Engineering a high-performance SNP detection pipeline

Vasudevan Rengasamy Kamesh Madduri
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA, USA
Email: vxr162@psu.edu, madduri@cse.psu.edu

April 18, 2015

Abstract

We present SPRITE, a bioinformatic data analysis pipeline for detecting single nucleotide polymorphisms (SNPs) in the human genome. A SNP detection pipeline for next-generation sequencing data uses several software tools, including tools for read preprocessing, read alignment, and SNP calling. We target end-to-end scalability and I/O efficiency in SPRITE by merging tools in this pipeline and eliminating redundancies. A key component of our optimized pipeline is PARSNIP, a new parallel method and software tool for SNP calling. Experimental results on synthetic and real data sets indicate that the quality of results achieved by PARSNIP is comparable to state-of-the-art SNP calling software.

1 Introduction

In this work, we consider the pervasive *genetic variation detection* workflow in biomedical informatics. The goal of this workflow is to automatically determine genetic variations present in the genome of an individual (called the *donor*), by comparing the donor genome to a *reference* genome. Among structural variations, the most well-studied are *Single Nucleotide Polymorphisms*, or SNPs. SNPs are nucleotide differences at a single position. Detecting these SNPs using current state-of-the-art tools can take more than a day of sequential compute time. We explain why this is the case in the next section. Our goal is to design parallel algorithms and identify optimizations to accelerate end-to-end performance of a SNP detection (or *SNP calling*) pipeline.

Data reduction by two or three orders of magnitude is a common feature in most genomic data analysis pipelines. The raw data, or data originating from the DNA sequencers, is usually in plain text format, compressed using gzip, and can be up to several 100 GB in size. Depending on the biological problem of interest, this data is processed in conjunction with additional data sets (which can be up to 1 TB), using a combination of analysis tools.

The output produced by the SNP detection software is much smaller and usually less than 1 GB. The entire workflow is implemented as a script that executes different programs, and is either run on a dedicated server or submitted to the job scheduler of a shared cluster. A modular approach to pipeline construction is a poor match to current workstations, and also results in very little overall speedup on supercomputers. For instance, the SMaSH benchmarking effort [34] documents the performance of several variant detection pipelines on representative genomic data sets and a large shared-memory server. Running times are reported to be in the order of tens of hours to multiple days with state-of-the-art software, achieving a very small fraction of peak system performance.

There is a lot of current research on new algorithms and optimizations for various stages of this pipeline (reviewed in the next section). Our contributions in this paper follow the same theme. Additionally, given that our end-goal is to identify SNPs, we explore possible ways for reduce unnecessary I/O, and design parallelization strategies with input data from current sequencing technologies in mind. The current state-of-the-art deployments of this pipeline use sequential software for certain stages, which turn out to be significant bottlenecks. We have focused our effort on the primary time-consuming steps in this paper and have designed new parallel algorithms and software (PRUNE, SAMPA, PARSNIP) for these steps. The end-to-end running time of our pipeline on 16 nodes of the NERSC Edison supercomputer is around 34 minutes for a realistic input data set. We also show that the resulting SNP detection quality is comparable to a pipeline constructed using state-of-the-art tools. The end-to-end time of the traditional pipeline is 28 hours, and so we achieve a nearly 49.5× speedup.

2 SNP Pipeline Overview

The genome sequences of any two (human) individuals are highly similar. However, the small percentage of genetic variation is believed to have important biological and medical implications. Identifying an individual's single nucleotide genetic variants has become a standard first step in many biological and biomedical applications. In this section, we describe the workflow to detect these variants and prior approaches to exploit parallelism.

The output of a DNA sequencer is a set of *reads*. A read is a short segment of the genome whose sequence is known, but whose location in the genome is not known. As sequencing data becomes ubiquitous, standard workflows for its analysis are becoming commonly used. The basic idea is to align the reads against the human reference genome and identify genomic locations where aligned reads show a variation in the nucleotide.

When sequence data is initially generated, it is typically stored on disk in multiple gzipped FASTQ (FQ) files. FQ is a text file format that stores each read using four lines, including the nucleotide sequence and the quality score of each nucleotide. FQ files are immediately compressed as they come off the sequencer. A single sequencing experiment will generate multiple fq.gz files, with each file containing its own set of reads. FQ files are then put through several preprocessing filters, including the removal of low quality reads and *trimming reads* to remove low quality nucleotides at the end. These filters are run separately on each compressed FQ file and the results are then written back to disk in the same format. The

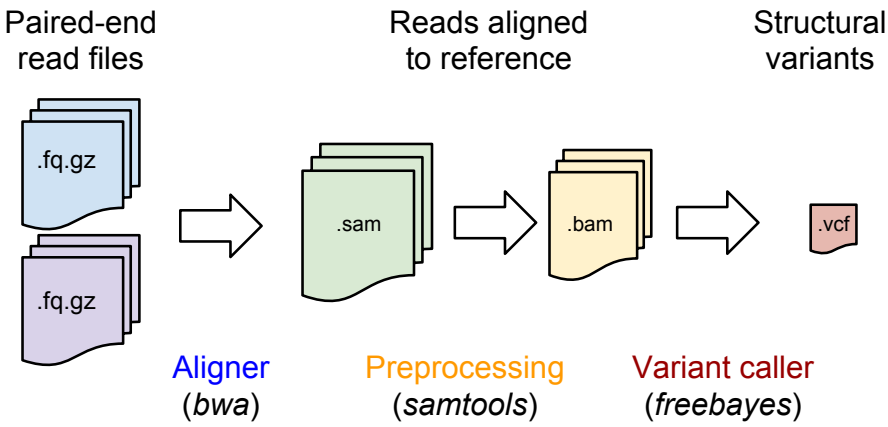

Figure 1: The three key stages in a genetic variant detection pipeline. We indicate popular tools used for each stage and the output format of files generated.

FQ files are then aligned against the human reference using one of several software tools. Alignment is the most computationally intensive task in the workflow. The output of the alignment is in the Sequence Alignment/Map (SAM) format [18], which is a text file with a single line representing the record for one alignment. This includes a read name, nucleotide sequence, the aligned location in the reference, and meta-information about the alignment. SAM has a corresponding binary file format, called a Binary Alignment/Map (BAM) format. BAM roughly stores the same information as SAM, but using fixed-length binary encoded records. The aligner outputs a separate SAM file for each fq.gz input file, and this file is often converted to a BAM file and written to disk. The BAM files are then each sorted according to the alignment position and written to disk. The BAM files may then be further merged and indexed. The search index is based on a binning scheme that is a representation of an R-tree [13,18]. Finally, a variant caller takes the single BAM file and generates a list of variants. Roughly, this is done by a left-to-right scan of the BAM file. Locations where all alignments do not agree with the reference nucleotide are statistically analyzed to determine if the alternate nucleotide is due to a variant in the donor, or due to a sequencing error. The output of this step is a text file in the Variant Calling Format (VCF), containing a single line for every variant. At this stage, the file size is very small (< 1 GB). While there are further filtering steps that are often performed, we do not include them here because they represent only a negligible contribution to the overall resource requirements of the workflow. The pipeline stages and the corresponding file sizes are indicated in Figure 1 and Table 1, respectively. Note that we indicate multiple SAM to BAM processing stages as *preprocessing* in the figure, as they are only preprocessing stages for the variant caller and have no significance in isolation.

The sequence alignment stage is considered the most compute-intensive task in the genetic variant detection workflow. There are several approaches to aligning reads against a reference

Table 1: The output format and typical file sizes of various steps in a SNP calling pipeline.

Step	**Output Format**	**Typical Size**
Read preprocessing	multiple fq.gz	240 GB
Alignment	multiple SAM	130 GB
Binary encode, sort, merge	multiple BAM	130 GB
Variant calling	single VCF	< 1 GB

genome. Usually, an alignment algorithm uses an index of the reference genome. The FM-index [9] is the index of choice for the most popular aligners of sequencing data [15, 17, 20]. It can be used to find if a query substring occurs in the reference in time independent of the length of the reference. The FM-index is a full-text index which is based on the Burrows-Wheeler transform [1, 5] developed for text compression. The implementation of the FM-index stores the Huffman-coded Burrows-Wheeler transform of the reference string along with two associated arrays and some neglible space overhead. In addition to alignment, it is used in other bioinformatic areas such as *de novo* genome assembly [16,32,33] and sequencing data analysis [31]. An alternate to the FM-index is hash-based index. In this approach, a hash table is created mapping all small strings (called seeds) to their locations in the reference (or, sometimes, in the reads). While initially popular [19, 30], hash-based approaches have largely been replaced by the FM-index approach due to its superior performance.

There are two main tasks that an alignment software package performs: (1) constructing an index given the set of reference sequences (a one-time operation in most pipelines), and (2) aligning reads to the reference sequences by accessing the index. While there is some work on parallelizing FM-index construction and its underlying subroutines [8], most current efforts on parallelizing alignment software focus on the alignment step, and prebuilt indexes are available for commonly-used reference sequences. In a shared-memory environment, the index is loaded into memory and reads are partitioned across multiple threads. This is the predominant parallelization strategy employed in most x86-based software, and the latest versions of BWA and Bowtie 2 already support multithreading. There is however a lot of room for improvement in these multithreaded implementations. Zhang et al. [37] profile performance of BWA on an Intel Sandy Bridge-based system and observe that the dominant subroutine is the backward search step. This step involves a tree traversal and results in significant cache and TLB misses. Computational load imbalances due to coarse-grained read partitioning can also lead to performance slowdown. Martínez et al. propose a dynamic read reordering scheme [25] for RNA sequence matching. There are also several recently-developed CUDA implementations for NVIDIA GPUs, such as SOAP3 [21], BarraCUDA [14], and CUSHAW [23]. NVIDIA has released the NVBIO software package [28], which provides CUDA subroutines that can be reused by bioinformatics tasks, and also includes a CUDA port of BWA. CUSHAW2-GPU [22] is a notable recent aligner that partitions reads across the host and the GPU. There are also aligners using FPGAs [24, 35]. Convey computer, using their HC-Series hybrid core hardware and tuned bioinformatics software, report what we believe are the current-best speedup results (15–20×) over BWA on multicore servers [6].

There are also parallel hash-based aligners such as SNAP [36], which are significantly faster than BWA and Bowtie 2. However, execution time is only one of the many criteria used to assess alignment software. Hatem et al. [12] compare several different x86 aligners using criteria such as mapping quality, support for gapped alignments, memory footprint, support for inexact matches, in addition to execution time. Their main conclusion is that there is no clear winner. Alignment is thus still an open problem and an active topic of current research. Note however that the relative proportions of time spent in various pipeline tasks depend on the hardware configuration, the software tools used for the tasks and the level of optimization they employ, and the specific problem solved using the pipeline. Amdahl's law will drastically limit overall speedups for any efforts that focus exclusively on the alignment stage.

Parallel I/O optimization, in the context of the genetic variant pipeline, has not received much attention. Recent work that is most relevant to the proposed research is *SeqInCloud* [26], a Windows Azure implementation of a genomics pipeline that comprises BWA and GATK. This pipeline implementation uses CRAM [10], a lossless alignment output alternative to the BAM file format, and reduces storage costs on this Azure deployment. Two popular approaches to support parallel sequence search are query partitioning and reference partitioning. Query partitioning implies splitting up the reads among multiple tasks and replicating the reference database. A more scalable strategy, used in tools such as mpiBLAST [7], is reference partitioning. Here, each processor searches whole query sequences against a fraction of the reference. Zhu et al. [38] investigate I/O behavior with mpiBLAST for different I/O access schemes, and study the performance impact of the degree of I/O parallelism and the contention of I/O resources on parallel BLAST. It is also one of the first papers to examine the impact of I/O contention on genomics pipelines. BLAST [2] is a classical alignment tool, albeit it is not suitable for the type of data used in variant detection workflows. Because of its popularity in other applications, however, it has been a focus of I/O optimization efforts.

3 The SPRITE Analysis Pipeline

We now present our new SNP calling pipeline called SPRITE and explain the rationale behind creating new modules. SPRITE is made up of three tools, PRUNE, SAMPA, and PARSNIP, roughly corresponding to the three key steps of alignment, preprocessing, and variant detection, indicated in the previous section. Figure 2 shows the pipeline stages. Note that the intermediate file formats have changed in this case, but we still work with the restriction of using FQ as input and generating VCF-formatted output files. Our pipeline is designed to exploit both shared- and distributed-memory parallelism as much as possible. We use the MPI library for inter-process parallelism and POSIX threads in shared memory. We use the terms *MPI task* and *MPI process* interchangeably in discussion below.

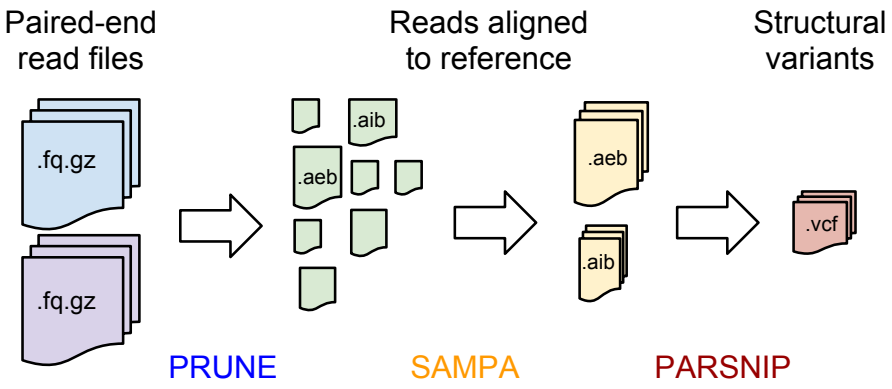

Figure 2: SPRITE: Our new HPC pipeline for SNP detection. Note that the intermediate formats have changed.

3.1 Alignment using PRUNE

For the alignment stage, any state-of-the art aligner which takes FASTA/FASTQ read files as input can be used. Most current alignment software packages have some support for single-node parallelism. We did not want to reinvent the wheel for this critical step of the pipeline, and so we evaluated performance of algorithms in several current aligners, such as BWA, Bowtie, SOAP, and SNAP. Based on some preliminary performance and quality results (some of which are discussed in the next section), we decided to use BWA (specifically, the BWA-MEM algorithm) for alignment.

Algorithm 1: Hybrid parallelization strategy in PRUNE.

Input: Read file(s) of size S, P MPI tasks, t threads per task
for *each process* $1 \leq p \leq P$ *in parallel* **do**
  startOffset$_p \leftarrow p \times \frac{S}{P}$;
  if $p > 1$ **then**
    Increment startOffset$_p$ until beginning of next read;
    send startOffset$_p$ to process $p - 1$;
  Barrier();
  if $p < P$ **then**
    endOffset$_p \leftarrow$ startOffset$_{p+1}$;
  while *startOffset*$_p <$ *endOffset*$_p$ **do**
    Read a section of reads starting at startOffset$_p$;
    Process chunk using t threads;
    startOffset$_p \leftarrow$ startOffset$_p$ + chunkSize;

BWA only supports shared-memory parallelism via POSIX threads. One simple and commonly-used approach to distributed-memory parallelization is read partitioning and reference sequence replication. For instance, the MPI-based tool pMap [4] provides multi-node parallelism for different aligner software using reference replication. The tool divides work among multiple tasks by assigning a portion of read files to each MPI task. Our overall parallelization strategy is similar to pMap. Instead of just calling the BWA binary as-is on a smaller read file (like what is done in pMap), we modify the latest source code of BWA. These changes were motivated by the following reasons:

- The output of BWA, like other aligners, is a SAM file. SAM is a tab-separated text file briefly described in the previous section. Not all fields present in the SAM file are required for variant calling.
- The textual records should be parsed to obtain usable field values by downstream tools in order to reorder and index the alignments.

Existing pipelines require that the SAM files(s) be converted to BAM file(s) to ease processing the alignment records. We shift some of the burden of downstream SAM file processing to the alignment stage itself, and this has led to make the following modifications to BWA.

- We exploit hybrid MPI and pthread parallelism to enable BWA to scale across multiple nodes.
- We generate separate alignment files for each *contig* (or contiguous sequences without gaps) in the reference sequence. The reference comprises several large contigs. A chromosome, for instance, could be a contig. In reality, there are about a hundred contigs of different lengths in the reference.
- We generate alignment records for the cases of *full* and *partial match* cases of alignments. We directly create binary output files of alignment records with condensed information.

We refer to our current implementation of this aligner with the above changes as PRUNE. Some key implementation details of these modifications are discussed in the following subsections. Note that we can also generate full SAM files with PRUNE, if necessary.

3.1.1 Hybrid MPI and POSIX thread parallelism

In BWA-MEM, the number of threads to be created can be specified using the command line option *-t*. BWA proceeds by reading a chunk of reads, which are processed by the threads in parallel, and a chunk of alignment records are written to the output SAM file. In our MPI parallelization, each task is assigned a fixed number of reads to align. A task in-turn chunks these reads and assigns to individual threads to process them.

Algorithm 1 summarizes our hybrid parallelization scheme. The input read files can be in compressed or uncompressed format, and hence BWA uses the *zlib* function *gzread* for

reading these files. If the read files are uncompressed, dividing the files among MPI processes is a straightforward task, as the initial startOffset$_p$ values that are calculated approximately divide the files equally among processes. But it is not so easy when the files are compressed, since it is not possible to obtain the uncompressed file size directly without inflating it. It is however possible to obtain the current uncompressed file offset (current file offset in uncompressed file) in the compressed file using the *gztell* function. Also, we can obtain the file descriptor associated with a compressed file. Using these information, we get the initial uncompressed file offset corresponding to compressed read files, as given by Algorithm 2. This algorithm works by incrementing the uncompressed file offset until the corresponding compressed file offset reaches the desired value. If BWA is used to align two paired-end read files, we obtain the offset for the first read file using this algorithm, and use the same offset for the second read file. This method does not work if the two read files have different read lengths. This, for instance, is the case for the NA18507 sampled human data set considered in our experimental study. We do not use paired-end alignment for this data set and just treat it as a single-end read data set. We will handle this case of different read lengths in future work.

Algorithm 2: Algorithm to find initial startOffset for a process in a compressed read file.

Input: Process p, Total number of MPI processes P, Size of compressed file S, File descriptor of compressed read file *fd* and corresponding *gzFile* handle *fp*.
Output: Initial startOffset$_p$
targetFileOffset ← p × S / P;
currentCompressedFileOffset ← 0;
while *currentCompressedFileOffset* < *targetFileOffset* **do**
    Seek to uncompressed file offset
    currentUncompressedFileOffset + targetFileOffset − currentCompressedFileOffset
    using *gzSeek*;
    currentUncompressedFileOffset ← current uncompressed file offset (obtained by *gztell(fp)* function call);
    currentCompressedFileOffset ← current compressed file offset (obtained by *lseek(fd, 0, SEEK_CUR)* function call);
startOffset$_p$ ← currentUncompressedFileOffset;

3.1.2 Separate Alignment Files

We also have modified BWA to create separate files for each contig, in order to reduce the per-process memory requirement in the subsequent parallel sort (SAMPA) and SNP calling (PARSNIP) stages. The separate files also help in partitioning work in an efficient manner in the upcoming stages. For each contig, a separate file is created for records with the CIGAR string ¡*L*¿*M*, where L is the length of the read (Full match alignment records) and records with other CIGAR strings (Partial match alignment records). Full match alignment records

doesn't have any insertions, deletions or soft/hard clipping, and as a result, are easier to process downstream.

We list the fields present in full and partial match records in Table 2. The fields alignment position, quality, and aligned read sequence appear in both full and partial match alignment record types. Read length is used in the full match record instead of CIGAR string in partial match record. The access methods mentioned in the table are used in the PARSNIP algorithm.

The total number of full and partial match alignment files is upper bounded by $2P \times$ num_contigs, since each MPI process creates a separate full and partial match file for each contig. Unaligned reads and zero quality alignments are excluded while creating full and partial match alignment files. This also reduces the input size for subsequent stages.

Table 2: Fields stored in our modified *full match* and *partial match* alignment records.

Field	**Full Match** record	**Part. Match** record	Access **Method**
Position	✓	✓	AlignPosition()
Quality	✓	✓	
Sequence	✓	✓	Sequence()
ReadLength	✓	×	ReadLength()
#CigarOps	×	✓	
CigarStr	×	✓	

3.2 SAMPA parallel sort

The next step is to sort the alignment files based on the alignment position. The primary goal of sorting is to improve alignment record access locality for the SNP calling step. We can now sort records of each contig independently, as they are written to separate files. This is not the case with BWA, and so the sort and index steps using the samtools package are extremely expensive.

We use MPI-based parallelization for this step, with each process sorting alignment output files corresponding to a set of contigs. Our algorithm partitions alignment files among MPI tasks such that

- Each MPI task gets approximately an equal number of alignment records to process
- All alignment files corresponding to a contig are processed by a single MPI task.

Algorithm 3 describes our counting sort-based algorithm for a given MPI task p. The time taken per MPI task by this sort step is linear in the number of alignment records assigned to the MPI task. In the first stage, the algorithm reads the alignment files and counts the number of reads or records aligned to each position of a reference contig. Then, we calculate a prefix sum of the count array. In the second stage, the position of each alignment record

in the output file is computed using the calculated prefix sum and the records are written to an output buffer at that position. The in-memory sorted output buffer is then written to disk. In comparison to the sort in samtools, our approach is entirely in-memory and the positions to be sorted span the length of a contig (as opposed to the entire reference). Hence it is significantly faster.

Algorithm 3: The SAMPA algorithm for sorting alignment records.

Input: Process p, Set of contigs and alignment files assigned to p
for *each contig c of length l assigned to p* **do**
  Construct the array COUNT[1..l] where COUNT(i) gives the number of reads aligned at position i in c;
  Construct the array SCAN[1..l] where SCAN(i) gives the sum of first i elements in COUNT array;
  for *each alignment record* r $\in$ *contig* c **do**
    Write r at position SCAN(p) of sorted output file where $p = AlignPosition(r)$;
    SCAN(p)$\leftarrow$ SCAN(p) + 1;

If alignment output files corresponding to a single contig are distributed across different MPI tasks, the sorted output of all such tasks have to be merged in a subsequent step. We avoid this by assigning each contig to a single task. However, doing so limits scalability, as the maximum number of MPI tasks is limited by the number of contigs in a reference genome.

Hybrid parallelism, where all alignment files corresponding to a contig are still assigned to the same MPI task, but a group of threads could work on sorting records belonging to a contig, could further increase scalability. There may however be contention among threads in accessing COUNT and SCAN arrays. We plan to implement and analyze performance of a multithreaded approach in future.

3.3 PARSNIP

We are now ready to present our PARSNIP counting-based SNP detection algorithm and optimizations. The inputs to this algorithm are **sorted binary alignment records** (consisting of full and partial read matches for each reference contig; files are created using SAMPA) and the **reference FA file**.

The output of the PARSNIP algorithm is a file containing SNP records in the VCF format. Each SNP record produced by our algorithm has the following fields:

- **Contig**: the reference contig identifier
- **Pos**: 1-indexed position in the reference sequence
- **RefBase**: the base at *Pos* in the reference
- **AltBase**: the SNP call

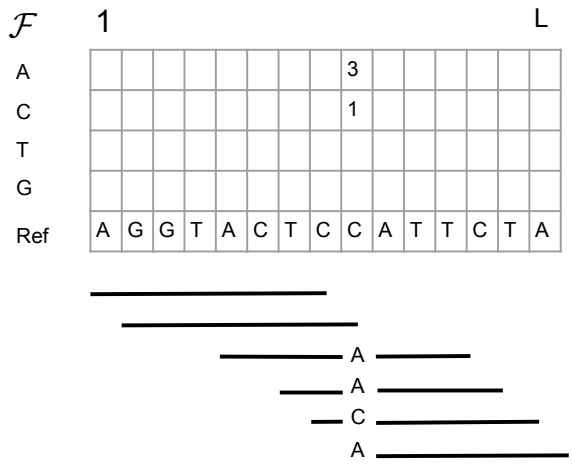

Figure 3: The organization of the frequency table $\mathcal{F}$ and the way it is updated. L is the length of the reference sequence.

- **Depth**: the total number of (non-zero quality) alignments at this position
- **AlleleCount**: the total occurrences of alleles (A, C, G, T) at this position
- **Confidence**: The ratio of number of occurrences of the alternate base to the read depth at this position.

We have not mentioned quality information associated with bases so far, but this is another key piece of information that SNP callers typically use. Our work currently considers just SNPs and the PARSNIP algorithm is customized for SNPs. Hence, we were able to modify the intermediate steps so that the input to PARSNIP is in an easy-to-parse format.

The key data structure used in our algorithm is a frequency table (denoted as $\mathcal{F}$). This array-based tabular representation is shown in Figure 3. Each logical row of this table stores the number of occurrences of each base. We also additionally store the read depth at that position.

PARSNIP processes full and partial match alignment records separately, since the overhead of reading CIGAR strings and subsequent conditional branching can be avoided for full match alignments. This results in a easily vectorizable and parallelizable innermost loop. Further, since the large majority of alignments are full match alignment records in high-coverage read data sets, this optimization handles the common case. For example, for the Venter human data set, nearly 97% of the aligned reads fall under full match category.

The first section of Algorithm 4 describes the processing of a full match alignment record. If read length is l, l rows in $\mathcal{F}$ starting from the alignment position of the read sequence are updated such that, in each of these rows, the base count (corresponding to the base aligned at that position) and the read depth are incremented by 1. A point to note is that the CIGAR

Algorithm 4: Updating the frequency table $\mathcal{F}$ for full and partial matches.

```
Input: Full alignment record R
l ← ReadLength(R);
refPosition ← AlignPosition(R);
for readPosition ← 0 to l do
    base ← Sequence(R)[readPosition];
    Increment F[refPosition].count[base];
    Increment F[refPosition].depth;
    Increment refPosition;
```

```
Input: Partial alignment record R
refPosition ← AlignPosition(R);
readPosition ← 0;
for each cigar term ⟨cOp, cLen⟩ do
    if cOp is 'M' then
        endPosition ← readPosition + cLen;
        while readPosition < endPosition do
            base ← Sequence(R)[position];
            Increment F[refPosition].count[base];
            Increment F[refPosition].depth;
            Increment refPosition and readPosition;
    else
        if cOp is 'I' or 'S' then
            readPosition ← readPosition + cLen;
        else
            endPosition ← refPosition + cLen;
            while refPosition < endPosition do
                Increment F[refPosition].depth;
                Increment refPosition;
```

operation *M* (Match) also contain SNPs, and all read bases in the matched positions need not actually match with the corresponding bases in the reference sequence.

For partial alignment records, the CIGAR string output of BWA consists of the following operations as described in the SAM Format Specification document [18].

- **M** - Match. This is the only operation which could contribute to SNPs.
- **I** - Insertion, **D** - Deletion, **S** - Soft clipping, **H** - Hard clipping.

As shown in Algorithm 4, partial alignment records are processed by iterating through the CIGAR operations. Each operation in the CIGAR string is a tuple consisting of CIGAR operation type *cOp* and length *cLen* of the reference sequence/read on which this operation should be applied.

If the operation is *Match*, the base count and depth fields of $\mathcal{F}$ are incremented for the matched positions, similar to the full match case described previously. If the operation is *Deletion*, then the depth field alone is incremented for these positions, since it indicates a gap character in these positions. For other operations, *cLen* bases are skipped in the aligned read to process the next CIGAR operation.

3.3.1 Parallelization

We use MPI to parallelize PARSNIP. The sorted full and partial match alignment files are mapped to MPI tasks in a many-to-one manner. Our mapping also ensures that

1. Each MPI task gets approximately equal number of alignment records to process.
2. Both the full and partial alignment record files corresponding to a contig are assigned to the same MPI task. This avoids the need to combine the partial entries in the frequency table $\mathcal{F}$ from different MPI tasks. Such a scenario could occur if files corresponding to the same contig are assigned to different MPI tasks.

After mapping input files to the MPI tasks, each task proceeds by reading a chunk of records (100,000) at a time, and processing them as described in Algorithm 4, depending on the record type read.

As is the case with the SAMPA step discussed in the previous section, PARSNIP is pleasantly parallel, since each MPI task has its own set of input files and creates separate output files containing the SNP records of the contigs that were assigned to it.

3.3.2 Algorithm Complexity

Both full and partial match algorithms run in time that is linear in terms of the length of the aligned reads. $\mathcal{F}$ is the major memory-consuming data structure. With PARSNIP, each MPI task requires only the current contig's frequency table $\mathcal{F}$ to be in memory. For human reference genome, the longest contig length is around 250 million base pairs corresponding to the contig *chr10*. A row in the $\mathcal{F}$ corresponding to one position of reference contig requires 11

bytes. So the largest contig requires approximately 2.75 GB memory, which is the maximum memory required per process. As larger number of MPI tasks are created per node, memory could become a potential bottleneck. This could be again be alleviated by hybrid parallelism, where 1 MPI task per process is created and multiple threads are created, and these threads within a task cooperate to process different alignment records corresponding to the same contig. While this is a straightforward extension, we leave this for future work, as the running times of both PARSNIP and SAMPA are extremely low compared to their counterparts in the reference pipeline.

4 Performance Results

In this section, we compare performance and quality results using tools in our new SPRITE pipeline to a reference pipeline with popular and commonly-used bioinformatics tools.

4.1 Experimental Setup

For all our experiments, we used the Edison supercomputer at the National Energy Research Scientific Computing center (NERSC) in California, USA. Edison is a Cray XC30 system with a peak floating-point performance of 2.57 PF/s, with a total of 5,576 compute nodes and 133,824 cores. Each compute node has two 12-core Intel Ivy Bridge processors. Each node also has 64 GB DDR3 memory. We build all the programs using the Intel C compiler and the Cray Programming Environment. Our input files are stored on of the local scratch partitions of Edison. We use a file stripe size of 8 MB and set large stripe settings to spread data across multiple I/O nodes.

We used the following four data sets, downloaded from the SMASH website [29]. Three of them use the human genome as reference, and one is based on the mouse genome.

- **Venter**: this is a synthetic human dataset created by inserting HuRef variants [3] into the hg19 reference genome.
- **NA12878 and NA18507**: these are real human datasets consisting of high-coverage Illumina reads.
- **Mouse**: this is a real dataset consisting of paired-end reads from the B6 mouse strain.

The SMASH benchmarking article [34] describes these data sets in more detail and explains their creation process. The website also gives running times and variant detection quality information for several different combinations of tools in the pipeline we have been focusing on.

Table 3 mentions the list of software tools that we used to build the reference pipeline. We have evaluated SPRITE's accuracy and efficiency by comparing our results to ones obtained from the reference pipeline. These tools were picked because of the combination of speed and result quality, as documented on the webpage.

Table 3: Tools used in experiments

Tool	Version	Purpose
BWA	0.7.10-r789	BWT-based Alignment
SNAP	1.0beta.17	Hash index-based Alignment
samtools	1.1	SAM file to BAM file, sorting and merging BAM files
freebayes	0.9.20	SNP calling

4.2 Alignment

We first evaluate the single-node and multi-node scalability characteristics of PRUNE, which is our extension of BWA-MEM to use MPI and generate a binary alignment output.

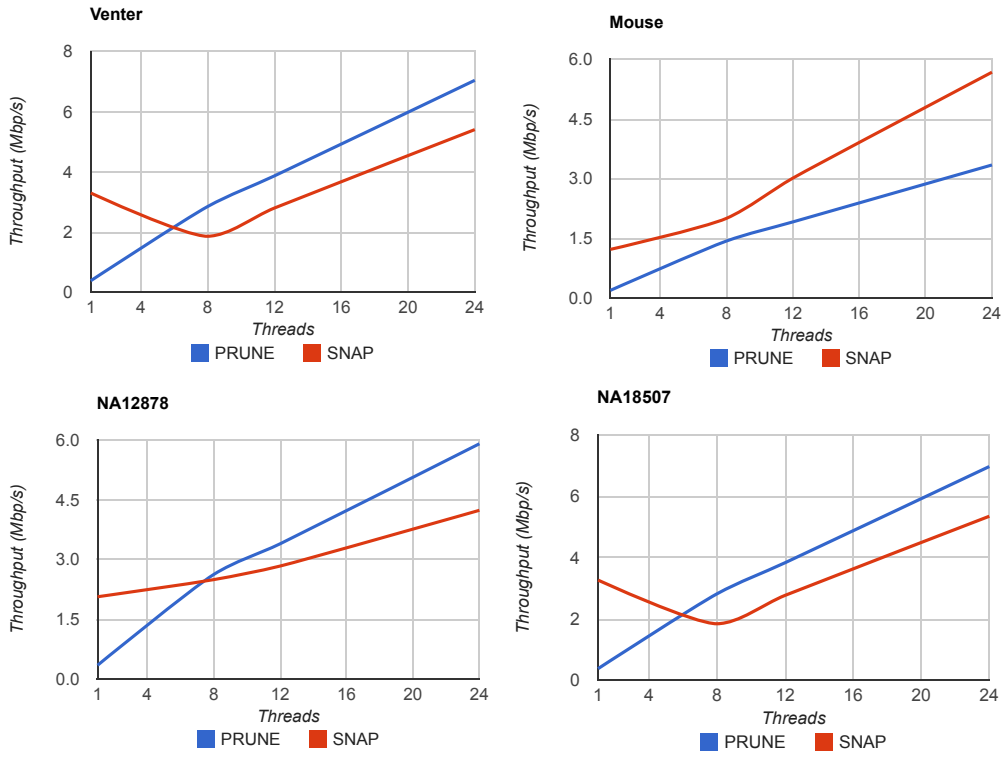

Figure 4: Shared-memory parallel performance comparison of PRUNE and SNAP aligners.

4.2.1 Single node scaling

Figure 4 gives the single node performance of the modified BWA implementation and SNAP aligner for the four datasets mentioned previously. We used about 1% of the reads present in each of the four datasets for calculating the throughput rates with our shared-memory runs.

These correspond to around 9 million (Venter), 15.6 million (Mouse), 16 million (NA12878), and 14 million (NA18507) reads. We note that the performance of original BWA-MEM is very similar to PRUNE, and hence we do not report its results. SNAP uses a different algorithmic strategy for indexing and hence we chose to compare performance to our approach.

Due to the hash index-based seed lookup, SNAP's single core performance is significantly better than that of BWA. But it can be seen that, BWA scales better than SNAP, especially when the number of threads is increased from 1 to 8. This could be because in Edison, each compute node has 2 sockets with 12 cores each and these 12 cores have shared access to L3 cache. SNAP assigns a range of reads to each core and hence when each core brings in a set of reads into L3 cache. It is likely that cores evict each other's read chunks, and this may lead to more cache misses. In comparison, BWA reads a chunk of reads, and each thread works towards mapping some of these reads. We believe this could be one possible reason for the performance gap. On 24 threads, SNAP is faster than PRUNE only for the mouse genomic data. We will need to further investigate the reasons for this performance switch. Another aspect to mention is that these alignments were carried out using the paired-end setting, which are often slower than single-ended alignment.

4.2.2 Multi-node scaling

We next use the full Venter and Mouse datasets to observe multi-node scalability of our hybrid parallel PRUNE implementation. Figure 4.2.2 shows the throughput (measured in terms of millions of bases aligned per second) of PRUNE (i.e., modified BWA-MEM). For our multi-node experiments, we use 1 MPI task per node and 24 threads per task. In Figure 5, we separate *compute time* from *I/O time*. Compute time is the time reported by BWA to look up alignments, once all the data is in memory. I/O time is the time for all other operations, including reading the chunks and writing them to possibly multiple files. We note that the compute time scales extremely well. Speedup is near-linear or even super-linear. But with increasing number of processes reading from the read files in parallel and writing to multiple files, I/O becomes the bottleneck and scalability drops, even though the compute time keeps reduces. For Venter, PRUNE scales upto 128 nodes whereas for Mouse dataset, overall scalability flattens after 64 nodes.

PRUNE also reduces the size of the output alignment files by filtering out reads of low quality and unaligned reads. Also, instead of writing the output in SAM file format, we write only those fields needed for SNP calling, as mentioned in Table 2. As a result, the size of output alignment records of our modified BWA for the Venter dataset is 131 GB, as compared to the 300 GB SAM file created normally.

4.3 SAMPA sort, PARSNIP variant calling

Table 4 shows the parallel performance of our SAMPA sort step and PARSNIP SNP calling phase. Our current MPI implementation is somewhat limited in its scalability, as we assign all alignment records corresponding to a contig to a single MPI task, in order to avoid synchronization bottlenecks. Memory requirement per MPI task of SAMPA is bounded by the

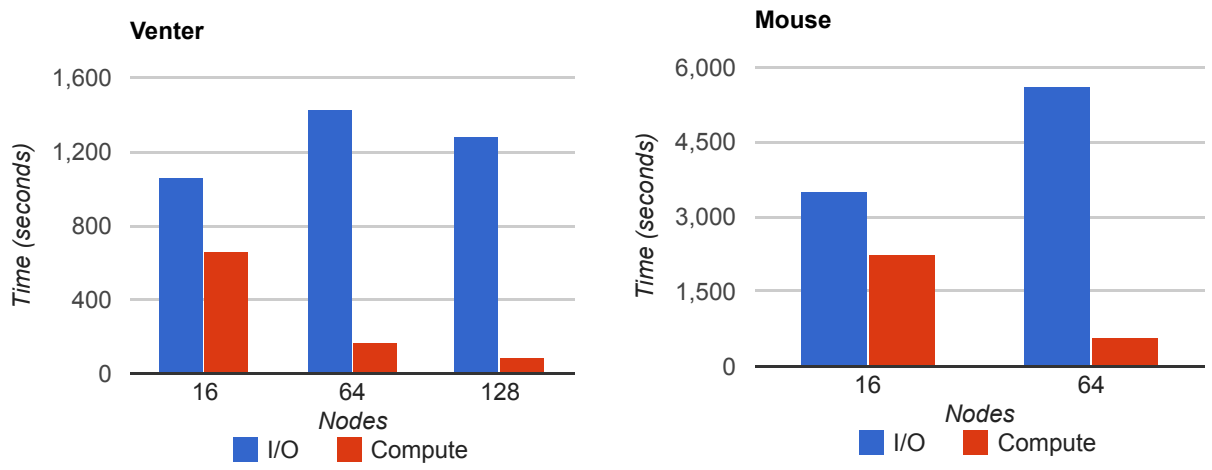

Figure 5: The breakdown of overall execution time of PRUNE into in-memory computation and I/O time of the Venter (left) and mouse (right) data sets.

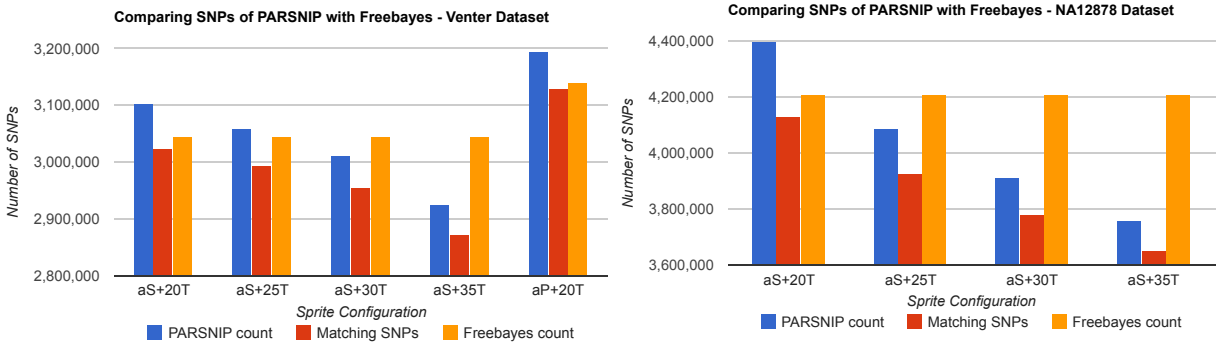

Figure 6: Comparing quality of results obtained using PARSNIP and freebayes on two human data sets and various confidence measures.

size of the largest alignment file created by PRUNE. For the Venter dataset, the maximum alignment output size is 11 GB corresponding to the contig *chr2* and hence, the maximum memory required per MPI process is 11 GB, since we perform the sort in memory. Consequently, we limit the number of MPI tasks per node to 4. With 16 MPI tasks, SAMPA for the entire Venter dataset finishes in 4 minutes and PARSNIP completes in less than 2 minutes. Because these routines are extremely fast, orders-of-magnitude faster than their counterparts, we did not pursue optimizing them further. We observed similar results for the other data sets evaluated.

4.4 End-to-end pipeline results

We now compare the whole pipeline execution time of SPRITE, as compared to the reference pipeline using a combination of serial and (shared memory-)parallel tools. For both these

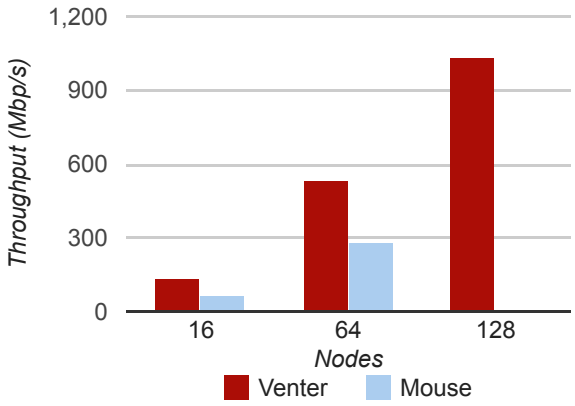

Figure 7: The multi-node parallel performance achieved by just the compute phase of PRUNE for two data sets.

Table 4: SAMPA and PARSNIP scalability for the Venter data set. Running time in minutes is indicated. The codes are MPI-only.

# Nodes	# Tasks	SAMPA	PARSNIP
1	1	21.50	8.0
1	4	8.54	3.5
4	16	4.04	2.0

pipelines, we have used the maximum supported parallelism. We execute BWA and samtools in the reference pipeline using a maximum of 24 cores available on a single node, and ran SPRITE on up to 128 nodes, and 24 cores on each node (3072 cores). As we can see in Table 5, the reference pipeline takes around 28 hours to execute on a single node, whereas SPRITE takes 34 minutes when using 16 nodes for alignment and 4 nodes for the sort and SNP calling steps. Using 128 nodes for alignment, the cumulative time is even lower, at 28 minutes. In the reference pipeline, SNP calling is by far the most expensive computation. However, alignment, specifically parallel I/O in the alignment phase, turns out to the limiting step in PARSNIP. Why do we see such large speedups with SAMPA and PARSNIP? It is because samtools and freebayes are designed to be run on workstations rather than parallel systems, and thus may have made some performance-crippling assumptions on available main memory and I/O bandwidth. Given the relative times of various steps in SPRITE, our next target would be to reduce I/O costs in alignment.

4.5 SNP-calling accuracy

Finally, we compare the output VCF files created using SPRITE to the output of the reference pipeline. We show comparisons for the full Venter (synthetic reads) and NA12878 (real reads)

Table 5: End-to-end pipeline execution time comparison for the Venter human data set.

Pipeline Stage	Ref. SNP Pipeline		SPRITE			
		Time (min)		Time (min)	# Nodes	Speedup
Alignment	BWA	393	PRUNE	28	16	14.03×
SAM file processing	samtools	401	SAMPA	4	4	100.25×
SNP Calling	freebayes	889	PARSNIP	2	4	444.50×
Overall		1683		34	16	49.50×

datasets in Figure 6. Since BWA is common to both the pipelines, this experiment reduces to comparing the outputs of the two SNP-calling methods: PARSNIP and freebayes.

In the Figure, *aS* denotes that the read files are aligned using single-ended alignment whereas *aP* denote paired-end alignment. T is the threshold employed by PARSNIP to report SNPs i.e, if an alternate allele mapped to a position of a reference contig occurs more than T percent of the read depth at that position, then PARSNIP reports the alternate allele as a SNP.

For Venter dataset with single-end read alignment, results are shown for 4 different T values, and for the same dataset with paired-end alignment, results are shown for *T=20*. For NA12878 dataset, 4 T configuration results are shown for single-end read alignment.

From Figure 6, we can infer that at as T increases, PARSNIP outputs fewer and fewer SNPs. But at the same time, the number of SNPs that do not match with the ones reported by freebayes also reduces. So we can use this threshold as a confidence measure to indicate that SNPs satisfying higher T values are more likely to be true SNPs. Because PARSNIP runs so fast, we can do parameter sweeps of this threshold to find an appropriate cutoff.

While we use a simple counting-based method to form the consensus string and subsequently call SNPs, freebayes employs a more sophisticated Bayesian probabilistic model [11]. Counting-based methods are known to perform well for high coverage genome datasets [27] like the ones we have used in our evaluations. But probabilistic models outperform counting based methods when the read depth is low. In future work, we will investigate adding probabilistic modeling to PARSNIP and apply it to the partial matched alignment records input, in order to further improve results. Another direction we intend to pursue is to accelerate PARSNIP performance through manycore parallelism.

5 Conclusions

In this paper, we present the design and implementation of a new pipeline for SNP detection called SPRITE. This pipeline consists of an aligner PRUNE based on BWA, a tool called SAMPA to sort the output of PRUNE based on the alignment position, and a parallel SNP caller PARSNIP. We have compared the performance of SPRITE to an existing pipeline using BWA for alignment, Samtools for processing SAM file output, and freebayes to call

variants. Using 16 compute nodes with 24 cores each, SPRITE completes in 34 minutes for the Venter human data set, as compared to 28 hours taken by the existing pipeline's single node execution. Also, the accuracy of SNPs reported by our variant calling step PARSNIP is comparable to the SNPs reported by freebayes. We will make our these tools publicly available at our project website: `sites.psu.edu/xpsgenomics`.

Acknowledgments

This research is supported by the National Science Foundation award # 1439057. We thank members of our project research team, particularly Paul Medvedev and Mahmut Kandemir, for helpful discussions.

References

[1] D. Adjeroh, T. C. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[3] N. Axelrod, Y. Lin, P. C. Ng, T. B. Stockwell, J. Crabtree, J. Huang, E. Kirkness, R. L. Strausberg, M. E. Frazier, J. C. Venter, S. Kravitz, and S. Levy. The HuRef Browser: a web resource for individual human genomics. *Nucleic Acids Research*, 37(suppl 1):D1018–D1024, 2009.

[4] BMI OSU HPC Lab. pMap: Parallel sequence mapping tool. `http://bmi.osu.edu/hpc/software/pmap/pmap.html`, last accessed April 2015.

[5] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical report 124. Technical report, Palo Alto, CA: Digital Equipment Corporation, 1994.

[6] Convey Computer™. TGAC speeds search with Convey. `http://www.conveycomputer.com/solutions/life-sciences/`, last accessed April 2015.

[7] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *Proc. 4th Int'l. Conf. on Linux Clusters: The HPC Revolution 2003 in conjunction with ClusterWorld Conference and Expo*, 2003.

[8] J. A. Edwards and U. Vishkin. Parallel algorithms for Burrows-Wheeler compression and decompression. *Theoretical Computer Science*, 525:10–22, 2014.

[9] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. Symp. on Foundations of Computer Science*, pages 390–398, 2000.

[10] M. H. Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughout DNA sequencing data using reference-based compression. *Genome Research*, 21:734–740, 2011.

[11] E. Garrison and G. Marth. Haplotype-based variant detection from short-read sequencing, 2012. `http://arxiv.org/abs/1207.3907`.

[12] A. Hatem, D. Bozdag, A. E. Toland, and U. V. Catalyurek. Benchmarking short sequence mapping tools. *BMC Bioinformatics*, 14:184, 2013.

[13] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at UCSC. *Genome research*, 12(6):996–1006, 2002.

[14] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. H. Yeo, and B. Y. H. Lam. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5:27, 2012.

[15] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

[16] H. Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 2012.

[17] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[18] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[19] H. Li, J. Ruan, and R. Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008.

[20] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[21] C. Liu, T. Wong, E. Wu, R. Luo1, S. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T. Lam. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.

[22] Y. Liu and B. Schmidt. CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing. *IEEE Design and Test*, 31(1):31–39, 2014.

[23] Y. Liu, B. Schmidt, and D. L. Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform. *Bioinformatics*, 28(14):1830–1837, 2012.

[24] H. Martínez, J. Tárraga, I. Medina, S. Barrachina, M. Castillo, J. Dopazo, and E. S. Quintana-Ortí. Concurrent and accurate RNA sequencing on multicore platforms. Technical Report ICC 2013-03-01, Jaume I University, 2013.

[25] H. Martínez, J. Tárraga, I. Medina, S. Barrachina, M. Castillo, J. Dopazo, and E. S. Quintana-Ortí. A dynamic pipeline for RNA sequencing on multicore processors. In *Proc. European MPI Users' Group Meeting (EuroMPI)*, pages 235–240, 2013.

[26] N. M. Mohamed, H. Lin, and W. Feng. Accelerating data-intensive genome analysis in the cloud. In *Proc. Int'l. Conf. on Bioinformatics and Computational Biology (BICoB)*, 2013.

[27] R. Nielsen, J. S. Paul, A. Albrechtsen, and Y. S. Song. Genotype and SNP calling from next-generation sequencing data. *Nat Rev Genet.*, 12(6):443–451, 2011.

[28] NVBIO library. `https://github.com/NVlabs/nvbio`, last accessed Apr 2015.

[29] SMaSH: A benchmarking toolkit for variant calling. `http://http://smash.cs.berkeley.edu/`, last accessed April 2015.

[30] S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno. Shrimp: accurate mapping of short color-space reads. *PLoS computational biology*, 5(5):e1000386, 2009.

[31] J. T. Simpson. Exploring genome characteristics and sequence quality without a reference. *arXiv preprint arXiv:1307.8026*, 2013.

[32] J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):367–373, 2010.

[33] J. T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

[34] A. Talwalkar, J. Liptrap, J. Newcomb, C. Hartl, J. Terhorst, K. Curtis, M. Bresler, Y. S. Song, M. I. Jordan, and D. Patterson. SMaSH: a benchmarking toolkit for human genome variant calling. *Bioinformatics*, 30(19):2787–2795, 2014.

[35] Y. Xin, B. Liu, B. Min, W. X. Y. Li, R. C. C. Cheung, A. S. Fong, and T. F. Chan. Parallel architecture for DNA sequence inexact matching with Burrows-Wheeler Transform. *Microlectronics Journal*, 44(8):670–682, 2013.

[36] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler. Faster and more accurate sequence alignment with SNAP, 2011. `http://arxiv.org/abs/1111.5572`, last accessed Apr 2015.

[37] J. Zhang, H. Lin, P. Balaji, and W. Feng. Optimizing Burrows-Wheeler Transform-based sequence alignment on multicore architectures. In *Proc. IEEE/ACM Int'l. Symp. on Cluster, Cloud, and Grid Computing (CCGrid)*, pages 377–384, 2013.

[38] Y. Zhu, H. Jiang, X. Qin, and D. Swanson. A case study of parallel I/O for biological sequence search on Linux clusters. In *Proc. IEEE Int'l. Conf. on Cluster Computing (Cluster)*, pages 308–315, 2003.