Abstract—Next-generation sequencing (NGS), also known as high throughput sequencing is currently being utilized in the Bioinformatics Department of the University of California at San Diego as a research tool to examine and detect variants (mutations) in genomes of individuals and related family members. Unfortunately, given the current tools and compute environment, a typical NGS work-flow presently takes between 12 to 14 days to complete. There is therefore a desire within the department to investigate the possibility of reducing the turnaround time to accommodate current/future research needs that would tread a similar path.

This paper describes the investigation that has been carried out over a period of few weeks, on the first phase of the sequencing that takes around 5 to 7 days with the current software tool chain and hardware utilizing a set of smaller but still representative data sets and it will conclude with some recommendations and lessons learned. Additionally, we were also able to build a surprisingly accurate model which predicts the behavior of the tool chain.

Keywords—NGS, Variant Calling, BWA, Picard, GATK, BAM, SAM

I. INTRODUCTION

A full end to end NGS work-flow involves a number of smaller work-flows/phases including utilizing hardware sequencers such as ones made by Illumina, pre-processing the output BAM files by running them through a set of software tools to perform tasks such as sequence mapping (hereby called the pre-processing phase) before concluding with variant calling, annotation and filtering. A BAM file is an industry standard binary version of a SAM file with the later being a tab-delimited text file containing sequence alignment data. For the purposes of this paper, we will only focus on the pre-processing phase since this is the work-flow that is currently taking the longest to complete 1.

An overview of the current hardware and software tool chain (including how much time each tool is currently taking to complete in a typical execution) has been provided in Table II.

The current work-flow is currently executed via a Perl script that executes each stage in Table II in sequential order. When possible, the communication between the stages is done through Unix pipes to reduce I/O with temporary files being used when piping is not feasible. Additionally, although each stage proceeds sequentially, wherever applicable, the tool utilized in each stage is currently executed with parameters that would take advantage of extra processors/cores.

II. OVERVIEW

We start by talking about the different tools that make up the pipeline in Section III. We then move to describing the workings of the pipeline itself in Section IV. Section V describes our framework to measure different system resource usage while running the pipeline and also tries explain challenges faced while replicating the environment on a new system. Section VI describes initial analysis and includes some insights into CPU, I/O and Memory usage along with Java GC and hardware performance counters. Section VII attempts to model the entire pipeline in mathematical terms and section VIII evaluates the model and lists some suprising results. Section IX uses the model to analyse reoccurring issues in the pipeline and makes some future predictions. Finally, section X concludes with lessons learnt and some of the recommendations for the pipeline.

III. TOOLS AND CONFIGURATIONS

To understand the pipeline it is important that we understand the tools that make up the pipeline. Hence, we begin by describing the tools and the roles they play. The pipeline consists of the following tool chains for processing 2:

1) BWA
2) SAMtools
3) HTSlib
4) Picard
5) Genome Analysis Toolkit (GATK)

A. BWA

BWA 3 is a software package for mapping low-divergent sequences against a large reference genome, such as the human genome. It consists of three variations of the Burton-Wheeler Aligner algorithm: BWA-backtrack, BWA-SW and BWA-MEM. The pipeline uses BWA-MEM as it is capable of for processing up to 1 million base pairs (bp)[1].BWA-MEM

TABLE I: Machine Description

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Model</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>1.8GHz</td>
</tr>
<tr>
<td>No. of Processors</td>
<td>1</td>
</tr>
<tr>
<td>No. of Cores per Processor</td>
<td>4</td>
</tr>
<tr>
<td>RAM Size</td>
<td>100 GB</td>
</tr>
<tr>
<td>Disk</td>
<td>228 GB</td>
</tr>
</tbody>
</table>

1http://www.slideshare.net/AustralianBioinformatics/introduction-to-nextgeneration


3http://bio-bwa.sourceforge.net/
TABLE II: Software Tool Chain

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Tool Family</th>
<th>Software Tool</th>
<th>Current Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shuffling and Aligning Input File</td>
<td>N/A</td>
<td>htscmd, bwa, samtools (C language)</td>
<td>33 hours</td>
</tr>
<tr>
<td>SAM File Sorting</td>
<td>Picard</td>
<td>SAMSort (Java)</td>
<td>8 hours</td>
</tr>
<tr>
<td>Mark Duplicates</td>
<td>Picard</td>
<td>MarkDuplicates (Java)</td>
<td>8 hours</td>
</tr>
<tr>
<td>BAM File Index construction</td>
<td>Picard</td>
<td>BuildBamIndex (Java)</td>
<td>1 hour</td>
</tr>
<tr>
<td>Building Insert Delete (Indel)</td>
<td>GATK</td>
<td>RealignerTargetCreator (Java)</td>
<td>8 hours</td>
</tr>
<tr>
<td>Realignment around Indel</td>
<td>GATK</td>
<td>IndelRealigner (Java)</td>
<td>8 hours</td>
</tr>
<tr>
<td>Base Q Covariance 1st Stage</td>
<td>GATK</td>
<td>BaseRecalibrator (Java)</td>
<td>30 hours</td>
</tr>
<tr>
<td>Base Q Covariance 2nd Stage</td>
<td>GATK</td>
<td>BaseRecalibrator (Java)</td>
<td>80 hours</td>
</tr>
<tr>
<td>Plot Base Q Results</td>
<td>GATK</td>
<td>Anayze Covariates (Java)</td>
<td>0 hours</td>
</tr>
<tr>
<td>Base Q Recalibration</td>
<td>GATK</td>
<td>PrintRead (Java)</td>
<td>33 hours</td>
</tr>
</tbody>
</table>

TABLE III: Parallelism in GATK

<table>
<thead>
<tr>
<th>Tool</th>
<th>Full Name</th>
<th>Supported Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC</td>
<td>Realign Target Creator</td>
<td>NT</td>
</tr>
<tr>
<td>IR</td>
<td>Indel Realigner</td>
<td>SG</td>
</tr>
<tr>
<td>BR</td>
<td>Base Recalibrator</td>
<td>NCT, SG</td>
</tr>
<tr>
<td>PR</td>
<td>Print Reads</td>
<td>NCT</td>
</tr>
</tbody>
</table>

is highly parallel as it works on independent chunks of base pair reads.

B. SAMtools

The SAM (Sequence Alignment/Map) file format is a generic format for storing large nucleotide sequence alignments. SAM Tools provides various utilities for manipulating alignments in the SAM format, including sorting, merging, indexing and generating alignments in a per-position format. It is inherently a single threaded application[2].

C. HTSlib

HTSlib\(^4\) is an implementation of a unified C library for accessing common file formats, such as SAM, CRAM and VCF, used for high-throughput sequencing data, and is the core library used by samtools and bcftools. The binary is called htscmd and it is used to shuffle the input data and convert the later into a single fastq file. This file is then provided as an input to the BWA tool. This tool is also single threaded and doesn’t have any parallelism in it.

D. Picard tools

Picard\(^5\) is comprised of Java-based command-line utilities that manipulate SAM files, and a Java API (HTSJDK) for creating new programs that read and write SAM files. Both SAM text format and SAM binary (BAM) format are supported by the tool. The pipeline uses 3 utilities from the Picard tool set for three purposes, namely, sorting the SAM file (SortSam), marking duplicates (MarkDuplicates) and building the BAM index (BuildBamIndex). All these utilities are again unfortunately single threaded and the algorithms don’t employ any parallelism.

E. Genome Analysis Toolkit (gatk)

Similar to Picard, GATK\(^6\) too is a set of tools and it serves as the core of the pipeline performing the major analysis tasks on the genome. The GATK is the industry standard for analyses such as identifying rare mutations among exomes as well as specific mutations within a group of patients. The specific tools that are being used are RealignerTargetCreator, IndelRealigner, BaseRecalibrator, PrintReads. The GATK was built from the ground up with performance in mind. It employs the concept of Map Reduce, which is basically a strategy to speed up performance by breaking down large iterative tasks into shorter segments whose output will then be merged into some overall result. Additionally, it also employs multi-threading heavily. Multi-threading is enabled simply by using the nt and nct command line arguments[3].

Here nt represents the number of data threads sent to the processor and nct represents the number of CPU threads allocated to each data thread. Apart from multi-threading, they also have a notion of “Scatter-Gather” which can be applied to a cluster of machines. Scatter-Gather (SG) is a very different process from multi-threading because the parallelization happens outside of the program itself. It basically creates separate GATK commands for a portion of input data and send this command to different nodes of the cluster.

Not all the tools support all different kinds of parallelism, below is the table which lists down these support.

As seen in Table III, SG is not supported by all the tools. Ones that support are not the bottle neck of the pipeline as explained in section VII

Next section describes our framework for measuring the system performance.

IV. THE PIPELINE

In this section, we attempt to describe our understanding of each step in the workflow in computer science layman terms as opposed to the point of view of a trained Bioinformatics scientist. For the purposes on analysis, we also divide the pipeline into logical phases.

The input to the workflow is a set of BAM files which is essentially a really large set of files (totalling 1 Billion short reads per genome per person) containing a collection of short reads. Short reads are fragments of a much longer DNA sequence and these are produced by hardware sequencers such as ones produced by Illumina. There are technologies in the marketplace that can produce long reads as well, but we will not discuss them here given our limited understanding of the topic.

\(^4\)https://github.com/samtools/htslib
\(^5\)http://picard.sourceforge.net/
\(^6\)http://www.broadinstitute.org/gatk/about/
Once this is done, these short reads are fed into the pipeline that is made up of 14 stages.

A. Phase 1: Shuffle and Align

In this stage, the short reads in the BAM files will first be shuffled to minimize bias during the alignment process. Once shuffling is completed, each of the short reads will then need to be aligned (mapped to a specific position) to a large reference genome (in our case this is a 3 billion long sequence). From the outset, read alignment seems to be a simple problem i.e. find a particular substring within a bigger string, however read errors can and do happen, hence turning this into an approximate hashing problem which is then solved utilizing the Burrows Wheeler Aligner (BWA) transform algorithm. It is our understanding that each of the short reads can be sequenced in this stage without knowledge of any other short read data as long as the input BAM file is valid. While running initial experiments we found an interesting problem related to missing base-pairs which is further discussed in section V.

B. Phase 2: SAM Sorting

Many of the downstream analysis programs which utilizes BAM files actually require the files to be sorted since this allows reading from these files to be done more efficiently.

C. Phase 3: Remove Files

Phase 3 removes the temporary files created in the earlier phases.

D. Phase 4: Mark Duplicates

In this phase, duplicates of any particular unique short read will be marked to prevent a skew during the variant calling process. Duplicates are usually produced due to a particular DNA preparation process and may be unavoidable. Marking duplicates sounds like something that can be built on top of the canonical example of using Map reduce i.e. counting the number of words in a given document.

E. Phase 5: Remove Files

This phase too we remove some files

F. Phase 6: Index Dedup (Build BAM Index)

In this phase, the output BAM files from preceding stages are indexed for fast access. This essentially allows a particular short read to be accessed by jumping immediately to a specific offset within a particular BAM file thus negating the need to read preceding data into memory. The output of this process is a set of accompanying index files to the original BAM files.

G. Phase 7 and 8: InDel Targets and Realign InDels

The next two phases pertains to Insertion Deletion (InDel) and thus would benefit from a short overview. The term InDel refers to a class of variations that is present in a human genome. The need to align short reads around InDels arises due to 2 major reasons. The first reason is that InDel can cause mappers (such as the BWA algorithm employed in Phase 1) to misalign short reads. The second reason is that those misalignments then would harm the accuracy of downstream processes such as base quality recalibration and variant detection.

Here, the regions in the BAM files that will need to be realigned are identified. In general there are three types of realignment targets: known sites such as ones coming from the 1000 Genome project, InDels that are seen in the original alignments (as part of the application of the BWA algorithm), and finally sites where evidence suggests a hidden InDel.

Once the InDel regions have been identified, this stage would then perform the actual realignment.

H. Phase 9: Remove Files

Phase 9 removes the temporary files created by the earlier phase.

I. Phase 10: Baseq (Base Quality) Covariance Stage 1 (Base Recalibration)

Hardware sequencers would associate a quality score with their reads. There is however a tendency for sequencers to be overly optimistic in terms of their confidence scores. In this stage, a recalibration table will be built utilizing some machine learning algorithm based on covariation among several features of base such as read group, the original quality score from the sequencer, 1st/2nd read in a pair, etc.

J. Phase 11: Baseq (Base Quality) Covariance Stage 2

In this phase, the recalibration table built on the previous stage would be utilized to recompute the base quality score.

K. Phase 12: Plot Base Quality Covariance

In this stage, the plot of the recalibration tables are generated so that an evaluation can be made on whether the recalibration has worked properly.

L. Phase 13: Base Quality Recalibration

In this phase, the recalibrated data is subjected to some final processing before written out to disk.

M. Phase 14: Remove Files

Phase 14 simply removes the Index realignment files.

---

7http://www.broadinstitute.org/gatk/guide/tagged?tag=bam
8http://hmg.oxfordjournals.org/content/19/R2/R131.full
V. Framework

In this section, we describe the framework that was setup to run the pipeline and the changes and modifications done to the framework for the purposes of analysis. In addition, we also discuss the challenges faced, while trying to get working the pipeline with a subset of the reads to ease analysis. Further, we also talk about the challenges we faced while duplicating the environment on another machine.

A. Framework Changes

The framework - essentially a script provided to us needed an overhaul and further additions so that we could start measuring system performance for each phase of the pipeline. To measure basic resources like, CPU utilization, Memory utilization and Disk I/O we decided to use dstat tool for each command. For collecting hardware performance counters, e.g. L1 cache misses, last level cache misses etc. we used perf-stat utility. Also, since most of the tools are Java based, we thought it would beneficial to collect JVM Garbage Collector logs as well.

B. Data Sources

Our initial experiments were run on the master node of the Bioinformatics lab during a short period of time when it was not being utilized by the the department. This gave us an advantage since we did not have to simultaneously understand both tools and environment at the outset. However, once the normal usage of the node was resumed, we had to utilize a different machine. The critical issue was that we had been using private patient data up to that point and due to privacy concerns it was impossible for us to move the data set into the new machine. We were thus faced with the challenge of finding an equivalent data set. We tried a number of sources and finally found the 1000 Genome project to be fruitful.\(^\text{10}\)

C. Pair issue and ‘R’

Our woes did not stop after being able to find a data set from 1000 Genome project, We expected the subset of the reads to successfully run in the pipeline. However, we faced cryptic errors in the first phase of the pipeline itself. After multiple long debugging sessions and help from the bioinformatics people we found out that the base pair reads come in pairs and the bwa tool requires that every base pair has its pair in the data set.

Since we were running our experiments for a subset of the reads there was a probability that certain base-pairs didn’t have their pair and because of this pipeline was failing at the first phase itself. So to remove this error, we had to write a wrapper script around the given pipeline to pre-process the input bam file and remove all the reads whose pairs didn’t exist.

Challenges didn’t end here, there is a phase in the pipeline which uses ‘R’ to plot graphs and it turns out that tools are using deprecated libraries of ‘R’ for plotting which we couldn’t install while replicating the environment, so we had to skip that phase during our experiments. Our initial analyses on bioinformatics machine showed that this phase did not take significant time estate of the pipeline and hence it shouldn’t impact our future experiments.

Finally, after all these changes, we were able to replicate the complete environment on a new server where we could experiment with different data-set sizes and analyse results.

VI. Baseline Run

In order to gain a better understanding of the pipeline, we started our investigation by running the pipeline using the simplest possible configuration and yet still have it perform meaningful work. This is accomplished by setting the number of threads to 1 in each phase of the pipeline as well as utilizing only 1% of the full data set used in a typical run in the Bioinformatics department, which comes to be around 10 million reads. The choice of 1% of the full set was informed through conversations with a student from the department.

The next level of experiments consisted of running the same pipeline and dataset but with higher number of threads and since we were working on a quad-core machine, we chose 4 threads for immediate comparison. Later in the paper, we look at how SMT performs.

Table IV shows that there are mainly 5 phases which contribute to approx. 84% of the total time. Looking at the graphs for those phases namely, figures 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10, it is evident that neither I/O or Memory is a bottleneck. Remaining graphs of the phases can be found in Appendix A.

This is further evident when we captured the resource usage pattern for an actual run that lasted for approx. 7 days. Figures 11 and 12 show that although I/O was increased, its not the bottle neck of the system and the tools never ran out of memory.

Although we believe that memory bandwidth contention might be one of the possible reasons why these phases are slow but due to timing constraints we were not able to explore that avenue.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
Phase Name & Single Thread Time (s) & 4 Thread Time (s) \\
\hline
Shuff Algn & 4932 & 1505 \\
Sort Sam & 244 & 244 \\
Remove BAM & 1 & 1 \\
DeDup Srd & 299 & 299 \\
Remove Srd & 1 & 1 \\
Indel Target & 40 & 40 \\
DeDup Target & 2961 & 830 \\
Indel Target & 393 & 394 \\
Remove Dedup & 1 & 1 \\
Base Covar 2 & 1276 & 719 \\
Base Covar 2 & 1907 & 1330 \\
Plot (Didn’t Measure) & 0 & 0 \\
Basey Realn & 1085 & 721 \\
Remove Realn & 1 & 1 \\
\hline
\end{tabular}
\caption{Time taken by different phases of the pipeline for different number of threads for dataset of size 10 Million reads}
\end{table}

A. Investigating Java Runtime Environment

With the exception of the first phase of the pipeline, all the utilized tools are programs written in Java. This provided us with another avenue of investigation to pursue through the enabling of Java Garbage Collection (GC) logging.

\(^{10}\)http://www.1000genomes.org/
Given the batch nature of each phase in the pipeline, we are primarily interested in knowing the phase throughput i.e. the percentage of time spent by each pipeline phase doing useful work instead of GC. In general, any throughput number at 95% and above are considered good. Additionally, should the throughput number falls below 95%, we are also interested in seeing any instances of a GC taking an excessively long amount of time.

We modified the script used to run the tool chain to augment every “java” command with the following flags

-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps

The set of flags serve to output GC data in sufficient details including the amount of memory freed in both Young and Old Generations in each iteration as well as the times during which GC happens into the specified log file. We then used a tool called JClarify Censum to visualize the data and collect the throughput metric.

1) Throughput Results: When it comes to the pipeline, the phases utilizing Java that contribute the most to the running time are Indel Targets, Base Covar 1, Base Covar 2, and Baseq Recal so we are going to constrain our discussion to those 4 phases. For any combination of input size and number of threads, we found that the throughput number never dropped below 95% with the exception of the Base Covariance phase where the number dropped steadily when the number of

---

12http://www.jclarity.com/censum/
threads specified is 8 (at 4 threads and below, the throughput remained at above 95)

We then made some modifications to the “java” command to see if we can improve the throughput number for that phase. In particular we found that for a data size of 40 million, we were able to increase the throughput from 93.6% to 96.1% by specifying a number of additional flags

<table>
<thead>
<tr>
<th>Indel Targets</th>
<th>10M</th>
<th>20M</th>
<th>40M</th>
<th>100M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indel Targets</td>
<td>98.6%</td>
<td>98.9%</td>
<td>98.7%</td>
<td>98.9%</td>
</tr>
<tr>
<td>Base Covar 1</td>
<td>95.8%</td>
<td>94.7%</td>
<td>93.6%</td>
<td>92%</td>
</tr>
<tr>
<td>Base Covar 2</td>
<td>97.1%</td>
<td>96.8%</td>
<td>96.6%</td>
<td>96.6%</td>
</tr>
<tr>
<td>Baseq Recal</td>
<td>97.4%</td>
<td>97.5%</td>
<td>97.5%</td>
<td>97.6%</td>
</tr>
</tbody>
</table>

TABLE V: GC Throughput for number of threads = 8

The choice of those numbers were informed based on the raw GC data collected in particular the sizes of the Young and Old Generation at the end of the GC log for that phase with some buffer built in to tolerate possible memory spikes.

Improvement aside, we don’t think that it is particularly meaningful, for example, if we refer to table V, improving Base Covar 1’s throughput from 92% to 95% would only result in an improvement of around 1 minute (3% of 35 minutes) in running time. It is true that given that the particular phase is currently taking around 30 hours (using the full data set) that we may be seeing some real savings in time, however we also know that the phase is currently being run with the number of threads set to 5 in the Bioinformatics head node (and earlier...
we have mentioned that throughput is not an issue when the
number of threads is 4). In other words, without further work,
we are not able to ascertain how much of an improvement
would tuning this particular phase result in.

Another thing that is worth mentioning is that early in
our experiments, we were seeing Java programs ran in single
threaded configuration through the use of nct (see some
other table for information on tools that accept number of
data/compute threads) consuming more than a thread’s worth
of CPU utilization (“top” would occasionally show utilization
above 30%) . We utilized an application called JConsole which
is shipped with any JDK since version 5.0 to investigate the
issue since the later has the ability to show all running threads
in a particular Java Virtual Machine (JVM).

What we found was that although the tool did start up
a number of data/compute threads as specified through a
configuration option, there were also a number of utility
threads started by the tool and the JVM including a Progress
Tracking thread (started by GATK), a number of GC threads
and a number of TCP/IP threads. We were not able to ascertain
why the JVM would start a number of TCP/IP threads, and
there does not seem to be any flags specific turning those
threads off, however the presence of these extra threads do
explain the CPU utilization phenomenon that we were seeing.

B. Performance counter measurements

Just to be sure that the delay in the phases are not caused
by lot of L1 cache misses, branch-predictor misses or Off-
Chip accesses, we measured hardware performance counters
using perf-stat tool and found it to be consistent across
multiple runs for different input sizes and thread numbers.
There was about 3.5% of L1 Data cache miss rate, 1.2% of
Branch predictor misses. But an interesting result was 31.3%
of LLC misses which was an interesting result but because of timing constraints we couldn’t explore this avenue as well. However, since we saw that these measurements seemed pretty much constant across runs with different sizes and threads we concluded that while the cache misses could be potentially interesting it wouldn’t affect our ability to model the pipeline.

VII. THE PIPELINE MODEL

One of the ideas behind profiling the pipeline was to build a model based of which we could make predictions. The idea being that if we could accurately predict and model the behavior of the software pipeline we had, in a manner of speaking, truly understood the workings of the system as a black box.

As mentioned previously, to better understand the pipeline we logically split the pipeline into phases. In retrospect, this turned out to be a crucial step in building the model as this enabled us to better predict the behavior of the entire pipeline as a combination of phases rather than as a single entity. Essentially, we have built a model for each of the phases which is then used to make a prediction for the entire pipeline. More specifically, the model will utilize the size of input data and number of threads to make a prediction for the time the pipeline will take to complete.

A. Building the Model

In building the model, we realized that there are four important factors that affect the running time. Further more, each of the phases had a different behavior which seemed to stem from change in the four factors.

The factors that affect running time and form an integral part of the model are:
Fig. 9: Single thread resource usage for Phase 13: Base Recalibration

Fig. 10: 4 threaded resource usage for Phase 13: Base Recalibration

Fig. 11: CPU and I/O utilization for full run on 1 billion reads.
1) \( f \): The fraction of the phase that is parallelizable
2) \( p \): The number of threads used
3) \( m \): The fraction of the phase that actually depends on the size of the input
4) \( s \): The size of the input

Every phase, has it's own set of \( f, m \) which causes the phases to show different behavior. One assumption we make here is that for fraction \( f \) of the pipeline the program is perfectly scalable. However, in reality it is possible that the performance of the phase will stop improving as the number of threads is increased, or in fact degenerates after a certain threshold has been reached. In fact, we see this exact behavior for one of the phases. Considering that we are treating all the programs as black-boxes it would be hard to pinpoint the exact reasons for the degeneration. However, we briefly attempt to address the issue later in this section.

1) Model Terminology: In this section we briefly elaborate on the terminology we intend to use for building the model. Each phase, in addition to the two internal parameters \( f \) and \( m \) has two external parameters that it depends on. One is the number of threads \( p \), and the second is the size of the input ie, the number of reads \( s \). These parameters are external and could possibly change.

Every phase is denoted by \( P_{p,s}^i \), where \( p \) is the number of threads and \( s \) is the input size and \( i \) is the phase number in the pipeline.

Thus, the total time for the entire pipeline \( E \) can be represented as

\[
\sum_{i=1}^{n} t(P_{p,s}^i) = t(E)
\]

For the purposes of the analysis we consider \( 10M \) as the base size. As mentioned in Section. VI, the model uses the data for \( 10M \) as ground truth because using an input size smaller than that does not produce meaningful results, not only from the correctness point of you but also from the behavior point of view. Also as discussed before, the time taken by every phase depends on \( f, p, s, m \). However, there are two factors that are inherent to the phases and are not stated explicitly. Hence, the success of the entire model revolves around two factors \( f, m \) and how well we estimate and compute these factors.

For this, we decided to stay simple and come up with a simple way to compute \( f \) and \( m \). For \( f \), Amdahl’s seemed like a good approximation. In fact, we see the Amdahl’s Law approximation works really well and seems to provide a good starting point. Considering our success for \( f \) with Amdahl’s law we decided to use a slightly modified version of the Amdahl’s law to get a value of \( m \) for each phase.

To get \( f \), we basically, for each phase \( P_{p,10M}^i \) and \( p = 2, 4, 6 \), get a corresponding \( f_{p,10M}^i \) and compute \( f \) as \( f^i = \min(f_{2,10M}^i, f_{4,10M}^i, f_{6,10M}^i) \)

Note, \( f_{p,s}^i \) denotes the fraction of the phase \( i \) that is parallelizable for size \( s \). However, since the size of the input \( s \) should not affect the parallelizability of the phase we can estimate \( f \) with \( s = 10M \).

On the other hand to get \( m \), we basically, for each phase \( P_{p,20M}^i \) and \( p = 2, 4, 6 \), get a corresponding \( m_{p,20M}^i \) and compute \( m \) as \( m^i = \max(m_{2,20M}^i, m_{4,20M}^i, m_{6,20M}^i) \)

Note, \( m_{p,s}^i \) denotes the fraction of the phase \( i \) that depends on the size of input when compared to \( s = 10M \) as the base. However, since the size of input \( s \) will not change the fraction of the phase that depends on the size we can estimate \( m \) with \( s = 20M \).

We choose the \( \min \) for \( f \) and the \( \max \) for \( m \) as we would like to make conservative estimates.

B. Enumerating the Model: Phase 1

In this section, we try to make concrete the above calculations by walking through enumerating the model for Phase 1. Essentially, we try to solidify the model by showing how we build the model and then make predictions for Phase 1.

1) Computing \( f \): To compute \( f^i \) we use the data for \( s = 10M \) and \( p = 2, 4, 6 \)

Using Amdahl’s Law we get

Fig. 12: Memory utilization for full run on 1 billion reads.
Table VI: Phase 1, \( s = 10M \)

<table>
<thead>
<tr>
<th>Threads ( p )</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4932</td>
</tr>
<tr>
<td>2</td>
<td>2663</td>
</tr>
<tr>
<td>4</td>
<td>1529</td>
</tr>
<tr>
<td>6</td>
<td>1151</td>
</tr>
</tbody>
</table>

Thus we get \( f^1 = 0.9200 \)

Table VII: Validate \( f \), Phase 1, \( p = 8 \) threads

<table>
<thead>
<tr>
<th>( P_{8,10M}^i )</th>
<th>Predicted (seconds)</th>
<th>Actual (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>962</td>
<td>963</td>
</tr>
</tbody>
</table>

As a quick validation of our estimation we use this to predict the value \( P_{8,10M}^1 \). As we can see in Table. VII that our estimation of \( f^1 \) is pretty spot on.

Table VIII: Phase 1, \( s = 20M \)

<table>
<thead>
<tr>
<th>Threads ( p )</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4974</td>
</tr>
<tr>
<td>4</td>
<td>2909</td>
</tr>
<tr>
<td>8</td>
<td>1867</td>
</tr>
</tbody>
</table>

2) Computing \( m \): To compute \( m^1 \) we use the data for \( s = 20M \) and \( p = 2, 4, 8 \)

To evaluate \( m \) we define a new term \( r \) which is a ratio of the size \( s \) to \( 10M \). Further, we modify the Amdahl’s Law equation slightly to make it compute \( m \).

\[
m = \frac{1}{r-1} \left(1 - \frac{t(P_{8,20M}^i)}{t(P_{8,10M}^i)}\right)
\]

\[
m_{2,20M}^1 = 0.9145
\]

\[
m_{4,20M}^1 = 0.9329
\]

\[
m_{8,20M}^1 = 0.9387
\]

Thus we get \( m^1 = 0.93870 \)

C. Evaluating the Pipeline

Finally, we have all the data we need to evaluate the pipeline and make predictions. It is important to understand that we evaluate the pipeline in phases rather than as a whole. Based on our measures for \( f \) and \( m \) we can now rewrite the time taken by a phase as

\[
t(P_{p,s}^i) = t(P_{1,10M}^i) \cdot \left(\frac{L}{p} + (1 - f)\right) \cdot (m, \frac{s}{10M} + (1 - m))
\]

Thus we can rewrite our previous equation as follows

\[
\sum_{i=1}^{n} t(P_{1,10M}^i) \cdot \left(\frac{L}{p} + (1 - f)\right) \cdot (m, \frac{s}{10M} + (1 - m)) = t(E)
\]

It is interesting to note that most of the phases do not exhibit a high deal of parallelism. This indicates that we should not get improvements if we keep increasing the number of threads. In Section. VIII we evaluate our model for accuracy and discuss possible issues and shortcomings.

D. Porting the Model

All the above runs were done on the Machine. I as described. One obvious worry is that this model will not be portable and needs to be redone for every machine. While this is a valid concern, we believe the model is structured the only parameter that the model depends on - and is variable is on \( P_{1,10M}^i \). Also, it is evident that \( P_{1,10M}^i \) will change based on the machine it runs on - primarily because the new machine might run at a different frequencies. We hypothesize that we can essentially port the entire model by adding a multiplier \( k \) which could be simply computed for each phase by computing

\[
k^i = \frac{M_{new}(P_{1,10M}^i)}{M_{ref}(P_{1,10M}^i)}
\]

Thus, we can rewrite our generic equation to be

\[
\sum_{i=1}^{n} t(P_{1,10M}^i) \cdot \left(\frac{L}{p} + (1 - f)\right) \cdot (m, \frac{s}{10M} + (1 - m)) \cdot k^i = t(E)
\]

VIII. Evaluation

In this section we evaluate our model by comparing our predictions to actual runtime for \( s = 10, 20, 40, 100 \) and for \( p = 4, 8 \). Essentially, we’d like to conclusively show that our simple model works surprisingly well. Table. X and Table. XI show the comparisons between our predictions and the actual runtime.

To reiterate, we use the model and the equation as developed in Section. VII to make the predictions.
1) To be honest, we are pleasantly surprised at how well our model performs. As we can see in Table XII the error for \( p = 4 \) for \( s = 100M \) which is 10% of the actual data size is less than 4 minutes.

2) One of the major sources of error seems to come from \( P_{13} \) for \( p = 8 \). As we noticed while building the model \( P_{13} \) showed erratic behavior for 8 threads. Infact, when we increased the number of threads to 8, the time actually increased rather than decrease.

3) If one were to remove the approx. 400 second error from \( P_{13} \) for \( p = 8 \) we get much better results.

4) However, it is important to note that even the 734 seconds error for a runtime of 38996 seconds means an accuracy of 99.9% which we believe is extremely good.

5) More importantly, it is evident that increasing number of threads does not significantly improve the results. This is supported by the \( f \) values that we computed in Section VII.

B. The kink in the model

Clearly, there seems to be an issue when we run with 8 threads for \( P_{13} \). However, the important question is why does \( P_{13} \) show the odd behavior with \( p = 8 \). Was the \( p = 8 \) a machine specific anomaly or is it a tool specific issue. We suspect that this particular issue could be because that \( P_{13} \) cannot fully utilize the SMT threads and infact causes contention leading to worsening behavior. However, due to
Table XIII: Cluster Node: Machine Description

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Model</td>
<td>Intel(R) Xeon(R) CPU E5620</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>3.50GHz</td>
</tr>
<tr>
<td>No. of Processors</td>
<td>1</td>
</tr>
<tr>
<td>No. of Cores per Processor</td>
<td>4</td>
</tr>
<tr>
<td>RAM Size</td>
<td>64 GB</td>
</tr>
<tr>
<td>Disk</td>
<td>18 TB</td>
</tr>
</tbody>
</table>

A. Trade-Off: More Threads or More Instances

One secondary goal while profiling was to be able to determine what is better in terms of time. Running a single instance with more threads or running more instances with fewer threads. Concretely, speaking it is better to spawn a single instance of the pipeline with say 4 threads twice as opposed to running two instances with 2 threads each simultaneously.

1) Building an Intuition: Based on our observations and the model, it is apparent that not a lot of the phases in the pipeline have a high degree of parallelizability. Furthermore, for some of the phases it seems that things infact get worse if we go beyond a particular number of threads.

Hence, considering that there is not much parallelizability to be had beyond a particular point there is a strong indication that spawning more instances with fewer threads might be better.

2) Potential Issues: One potential issue with spawning multiple issues of the pipeline is resource contention or scheduler conflicts. However, we did a quick run and saw that there were no significant changes in runtime as compared to a single instance with the same number of threads. Concretely, we saw that two instances of the pipeline running with 2 threads each took the same time as a single instance of the pipeline running with 2 threads. This indicates that resource contention or scheduler conflicts, if present, do not significantly affect the run.

3) Actual Analysis: As can be seen in Table. XIV it is definitely more beneficial to run multiple instances of the pipeline as opposed to a single instance with more threads multiple times. Clearly, there seems to be much advantage in running multiple instances - infact it seems that the more the number of instances you can support and run the better it would be.

TABLE XIV: Analysis 10M: More Threads vs More Instances

<table>
<thead>
<tr>
<th>Threads</th>
<th>Instances</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = 2</td>
<td>2</td>
<td>143613</td>
</tr>
<tr>
<td>p = 4</td>
<td>1</td>
<td>2663</td>
</tr>
<tr>
<td>p = 6</td>
<td>1</td>
<td>3453</td>
</tr>
</tbody>
</table>

B. Prediction for Complete Run

One of the biggest issues for doing a complete run, is that a complete run takes approximately 4-5 days. Hence, it would have been impractical to run analysis on full runs given the high turnaround time. Also, given the limited time at hand, we could not do a complete run. However, given that we have a fancy model and all it only makes sense to do a prediction for the complete run on the Machine I. In Table. XV we make our predictions based on the model developed in Section. VII. We predict that to finish a complete run on we will take 119 hours ie. approx. 5 days.

C. Prediction on Cluster Node

Another interesting problem as mentioned previously is porting the model to other machines. For the same, we proposed the idea of a multiplier \( k \). However, while proposing the multiplier \( k \) we suggested an empirical way of getting \( k^i \) for each of the phases. In this case, however, considering we do not have any relevant data to make an empirical estimate.

1) Estimating \( k \): Given that we do not have data to estimate \( k \) for each phase we would like to find a way to estimate \( k \). Considering, the similar nature of the machines at hand, if we ask the question what could be the potential reasons for the running time to change.

1) Higher Clock Speed
2) Different Memory System
3) IO

However, as we have seen in Section. VI that IO is definitely not a bottleneck and hence not a consideration. Further, in VI-B we also comment on the fact that the cache behavior remains same. Hence, we are potentially left with only the clock speed as a possible source of change in time.
Note, while this might not be entirely accurate, we believe this is a good approximation of what we should expect.

Thus we can compute $k$ as a single factor that can be applied to all phases as

$$k = \frac{f_{\text{Mref}}}{f_{\text{Mnew}}} = 1.834 \times 0.5294$$

Based on this we can estimate that approx. 3 days.

X. Lessons

The overarching goal of the project was to try and suggest spheres for improvement and make suggestions on potential trade-offs and hardware suitability. As explained in the previous sections we think that the tools have been optimally set up - that is the tools are being in used in a way that extract the most possible out of them and have been able to rule out most of the obvious bottlenecks. Further, we believe our ability to model the pipeline fairly accurately reflects that we understand the pipeline - atleast as a black box. Further, it also reflects that whatever we find true for smaller artificial data sets will also be applicable for actual data sets.

This said, in this section we would like to briefly talk about the lessons we learnt and some things to keep in mind for future work with the pipeline.

A. Lessons on Pipeline

1) Run More Instances as Opposed to More Threads: We conclusively show that the pipeline as a whole isn’t exactly parallelizable. Hence, it makes more sense to run multiple instances of the pipeline with lesser threads simultaneously as opposed to a single instance with more threads multiple times.

2) Tools Need Change: One thing that was evident from the entire exercise was that there needs to definitely be an impetus to incorporate more parallelism in the tools. However, there seem to be major issues regarding this. First, the algorithms themselves don’t lend themselves well to parallelism. Second, the bioinformatics community as a whole is incredibly cautious of moving to new tools. The latter is primarily because some of the problems that these tools try to solve are essentially open problems which means different tools solve the same problems differently leading to possibly different results. Thus, unless the tools are proven the community seems reluctant to adopt them.

3) Hardware Investments: The pipeline as a whole doesn’t afford a lot of parallelizability. Hence, there could be two ways that the system could be built up to accomodate the pipelining. The first approach, the one that is currently being followed now, is to buy huge machines with lots of memory and run multiple instances of the pipeline simultaneously. The other approach is to have multiple lesser powerful machines. Hence, instead of having a 100GB quad-core machine one could possibly do away with less powerful dual cores and lesser memory. It would be worthwhile to investigate the latter approach if it could potentially lead to any savings.

B. Lessons in general

1) (More Cores or More Threads) != Better Performance: One of the biggest takeaways was that more cores or more threads needn’t neccessarily mean better performance and hence care should be taken while deciding your configuartion parameters.

2) Simple Models Work: Probably, the biggest and most important part of the project was the model. The simplicity of the model has possibly been the key to it’s performance.

ACKNOWLEDGMENT

We would like to thank Prof. Voelker - the guy who knew the guy who had a project for us, without whose guidance and support much of this wouldn’t have been possible. We would also like to thank Roy Ronnen and also Viraj Deshpande for patiently guiding us through the bioinformatics aspects of the project. Finally, we would like to thank Prof. Snoeren for helping us get key insights into the project and furthermore for the course itself.

REFERENCES


APPENDIX

Graphs of resource usage for the remaining phases of the pipeline for single thread and 4 thread runs on 10M data-set
Fig. 13: Single threaded resource usage for Phase 2: Sort SAM

Fig. 14: 4 threaded resource usage for Phase 2: Sort SAM
Fig. 15: Single threaded resource usage for Phase 4: DeDup Sorting

Fig. 16: 4 threaded resource usage for Phase 4: DeDup Sorting
Fig. 17: Single threaded resource usage for Phase 6: Index DeDup

(b) Memory consumption

Fig. 18: 4 threaded resource usage for Phase 6: Index DeDup
Fig. 19: Single threaded resource usage for Phase 8: Realign InDels

Fig. 20: 4 threaded resource usage for Phase 8: Realign InDels