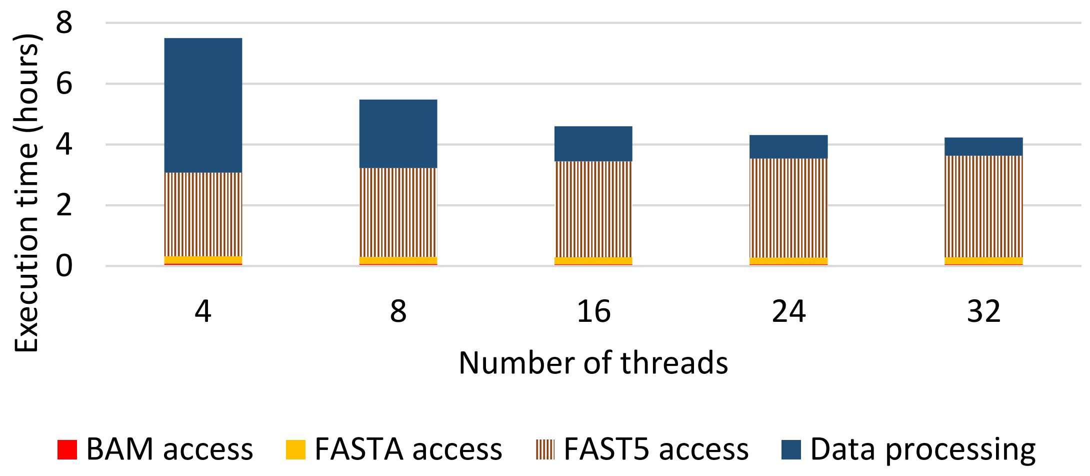


Figure 1.5: Decomposition of time for individual components in restructured *Nanopolish*.

and data processing). The results are presented in Fig. 1.5. The x-axis in the figure represents the number of threads used and the y-axis represents the total execution time. Different colours in the bars (see legend) denotes the decomposition of the total execution time into different components[7].

We observe from Fig. 1.5 that: 1) the contribution by the BAM access and FASTA access to the overall execution time is negligible; 2) a major portion of the time is consumed by FAST5 access (patterned brown); 3) time consumption of FAST5 access (patterned brown) does not improve with increasing number of threads; and, 4) data processing time improves with increasing number of threads (solid blue). This confirms that the bottleneck is caused by file I/O and not because of any data processing bottlenecks.

1.3.3 Detailed Explanation of the Identified Bottleneck

In this subsection, we explain the major limitation in HDF5 library that prevents efficient parallel accesses, consequently causing the bottleneck.

[7]FASTA access refers to random access to reference genome (stored in FASTA file format) performed using *faidx* component in *htslib* library. BAM access refers to sequential access performed through *htslib* library to the genomic alignment records (stored in BAM file format)

HDF5 Library and its Limitations in Thread Efficiency: HDF5 library uses synchronous I/O calls and even the latest HDF5 implementation (HDF5-1.10) does not support asynchronous I/O[8]. This, by itself, is not an issue as multiple synchronous I/O operations can be performed in parallel using multiple I/O threads to exploit the high throughput of RAID systems in HPC systems. Fig. 1.3 demonstrates how multiple I/O threads can be used to perform parallel disk accesses using synchronous I/O. Suppose the disk system has K disks, up to K requests may be served simultaneously depending on the RAID level; i.e., K simultaneous parallel reads are possible on a RAID 0 system with K disks. Let t be the average disk request service time (from the time of the system call to when the thread is woken up). For a program that launches K I/O threads and if the disk controller can serve K requests in parallel, the total time for n disk reads is $T' = t \times \frac{n}{K}$.

However, the HDF group (that maintains the HDF5 library) mentions that the thread-safe version of the HDF5 library is not thread efficient and that it effectively serialises the calls for disk read requests [18]. The global lock in the thread safe version of the HDF5 library creates limitations. Following is an extract from the FAQ section of the HDF web site [18]. "Users are often surprised to learn that (1) concurrent access to different datasets in a single HDF5 file and (2) concurrent access to different HDF5 files both require a thread-safe version of the HDF5 library. Although each thread in these examples is accessing different data, the HDF5 library modifies global data structures that are independent of a particular HDF5 dataset or HDF5 file. HDF5 relies on a semaphore around the library API calls in the thread-safe version of the library to protect the data structure from corruption by simultaneous manipulation from different threads. Examples of HDF5 library global data structures that must be protected are the freespace manager and open file lists."

Thus, in spite of having multiple I/O threads, I/O requests for HDF5 files have to go through the HDF5 library. Fig. 1.6 demonstrates this, where K I/O threads are requesting

[8]In synchronous I/O calls, the OS, upon receiving the call puts the user-space thread to sleep and the thread can no longer submit I/O requests until the disk reading is completed and woken by the OS. Conversely, asynchronous I/O system calls return immediately without the thread being put to sleep and the thread can continue to submit another asynchronous request.

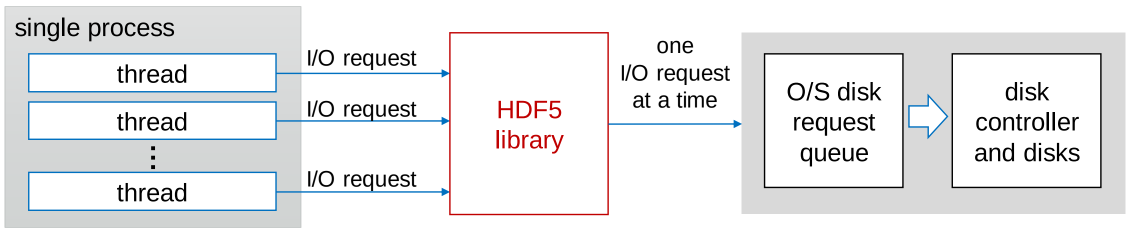


Figure 1.6: Elaboration of the limitation in HDF library.

I/O from the HDF5 library in parallel. However, the lock inside the HDF5 serialises the parallel requests, effectively issuing only one request at a time to the operating system disk request queue. The operating system will put the thread to sleep and this is equivalent to a single I/O thread. Thus, the total time spent on disk accesses T will be $T = t \times n$, and essentially, the high throughput capability of multiple disks in a RAID configuration is under-utilised.

1.4 Proposed Optimisations

In this section, we present two types of solutions to overcome the bottleneck in nanopore software tools. The first approach is to use an alternate file format (Section 1.4.1). However, current nanopore software tools have been developed on top of the FAST5 (HDF5) format because of its adoption by Oxford Nanopore technologies as the file format for storing the raw signal. Thus, using a new file format may not always be practical. For such scenarios, we present a second solution that uses multi-processes instead of multi-threads for I/O operations (Section 1.4.2). This second solution does not require any changes to FAST5 format or the HDF5 library. Furthermore, we also present a few more optimisations to *Nanopolish* that enable further speed-up (Section 1.4.3).

1.4.1 Alternate File Format (SLOW5)

We propose a new file format called SLOW5[9] for storing nanopore raw signal data as an alternate to FAST5. We considered the domain knowledge from nanopore sequence analysis and the characteristics of disk accesses to design the new file format.

SLOW5 File Format: We design our proposed SLOW5 file format by extending the simple and well-known tab-separated values (TSV) format, using inspiration from the gold standard genomic file formats such as SAM [27] and VCF [8]. An example of the proposed file format is shown in Table 1.1. The structure of the file is explained below.

The first set of lines of the SLOW5 file comprises the file header. Each header line starts with the character #. Generic metadata such as the file version and global metadata of the sequenced genome sample are also stored in the header. The global metadata is common to all the *genomic reads* and contains information such as the sequencing flow-cell identifier, and sequencing run identifier, etc. The last line in the header gives the column names of the upcoming data, which are tab-separated. Note that, not all metadata and data fields are depicted in Table 1.1 for the sake of brevity.

The header is followed by data where each line (row) represents a single *genomic read*. In other words, for *N genomic reads*, there are *N* data lines in the file. The *genomic read* information fields (e.g., read-identifier, number of signal samples, and the raw signal) are tab separated and are in the same order as defined in the last line of the header. The *raw_signal* column contains the current signal values separated by commas. Note that all data corresponding to a single *genomic read* are placed contiguously in the same row, thus facilitating locality in disk accesses.

Working Explanation: Random accesses to the *genomic read* records in a SLOW5 file are facilitated by an index called the SLOW5 index. The SLOW5 index is another tab-separated file as shown in Table 1.2. Each line corresponds (except the first header line) to a *genomic read*. The first column is the read-identifier of the *genomic read*, the second

[9]The name SLOW5 is ironical to FAST5

column is the file offset to the corresponding SLOW5 record, and the third column is the size of the corresponding SLOW5 record in bytes (including the new line character). For performing random disk accesses to SLOW5, the SLOW5 index is first loaded to a hash table in RAM where the read-identifier serves as the hash table key and the rest of the data is used as hash table values. For a given read-identifier, the file offset and the record length is obtained from this hash table and the program can move the file pointer to the offset (i.e. using *lseek*) and load the record to the memory. Multiple random accesses to SLOW5 can be performed in parallel either through synchronous I/O calls with multiple threads, or through asynchronous I/O if supported by the operating system. Note that the raw signal data is read-only during nanopore sequence analysis. Thus, SLOW5 is inherently thread-safe without any need of global locks.

1.4.2 Multi-process based Solution

The original *Nanopolish* runs a single process with multiple threads. We propose a multi-process based solution for scenarios where the existing FAST5 cannot be replaced. Multiple threads in a single process share same address space and thus the lock in HDF5 library affects multiple threads. Multiple threads are typically used to run sub-tasks in parallel while conveniently sharing data amongst the threads. In contrast, multiple processes have their own independent address spaces and are typically used to run isolated tasks in parallel. We exploit the presence of independent address spaces in multiple processes to circumvent the lock in HDF5 library.

Overview: Our proposed multi-process based solution is elaborated in Fig. 1.7. We use multi-threads in the single parent-process for data processing and multiple child-processes for I/O. The parent-process performs data processing using multiple threads in parallel. Each child-process has its own instance of the HDF5 library, as a consequence of independent address spaces. Moreover, each child-process has only a single thread that requests I/O. Thus, a single instance of the HDF5 library gets only one request at a time. In effect, there are multiple instances of the HDF5 library that can submit multiple I/O requests in

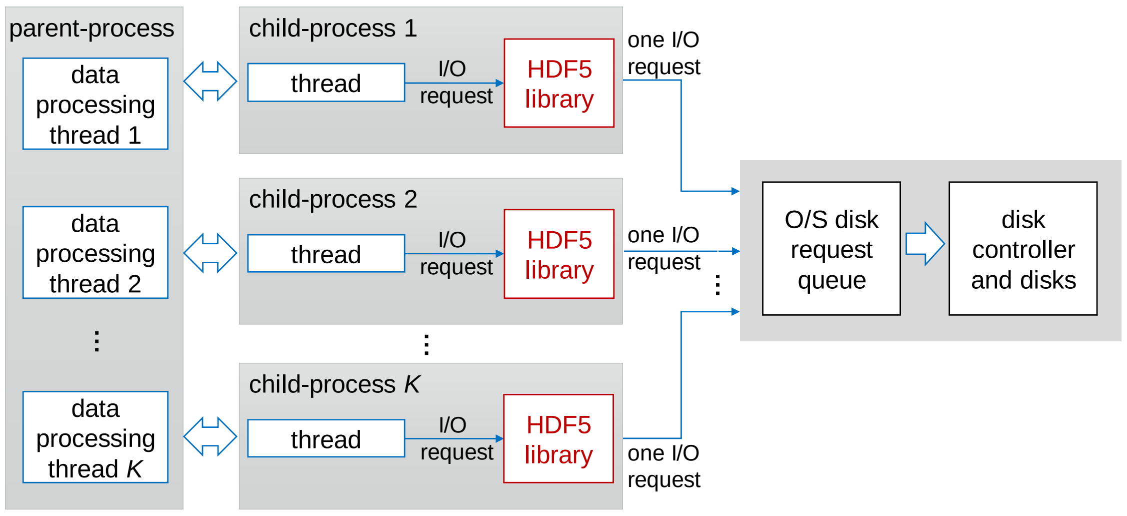


Figure 1.7: The proposed multi-process based solution.

parallel to the operating system (as opposed to the situation in Fig. 1.6), thus benefiting from the high throughput offered by the RAID configuration. Formally, if there are K processes and if the disk controller can serve K requests in parallel, the total time spent on I/O operations will be $T' = t \times \frac{n}{K}$ (similar to the case in Fig. 1.3).

Details: The proposed multi-process based solution can be adopted for Nanopore data processing using a pool of processes that performs FAST5 I/O. Multiple processes are spawned at the beginning of the program using the *fork* system call. These forked child-processes form a pool of processes that exist until the lifetime of the parent-process, solely performing I/O of FAST5 files. The data processing can be performed by multiple threads spawned by the parent-process as usual. The parent-process when it requires to load signal data of N reads (FAST5 accesses), first splits the list of reads to K parts where K is the number of child-processes. Then, each part is assigned to each child-process, which performs the assigned FAST5 accesses. When data is loaded, the child-processes send data to the parent-process. The communication (data transfer) between the parent-process and child-processes can be implemented relatively easily using unnamed pipes out of the available Inter Process Communication (IPC) techniques (still not easy as threads that share the same memory space).

Note-1: A fork-join model for multi-processes (as could be done for the multi-threading) is

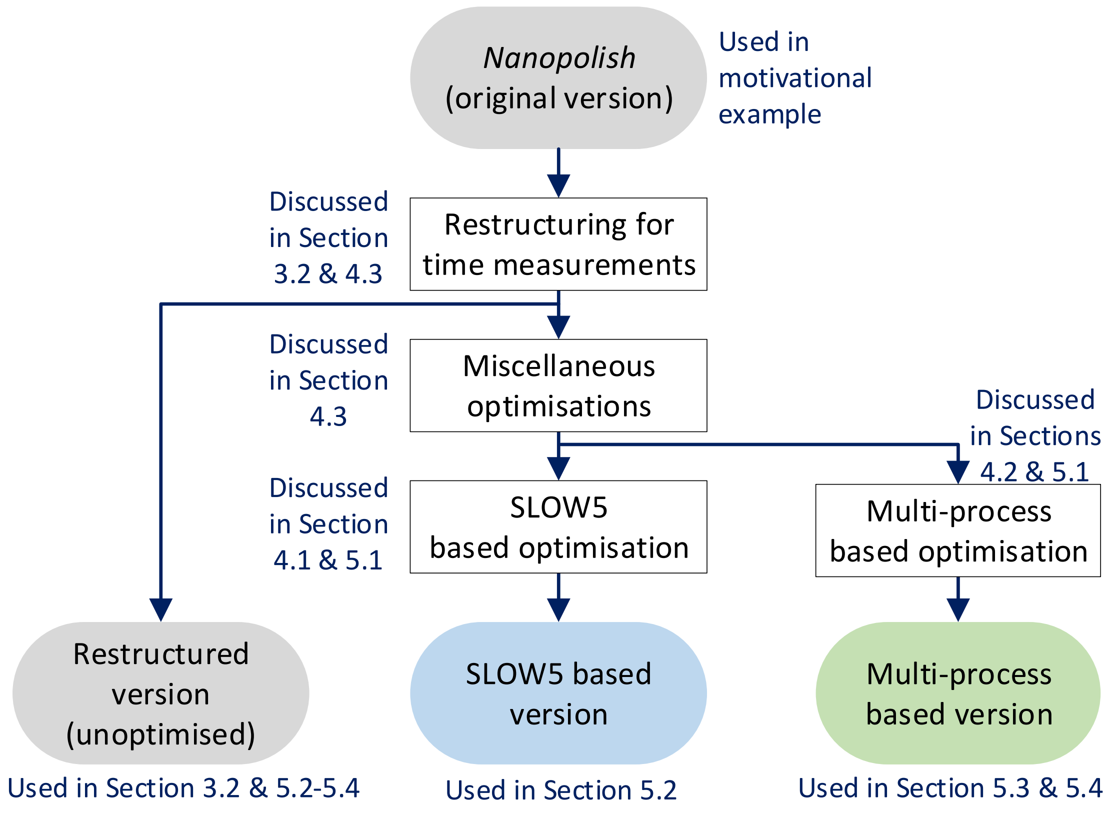

Nanopolish
(original version)
Used in motivational example
Discussed in Section 3.2 & 4.3
Restructuring for time measurements
Discussed in Section 4.3
Miscellaneous optimisations
Discussed in Sections 4.2 & 5.1
Discussed in Section 4.1 & 5.1
SLOW5 based optimisation
Multi-process based optimisation
Restructured version (unoptimised)
SLOW5 based version
Multi-process based version
Used in Section 3.2 & 5.2-5.4
Used in Section 5.2
Used in Section 5.3 & 5.4

Figure 1.8: Flow diagram depicting modifications to *Nanopolish*

unsuitable to be used instead of the process pool model presented above. Firstly, creating a process can be very expensive and could easily become the biggest bottleneck than the file reading itself. Secondly, forking in the middle of a program could double the memory usage and is usually problematic.

Note-2: We propose the use of multi-threads in the single parent-process for data processing and multiple child-processes for I/O. The possibility of using separate processes for both data processing and I/O is discussed in Section 1.6.1 with its caveats.

1.4.3 Restructuring & Miscellaneous Optimisations

In addition to the above I/O related optimisations, we also performed restructuring and a few other software optimisations with respect to multi-threading and memory. Our restructuring allows us to measure the execution time separately for I/O (including the execution time breakdown for different file formats) and data processing, without significant effect on performance, whereas, the software optimisation improve the processing time.

The original *Nanopolish* implementation uses openMP for multi-threading. We restructured *Nanopolish* to perform multi-threading using a lightweight fork-join model with work-stealing implemented using POSIX Threads (*pthreads*). Moreover, the restructured *Nanopolish* performs I/O operations and data processing batch by batch (batch of *genomic reads*), i.e., a batch of *genomic reads* are loaded from the disk and the batch is then processed, subsequently, results of the batch are written to disk. I/O operations are interleaved with data processing, i.e. when the first batch is being processed, the second batch will be loaded from the disk.

The restructured *Nanopolish* was further optimised with strategies such as: reducing the number of memory allocations (*malloc*) for dynamic 2D arrays by allocating a 1D array, an appropriate batch size that fits the available RAM, and a better load-balancing between multi-threads, etc. While space limits our ability to explain each of these optimisations, the details can be found in the open sourced code of this research project. For the sake of clarity, the overview of our restructuring and optimisations to original *Nanopolish* and various resulting versions with their usage, are shown in Fig. 1.8.

Table 1.1: Example of SLOW5 file format

SLOW5 file format						
#fileformat: slow5v1.0						
#exp_start_time: 2020-01-01T00:00:00Z						
#run_id: 855cdb4b26948						
#flow_cell_id: FAH00000						
#read_id	n_samples	digitisation	offset	range	sampling_rate	raw_signal
read-0	123456	8192	6	1467.6	100000	498,492,501,508,503,505,509,...
read-1	2000	8192	5	1467.6	4000	400,401,500,403,407,478,510,...
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
read-N	10000	8192	3	1467.6	4000	559,545,560,551,550,565,701,...

1.5 Experiment and Results

1.5.1 Experimental Setup

Implementation of the Alternate File Format: We implemented a C program to convert FAST5 (HDF5) files into our SLOW5 format. The program also constructs a SLOW5 index as per the description in section 1.4.1. The restructured and optimised *Nanopolish* (discussed in section 1.4.3) was modified to support reading from SLOW5 format (Fig. 1.8).At the beginning of the program, the SLOW5 index is loaded onto a hash table that resides in RAM. For each *genomic read* in a batch, the start position of the corresponding SLOW5 record (file offset) and size of the record (in bytes) is obtained from the index. Then, those information for all the *genomic reads* in the batch are submitted as I/O requests. The *POSIX AIO* library in *glibc* is used for performing asynchronous I/O.

Implementation of the Multi-process Pool: The restructured and optimised *Nanopolish* was modified such that FAST5 files are loaded using a multi-process pool as per the description in Section 1.4.2 (Fig. 1.8). At the beginning of the program, K child-processes are spawned using *fork* system call. Then, during the execution of the program, the parent-process divides the batch of *genomic reads* into K parts and assigns each part to a child-process. Child-processes performs FAST5 file reading (through HDF5 library) in parallel. After completion of reading by the child-processes, the data is collected by the parent-process. The inter-process communication is implemented using *unnamed pipes* in

Table 1.2: Example of SLOW5 index

SLOW5 index		
#read_id	file_offset	rec_length
read-0	67	500000
read-1	500001	1000000
.	.	.
.	.	.
.	.	.
read-N	364459005610	1580072

Table 1.3: Dataset

ID	**Sample**	**No. of Gbases**	**No. of reads**	**Average read length**	**Max read length**	**FAST5 file size**
D1	T778	8.787	771 325	11 393	194 983	845GB

Linux.

Datasets and Computer Systems:

A representative nanopore dataset of the human genome was used for the evaluation and the details are in Table 1.3. This dataset is a complete nanopore MinION dataset of the T778 cancer cell-line [16, 44]. The computer systems used for the experiments and their specification are given in Table 1.4. Unless otherwise stated, the experiments in the chapter have been performed on system S1. System S2 was used for limited number of experiments due to the limited availability. For the experiments associated with Network File System (NFS), the NFS storage on system S3 was mounted on system S1. For NFS, default parameters for the NFS server and client in Linux were used. Note that the operating system disk cache on S3 was also cleared before any NFS experiment.

Measurements and Calculations: The measurement and calculations for our results are performed as follows.

1) The Overall execution time (wall-clock time) and *the CPU time* (user mode + kernel mode) of the program (all version shown in Fig. 1.8) were measured by running the program through *GNU time* utility in Linux.

2) The CPU utilisation percentage is computed as in equation 1.1. Note that this CPU utilisation percentage is a normalised value based on the number of data processing threads that which the program was executed with.

$$CPU\ utilisation = \frac{CPU\ time}{execution\ time \times number\ of\ threads} \times 100\% \tag{1.1}$$

Table 1.4: Computer systems

System ID	S1	S2	S3
Description	HPC with HDD RAID	HPC with SSD RAID	NFS server
CPU	2 × Intel Xeon Gold 6154	2 × Intel Xeon Gold 6148	4 × Intel Xeon X7560
CPU cores	36	40	32
RAM	384 GB	768 GB	256 GB
Disk System	12×10TB HDD drives	6×4TB NVMe drives	10×3TB HDD drives
File System	ext4	ext4	ext4
RAID config.	RAID6	RAID0	RAID5
OS	Ubuntu 18.04.3 LTS	CentOS 7.6.1810	Ubuntu 14.04.6 LTS

3) Execution time for individual components (I/O operations and data processing) in the restructured and/or optimised *Nanopolish* (three versions at the bottom of Fig. 1.8) was measured by inserting *gettimeofday* function calls into appropriate locations in the software source code. To prevent the operating system disk cache affecting the accuracy of I/O results, we cleared the disk cache (*pagecache*, *dentries* and *inodes*) each time before a program execution. Despite the effect of the hardware disk controller cache (∼8GB) being negligible due to the large dataset size (∼850GB), we still executed a mock program run prior to each experiment. Note that the operating system disk cache on S3 was also cleared before any NFS experiment.

4) Core-hours is calculated as the product of the number of processing threads employed and the number of hours (wall-clock time) spent on the job. This metric is inspired by the metric man-hours used in labour industry and is used in Cloud Computing domain to calculate the data processing cost [5]. In an ideally parallel program, this metric remains constant with the number of cores/threads.

1.5.2 Results: Alternate File Format (SLOW5)

Overall Execution Time and CPU Utilisation: The overall execution time when our proposed SLOW5 file format is used in the restructured and optimised *Nanopolish* is shown in Fig. 1.9a, while the CPU utilisation and the core-hours are depicted in Fig. 1.9b. The x-axis represents the number of data processing threads which the program was executed with. The number of I/O threads for *glibc* POSIX AIO was also set to the same number of threads as the number of data processing threads.

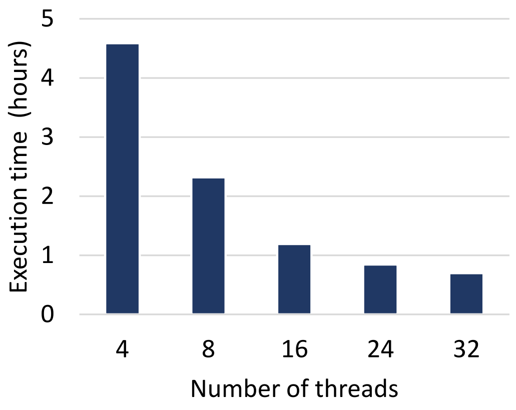
Observation-1: Performance has improved w.r.t original Nanopolish for a given number of threads. To observe this, we compare Fig. 1.1a with Fig. 1.9a. At 4 threads, execution time improved by ∼2× compared to original *Nanopolish*. At 8, 16, and 24 threads speedups of ∼2.5×, ∼4×, ∼5.5× can be observed, respectively. At 32 threads, ∼6.5× speedup is observed. In other words, speedup of our optimised version over original *Nanopolish* increases with the number of threads.

Observation-2: CPU Utilisation has improved with the number of threads when compared with original Nanopolish. Comparing Fig. 1.1b with Fig. 1.9b reveals that CPU utilisation at 4 threads improved to 99% which was 69% for original *Nanopolish*. The CPU utilisation increases to 99% from 56%, 97% from 39%, and 92% from 28%, at 8, 16 and 24 threads, respectively w.r.t original *Nanopolish*. At 32 threads, an improvement of CPU utilisation to 85% was observed which was as low as 22% for original *Nanopolish*.

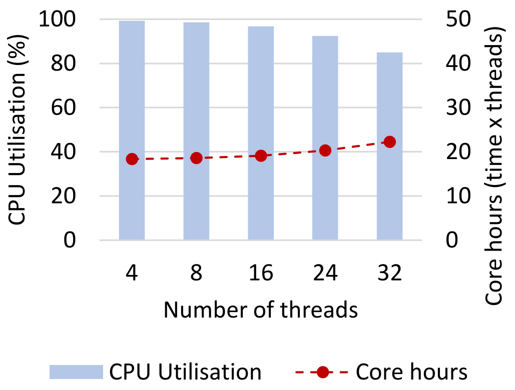
Observation-3: Performance scaling with number of threads is improved. This is evident by the core-hours plot in Fig. 1.9b, whose values are much smaller and almost constant when compared with its counter-part in Fig. 1.1b. For the original *Nanopolish*, the execution time with 32 threads improved only by ∼2.1× compared to running with 4 threads (from 9.7h to ∼4.5h). Our optimised version improved by ∼6.5× at 32 threads compared to 4 threads (∼4.5h to ∼0.7h).

I/O Time Consumption:

It was discussed previously that the identified bottleneck is due to I/O. Therefore, to get more insight into the effectiveness of our proposed solution, we plot and compare the time spent in I/O operations. Specifically, the time spent for reading nanopore raw signal data on system S1 when using SLOW5 format is compared to when using FAST5 format in Fig. 1.10a. We make the following observations from the figure: 1) there is no improvement in FAST5 access time (brown bars) despite increasing the number of threads used (due to the lock in HDF5 library); 2) in contrast, there is a significant improvement for the



(a) Execution time

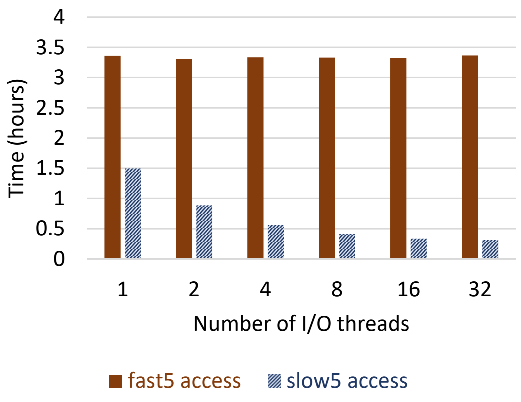


(b) CPU utilisation

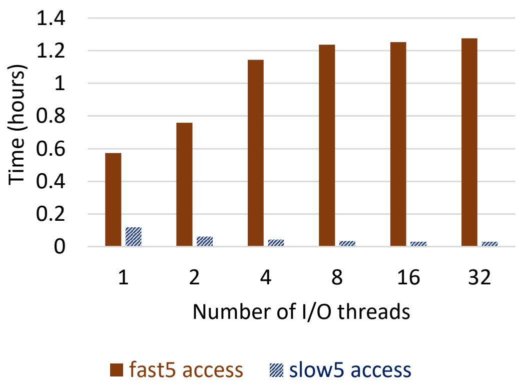
Figure 1.9: Overall execution time and CPU utilisation when SLOW5 format is used

proposed SLOW5 access time (blue bars) with the increased number of threads; and, 3) even at a single thread, the proposed SLOW5 is ∼2× times faster than FAST5, and at 32 threads the improvement of SLOW5 compared to FAST5 is ∼10×. The speed-up in I/O time for the single thread is contributed by the exploitation of locality (discussed in Section 1.4.1) and our lightweight SLOW5 access implementation.

Above experiments on system S1 demonstrated that our proposed solution effectively improves performance of *Nanopolish*. The S1 system consists of HDD RAID. Now, we demonstrate that our solution is also effective on SSD RAID using experiments on system

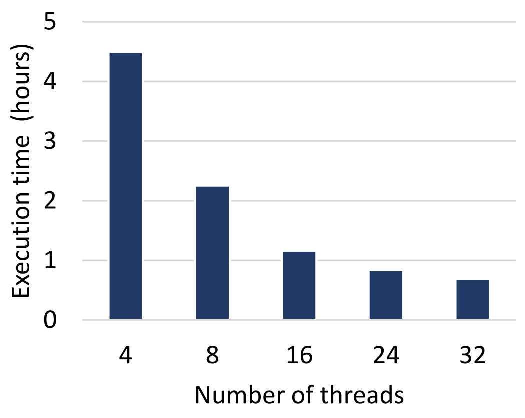


(a) On system S1: HDD RAID

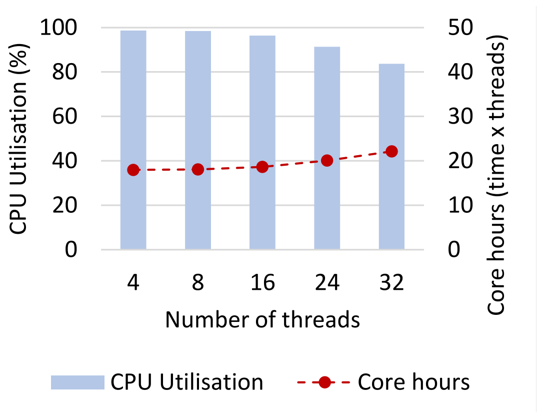


(b) On system S2: SSD RAID

Figure 1.10: Comparison of FAST5 vs SLOW5 access



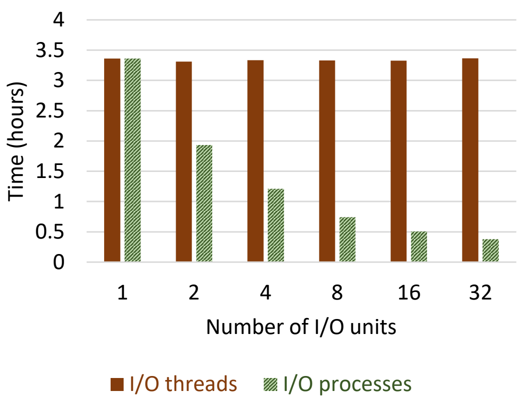
(a) Execution time



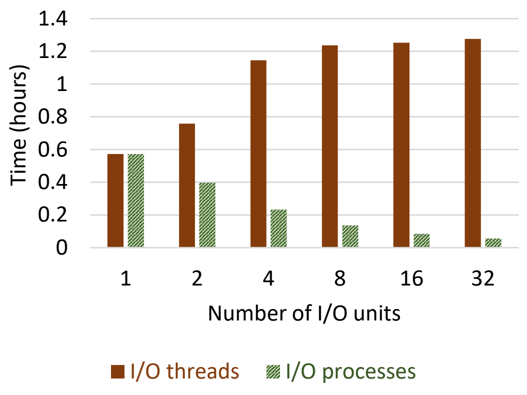
(b) CPU utilisation

Figure 1.11: Overall results for multi-process pool

S2. As discussed above, the I/O decomposition results are more insightful, therefore we present the I/O decomposition results on S2 system (SSD RAID based) for the sake of brevity of the manuscript. Fig. 1.10b shows the comparison of FAST5 access time to SLOW5 access time, where similar observations can be made. In fact, FAST5 access time (brown bars) got worse with the number of threads, whereas SLOW5 access time (blue bars) improved with the number of threads. At 32 threads SLOW5 was ∼42× faster than FAST5 on SSD RAID. Thus, our proposed solution is effective for the HDD based RAIDs as well as the SSD based RAIDs.



(a) On system S1: HDD RAID



(b) On system S2: SSD RAID

Figure 1.12: FAST5 file access using multiple I/O threads vs I/O processes

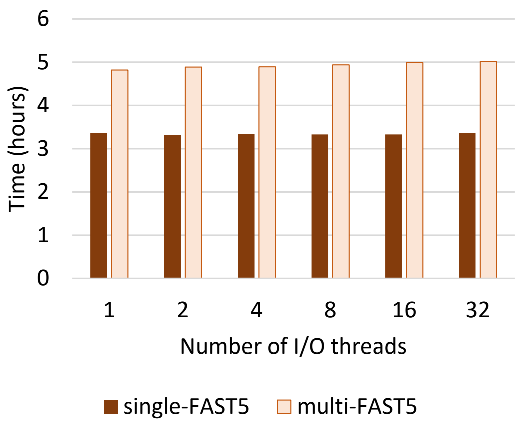
Note: The file sizes of the new SLOW5 format are comparable to the existing FAST5 format. Specifically, the dataset which was 845 GB in FAST5 format (Table 1.3), reduced to 340 GB when converted to SLOW5. The reduced size when converted to SLOW5 is due to storing global metadata in the header in SLOW5, instead of redundantly storing those for each read. SLOW5 index is quite small (47 MB) compared to gigabytes of RAM available on an HPC.

1.5.3 Results: Multi-process Pool

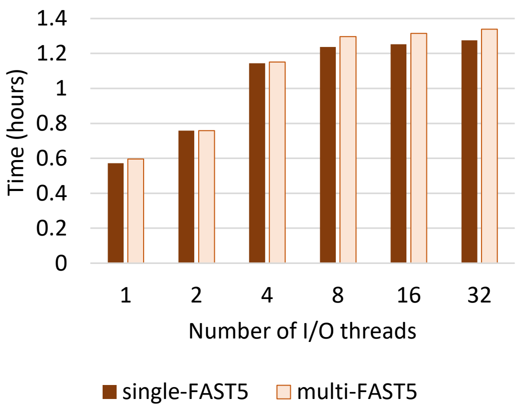
Overall Execution Time and CPU Utilisation: Overall execution time for the re-structured and optimised *Nanopolish* when a multi-process pool is used for FAST5 access is shown in Fig. 1.11a, whereas the CPU utilisation and the core-hours are depicted in Fig. 1.11b. The x-axis of the figure corresponds to the number of data processing threads which is also equal to the number of I/O processes. The results in Fig. 1.11 are similar to that of the SLOW5 solution discussed in previous subsection. The key observations in Fig. 1.11 compared to original *Nanopolish* are also similar to the first solution. These are: 1) improved performance w.r.t. the original *Nanopolish* for a given number of threads; 2) improved CPU Utilisation; and, 3) better performance scaling with increasing number of threads, as depicted by the near-flat core-hour plot.

I/O Time Consumption: Similar to the previous section, we evaluate the time spent in I/O operations. We compare the results for the multi-threaded and the multi-process based versions. The plots are presented in Fig. 1.12, with the x-axis denoting the number of processes/threads used. On HDD RAID (Fig. 1.12a), the FAST5 access time does not improve with increased I/O threads (brown bars), while it significantly improves with increased I/O processes (green bars). At 32 threads/processes the improvement was $\sim 9\times$. On SSD RAID (Fig. 1.12b), the FAST5 access time gets worse with increased I/O threads. In contrast, it significantly improves with increased I/O processes. Using 32 I/O processes is $\sim 23\times$ faster than using 32 I/O threads on SSD RAID.

In summary, using processes instead of threads for I/O operations alleviates the I/O



(a) On system S1: HDD RAID



(b) On system S2: SSD RAID

Figure 1.13: Single-FAST5 vs Multi-FAST5 using I/O threads

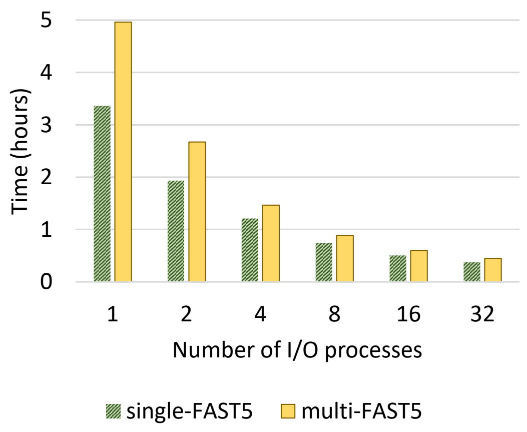
bottleneck, while using multiple-threads for data processing in a single parent-process avoids introduction of any additional significant bottlenecks, as depicted by the above results.

1.5.4 Comparison of SLOW5 to FAST5 with Multi-process Pool

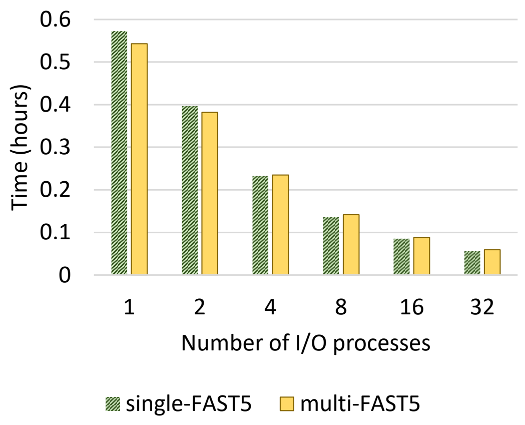
Comparing the time for SLOW5 access in Fig. 1.10 with I/O process based pool for FAST5 (Fig. 1.12) shows that SLOW5 outperforms FAST5 even when multiple I/O processes are used especially at lower number of threads/processes.

1.5.5 Comparison with Multi-FAST5 Format

ONT is recently working on a new file format: known as *multi-FAST5*. It is projected to replace the existing FAST5 format in near future. The raw signals from multiple *genomic reads* (by default 4000 *genomic reads*) are packed into a FAST5 file and such files are termed as *multi-FAST5*. Multi-FAST5 reduces the gigantic amount of small single-FAST5 files generated from a sequencing run, easing the file management (eg: copying/moving files, listing files). Multi-FAST5 files are also HDF5 files where the schema is an extended version for that of single-FAST5. Next, we demonstrate that multi-FAST5 suffers from



(a) On system S1: HDD RAID



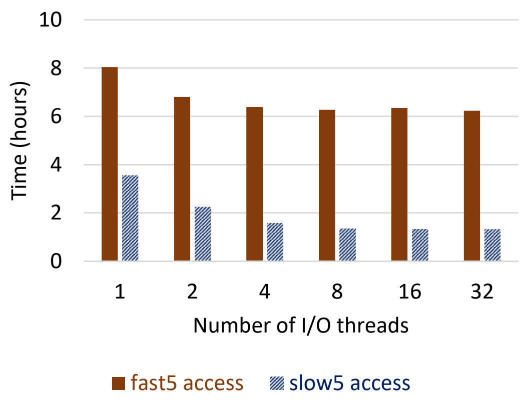
(b) On system S2: SSD RAID

Figure 1.14: Single-FAST5 vs Multi-FAST5 using I/O processes

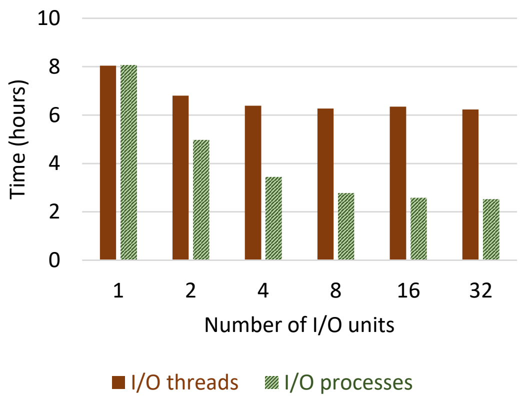
a similar bottleneck, and thus our proposed SLOW5 is superior to the new multi-FAST5 format. Moreover, our multi-process based solution is also applicable and effective for multi-FAST5 format.

Performance Bottleneck in Multi-FAST5: First, we compare the file access time in multi-FAST5 to single-FAST5 in Fig. 1.13. Unfortunately, the access time does not improve by the use of multiple I/O threads on HDD RAID, similar to single-FAST5 (Fig. 1.13a). In fact, multi-FAST5 performance is actually worse than that of single-FAST5. On SSD RAID (Fig. 1.13b), the performance of multi-FAST5 and single-FAST5 are almost the same and gets gradually worse with the number of threads.

Proposed Multi-process Solution on Multi-FAST5: Now we demonstrate that our multi-process based solution is also applicable and effectively improves the performance for the new multi-FAST5 format. The access time for Multi-FAST5 and single-FAST5 With our multi-process solution for different number of threads are in Fig. 1.14. With our solution, the trend of multi-FAST5 access time is similar to that of single-FAST5 files (on both HDD RAID and SSD RAID), that is, it gets significantly better with the number of I/O processes used. Note that when the time with single-FAST5 is compared, multi-FAST5 takes more time than single FAST5, visibly in the HDD RAID.



(a) SLOW5 performance on NFS



(b) FAST5 performance on NFS

Figure 1.15: Performance on NFS

1.5.6 On NFS

Our proposed optimisations has the potential to benefit direct execution of nanopore data analysis tools on data residing on a network attached storage. HPC cluster environments predominantly use such network attached storage in addition to local RAID systems. We demonstrate the performance of our proposed methods on NFS in Fig. 1.15.

Fig. 1.15a compares our SLOW5 format with FAST5 on NFS over multiple I/O threads. Use of multiple I/O threads for accessing FAST5 files on NFS (brown bars), slightly improves the performance upto around 4 threads (unlike previously on local RAID), which then saturates. SLOW5 access (blue bars) is much faster than FAST5. SLOW5 access time improves upto around 8 threads which then saturates.

Fig. 1.15b compares our proposed process pool based method to using multiple I/O threads. Use of multiple I/O processes (green bars) considerably improves the FAST5 access performance upto around 8 processes, which then slowly saturates, a similar trend to that with SLOW5. Comparing SLOW5 (blue bars in Fig. 1.15a) to FAST5 access using multiple I/O processes (Fig. 1.15b) shows that SLOW5 performance is superior.

1.6 Discussion

1.6.1 Other Possible Solutions

In this chapter, we presented two solutions to overcome the I/O bottleneck caused by the FAST5 file format and demonstrated their efficacy using experiments. Additionally, there are few other possible solutions to the problem, as discussed below.

Fixing HDF5 Library: As discussed above, using a new file format may not always be practical and we presented a multi-process based solution in such a scenario. Another candidate solution is to fix (optimise) the HDF5 library to be thread efficient. However, HDF5 library is a complicated library with a large code base of >300,000 lines of C code and such a fix has to be potentially done by the HDF Group [36, 37]. The HDF5 Group mentions that the future plan to implement efficient multi-threaded access is currently hindered by inadequate resources [18]. Therefore, such a fix is unlikely to happen in the near future. Moreover, there is no other alternate library to read HDF5 files [36, 37].

Naive Approaches of Multi-processing: Instead of using a process pool solely for FAST5 I/O and multi-threads for parallel data processing (as proposed in Section 1.4.2), programmers may use multi-processes for both the I/O operations and parallel data processing. This would be easier than implementing a pool of processes, however, this is only suitable for trivially parallel cases. If the application needs to share data among multiple processing units, processes are unsuitable due to the complexity that arise when performing inter-process communication.

Alternatively, the programmer may let users manually split data and launch multiple processes. Unfortunately, this method exerts additional burden on the user, i.e., custom scripts must be written for data splitting, launching data processing and concatenating the result.Moreover, this is only suitable for trivially parallel applications where data can be easily split. Also, an expensive HPC system with dozens of cores is superfluous as the user could use a cluster of low cost networked computers (chapter XX).

In summary, using processes instead of threads potentially solves the I/O bottleneck as we demonstrated in results. However, it is important to note that *processes* in an operating system are meant for isolation whereas *threads* are for sharing data. Inter-process communication requires system calls, while inter-thread communication involves sharing the same memory space. Further, processes are expensive to be spawned and are not lightweight (unlike threads). Thus, using processes as a replacement to threads makes the code relatively complicated. Therefore, we suggest that using the SLOW5 format is a superior solution than the multi-processes based solution.

1.6.2 Future Directions

As shown in section 1.5.2, SLOW5 file size is smaller than for FAST5 due to the efficient storage of metadata. SLOW5 file size can potentially be further reduced by using a binary encoding instead of ASCII and/or by applying block compression techniques such as BGZF that still allows random access [26]. Having both ASCII and binary formats is useful, where the former is human readable and the latter is space efficient. In fact, gold standard file formats in genomics such as SAM and VCF that are in ASCII have their binary counterparts BAM and BCF.

After applying our proposed optimisations proposed in this chapter, the next bottleneck in *Nanopolish* could be the FASTA access (random access to reference genome) which is performed using the *faidx* component in *htslib* library. This *faidx* is not currently thread-safe and thus only single threaded access is possible. However, FASTA is a simple ASCII based format and thus extending *faidx* for thread efficiency is feasible as future work.

1.6.3 Potential Impact on other Toolkits and Domains

It is likely that the identified limitation in HDF5 libary is a primary bottleneck in several other nanopore software toolkits, which also use the HDF5 library such as *Tombo* [43], *NanoMod* [28] and *SquiggleKit* [12]. Thus, our proposed optimisations are potentially useful in such toolkits. Our work may also guide nanopore software developers to avoid the identified bottleneck in future. Furthermore, HDF5 is also used in other engineering domains such as physics, astronomy, weather forecasting [13]. Therefore, we believe that our work will inspire optimisations in those domains.

1.7 Summary

Latest nanopore sequence analysis software tools demand substantial computing power and require HPC systems to target quick turn-around of results. In this chapter, we demonstrated with an example that nanopore software fail to take maximal advantage of the computing power offered by many-core processors in HPC systems, despite multi-threaded implementation. To address this problem, we presented a systematic experimental analysis to identify potential performance bottlenecks in nanopore software tools for running on many-core CPUs. We identified that the bottleneck is caused by inefficient file I/O associated with the HDF5 library used for loading nanopore raw data. The inefficiency in file I/O in HDF5 is due to a global lock which limits multiple threads requesting file accesses in parallel. Then, we proposed multiple optimisations to alleviate the bottleneck. We proposed a new file format that facilitates efficient file access using multiple threads. For the scenarios where the original format must be used, we presented a multi-process based solution. Thus, our proposed optimisations can be used as an alternative, or alongside the existing file-format. Our experiments demonstrated that our optimisations not only enable improved performance for a given number of threads ($\sim$2$\times$ for 4 threads and $\sim$6.5$\times$ for 32 threads), but also enable improved CPU utilisation (from 69% to 99% for 4 cores and from 22% to 85% for 32 threads) when compared to original *Nanopolish*. Consequently, improved performance scaling with the number of threads was also achieved ($\sim$6.5$\times$ for 4 vs. 32 threads).

References

[1] "Re: [patch 09/13] aio: add support for async openat()," Jan. 2016. [Online]. Available: https://lwn.net/Articles/671657/

[2] R. Al-Ali, N. Kathiresan, M. El Anbari, E. R. Schendel, and T. A. Zaid, "Workflow optimization of performance and quality of service for bioinformatics application in high performance computing," *Journal of Computational Science*, vol. 15, pp. 3–10, 2016.

[3] S. L. Amarasinghe, S. Su, X. Dong, L. Zappia, M. E. Ritchie, and Q. Gouil, "Opportunities and challenges in long-read sequencing data analysis," *Genome biology*, vol. 21, no. 1, pp. 1–16, 2020.

[4] A. S. Bland, W. Joubert, D. E. Maxwell, N. Podhorszki, J. H. Rogers, and A. N. Tharrington, "Contemporary high performance computing from petascale toward exascale," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States); Center for . . . , Tech. Rep., 2013.

[5] K. Chan, "What are core-hours? how are they estimated?" 2019. [Online]. Available: https://support.onscale.com/hc/en-us/articles/360013402431-What-are-Core-Hours-How-are-they-estimated-

[6] J. Corbet, "Fixing asynchronous i/o, again," Jan. 2016. [Online]. Available: https://lwn.net/Articles/671649/

[7] ——, “Toward non-blocking asynchronous i/o,” May 2017. [Online]. Available: https://lwn.net/Articles/724198/

[8] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry *et al.*, “The variant call format and vcftools,” *Bioinformatics*, vol. 27, no. 15, pp. 2156–2158, 2011.

[9] J. de Jesus, C. I. SC, F. Salles, E. Manulli, D. da Silva, T. de Paiva, M. Pinho, A. Afonso, A. Mathias, L. Prado *et al.*, “First cases of coronavirus disease (covid-19) in brazil,” *South America (2 genomes, 3rd March 2020)(http://virological. org/t/first-cases-ofcoronavirus-disease-covid-19-in-brazil-south-america-2-genomes-3rd-march-2020/409, Virological, 2020)*, 2020.

[10] S. R. Eddy, “Accelerated profile hmm searches,” *PLoS computational biology*, vol. 7, no. 10, 2011.

[11] B. Fenski, “htop(1) - linux man page.” [Online]. Available: https://linux.die.net/man/1/htop

[12] J. M. Ferguson and M. A. Smith, “Squigglekit: A toolkit for manipulating nanopore signal data,” *Bioinformatics*, vol. 35, no. 24, pp. 5372–5373, 2019.

[13] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, “An overview of the hdf5 technology suite and its applications,” in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 2011, pp. 36–47.

[14] M. H.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney, “Efficient storage of high throughput dna sequencing data using reference-based compression,” *Genome research*, vol. 21, no. 5, pp. 734–740, 2011.

[15] H. Gamaarachchi, A. Bayat, B. Gaeta, and S. Parameswaran, “Cache Friendly Optimisation of de Bruijn Graph based Local Re-assembly in Variant Calling,” *IEEE/ACM transactions on computational biology and bioinformatics*, 2018.

[16] D. W. Garsed, O. J. Marshall, V. D. Corbin, A. Hsu, L. Di Stefano, J. Schröder, J. Li, Z.-P. Feng, B. W. Kim, M. Kowarsky *et al.*, "The architecture and evolution of cancer neochromosomes," *Cancer Cell*, vol. 26, no. 5, pp. 653–667, 2014.

[17] HDFgroup, "Hdf5 specifications." [Online]. Available: https://support.hdfgroup.org/HDF5/doc/Specs.html

[18] hdfgroup.org, "Questions about thread-safety and concurrent access." [Online]. Available: https://portal.hdfgroup.org/display/knowledge/Questions+about+thread-safety+and+concurrent+access

[19] M. Jain, S. Koren, K. H. Miga, J. Quick, A. C. Rand, T. A. Sasani, J. R. Tyson, A. D. Beggs, A. T. Dilthey, I. T. Fiddes, and others, "Nanopore sequencing and assembly of a human genome with ultra-long reads," *Nature biotechnology*, vol. 36, no. 4, p. 338, 2018.

[20] M. Jain, H. E. Olsen, B. Paten, and M. Akeson, "The oxford nanopore minion: delivery of nanopore sequencing to the genomics community," *Genome biology*, vol. 17, no. 1, p. 239, 2016.

[21] N. Kathiresan, R. Al-Ali, P. V. Jithesh, T. AbuZaid, R. Temanni, and A. Ptitsyn, "Optimization of data-intensive next generation sequencing in high performance computing," in *2015 IEEE 15th International Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE, 2015, pp. 1–6.

[22] N. Kathiresan, R. Temanni, H. Almabrazi, N. Syed, P. V. Jithesh, and R. Al-Ali, "Accelerating next generation sequencing data analysis with system level optimizations," *Scientific reports*, vol. 7, no. 1, pp. 1–11, 2017.

[23] A. Kawalia, S. Motameny, S. Wonczak, H. Thiele, L. Nieroda, K. Jabbari, S. Borowski, V. Sinha, W. Gunia, U. Lang *et al.*, "Leveraging the power of high performance computing for next generation sequencing data analysis: tricks and twists from a high throughput exome workflow," *PloS one*, vol. 10, no. 5, p. e0126321, 2015.

[24] M. Kerrisk, "Aio(7) linux programmer's manual," Mar. 2019. [Online]. Available: http://man7.org/linux/man-pages/man7/aio.7.html

[25] M. Leija-Salazar, F. J. Sedlazeck, M. Toffoli, S. Mullin, K. Mokretar, M. Athanasopoulou, A. Donald, R. Sharma, D. Hughes, A. H. Schapira *et al.*, "Evaluation of the detection of gba missense mutations and other variants using the oxford nanopore minion," *Molecular genetics & genomic medicine*, vol. 7, no. 3, p. e564, 2019.

[26] H. Li, "Tabix: fast retrieval of sequence features from generic tab-delimited files," *Bioinformatics*, vol. 27, no. 5, pp. 718–719, 2011.

[27] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, "The sequence alignment/map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[28] Q. Liu, D. C. Georgieva, D. Egli, and K. Wang, "Nanomod: a computational tool to detect dna modifications using nanopore long-read sequencing data," *BMC genomics*, vol. 20, no. 1, p. 78, 2019.

[29] N. J. Loman, J. Quick, and J. T. Simpson, "A complete bacterial genome assembled de novo using only nanopore sequencing data," *Nature methods*, vol. 12, no. 8, p. 733, 2015.

[30] H. Lu, F. Giordano, and Z. Ning, "Oxford nanopore minion sequencing and genome assembly," *Genomics, proteomics & bioinformatics*, vol. 14, no. 5, pp. 265–279, 2016.

[31] M. Majkowski, "io_submit: The epoll alternative you've never heard about," Jan. 2019. [Online]. Available: https://blog.cloudflare.com/io_submit-the-epoll-alternative-youve-never-heard-about/

[32] J. E. Moyer, "libaio." [Online]. Available: https://pagure.io/libaio

[33] nanoporetech, "Jared simpson: Signal analysis using nanopolish," 2019. [Online]. Available: https://nanoporetech.com/resource-centre/jared-simpson-signal-analysis-using-nanopolish

[34] ——, "Oxford nanopore technologies fast5 api software," 2019. [Online]. Available: https://github.com/nanoporetech/ont_fast5_api

[35] M. Riba, C. Sala, D. Toniolo, and G. Tonon, "Big data in medicine, the present and hopefully the future," *Frontiers in Medicine*, vol. 6, 2019.

[36] C. Rossant, "Moving away from hdf5," 2016. [Online]. Available: https://cyrille.rossant.net/moving-away-hdf5/

[37] ——, "Should you use hdf5?" 2016. [Online]. Available: https://cyrille.rossant.net/should-you-use-hdf5/

[38] B. Schmidt and A. Hildebrandt, "Next-generation sequencing: big data meets high performance computing," *Drug discovery today*, vol. 22, no. 4, pp. 712–717, 2017.

[39] L. Schmitt, "From computational genomics to precision medicine," 2016. [Online]. Available: https://cs.illinois.edu/news/computational-genomics-precision-medicine

[40] M. Scholz, D. V. Ward, E. Pasolli, T. Tolio, M. Zolfo, F. Asnicar, D. T. Truong, A. Tett, A. L. Morrow, and N. Segata, "Strain-level microbial epidemiology and population genomics from shotgun metagenomics," *Nature methods*, vol. 13, no. 5, pp. 435–438, 2016.

[41] J. Simpson, "Nanopolish," 2016. [Online]. Available: https://github.com/jts/nanopolish/

[42] J. T. Simpson, R. E. Workman, P. Zuzarte, M. David, L. Dursi, and W. Timp, "Detecting dna cytosine methylation using nanopore sequencing," *Nature methods*, vol. 14, no. 4, p. 407, 2017.

[43] M. H. Stoiber, J. Quick, R. Egan, J. E. Lee, S. E. Celniker, R. Neely, N. Loman, L. Pennacchio, and J. B. Brown, "De novo identification of dna modifications enabled by genome-guided nanopore signal processing," *BioRxiv*, p. 094672, 2016.

[44] E. W. Stratford, R. Castro, J. Daffinrud, M. Skårn, S. Lauvrak, E. Munthe, and O. Myklebost, "Characterization of liposarcoma cell lines for preclinical and biological studies," *Sarcoma*, vol. 2012, 2012.

[45] unknown, “Io_setup(2) linux programmer’s manual,” Sep. 2019. [Online]. Available: http://man7.org/linux/man-pages/man2/io_setup.2.html