IMPROVING THE PERFORMANCE OF LONG RUNNING SCIENTIFIC PIPELINES
IN A BIOINFORMATICS PIPELINE PLATFORM

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
HAO TONG

CHICAGO, ILLINOIS
2020
# TABLE OF CONTENTS

LIST OF FIGURES ............................................................... vi

LIST OF TABLES ............................................................... vii

ABSTRACT ........................................................................... viii

1 INTRODUCTION ................................................................. 1

2 BACKGROUND ................................................................. 3
   2.1 Overview of Bioinformatics ........................................... 3
      2.1.1 Brief Introduction to Bioinformatics ......................... 3
      2.1.2 Bioinformatics Pipelines ...................................... 7
      2.1.3 Genomic Databases ............................................. 9
      2.1.4 Data Commons .................................................. 10
      2.1.5 Bioinformatics Pipeline Platforms ......................... 11
   2.2 Technical Review of Bioinformatics Pipeline Platform .......... 12
      2.2.1 Abstract of a Bioinformatics Pipeline Platform .......... 12
      2.2.2 Bioinformatics Pipeline Execution Model .................. 14
      2.2.3 Bioinformatics Pipeline Job Scheduling .................... 16
      2.2.4 Frameworks for Bioinformatics Tools ........................ 18
   2.3 Organization of the GDC and GPAS ................................. 19
      2.3.1 Organization of the GDC ..................................... 19
      2.3.2 Organization of the GPAS ................................... 22
   2.4 Computing in Bioinformatics Platforms ............................ 24
      2.4.1 Computing Paradigms ....................................... 24
      2.4.2 HPC Computing Environment ................................ 26
      2.4.3 Bioinformatics Applications in High Throughput Computing ................................................. 27
      2.4.4 Pipeline/Workflow Scheduling in the Cloud ............... 27
   2.5 Virtual Machine Technology Background .......................... 28
      2.5.1 Overview of Virtual Machines and VM Hypervisors ....... 28
      2.5.2 Important VM technology: Second Level Address Translation ............................................. 30
      2.5.3 Related Evaluations and Researches on SLAT ............. 32

3 PIPELINE PERFORMANCE SUMMARY FOR THE GPAS ............... 35
   3.1 Overview ................................................................... 35
   3.2 Platform Statistics Synthesizing Service biosyncdb and Statistics Database res_biodb ........................................ 35
   3.3 Job Performance (processing rate) on the GPAS’ VM Cluster ........ 38
      3.3.1 Job Performance Definition: processing rate ............... 38
      3.3.2 Performance in the GPAS and Performance Tails .......... 38
      3.3.3 Variation in Non-tail Performance ............................. 41
   3.4 Overall System Status ................................................ 43
# LIST OF FIGURES

2.1 Sequencing Cost Per Human Genome v.s. Moore’s law .......................... 3  
2.2 General Process of Bioinformatics Analysis ........................................ 5  
2.3 Directed graph Presentation of a Pipeline ......................................... 7  
2.4 Presentation of pipeline jobs ............................................................. 9  
2.5 Abstract of Bioinformatics Pipeline Platform ..................................... 13  
2.6 Abstract of a Pipeline Job Execution ............................................... 17  
2.7 System Organization of the GDC ....................................................... 19  
2.8 Graph-oriented Data Model in the GDC ............................................. 20  
2.9 Elasticsearch Data Model for the GDC .............................................. 21  
2.10 System Organization of the GPAS ................................................... 23  
2.11 VM Memory Access with Extended Page Table (EPT) .......................... 31  

3.1 Gaussian Kernel Density Estimations for Processing rate (Seconds/GB) of Four Pipelines on VMs .............................................................. 39  
3.2 Processing rate (Seconds/GB) Comparison for Jobs on Bare Metal Nodes and VMs ....................................................................................... 40  
3.3 Input Size and Processing rate Positively Correlated ............................ 43  
3.4 Input Size and Processing rate Negatively Correlated (1) ...................... 43  
3.5 Input Size and Processing rate Negatively Correlated (2) ...................... 44  
3.6 Abnormal Processing rate Distribution ............................................... 44  
3.7 Accumulated Input Data Consumption in the GPAS .............................. 45  
3.8 Accumulated Output Data Production in the GPAS ............................... 45  
3.9 Total Computing Time by the GPAS in Years ...................................... 46  
3.10 Monthly Average CPUs Allocated in GPAS ....................................... 46  
3.11 Monthly Job Attempts in the GPAS .................................................. 47  
3.12 Pipeline Jobs Success Rate ............................................................... 47  

4.1 CDF of processing rate .......................................................................... 49  
4.2 Processing rate variance on VMs .......................................................... 50  
4.3 VarScan2 experiment ........................................................................... 52  
4.4 CPU utilization of sysbench .................................................................. 53  
4.5 EPT violation ....................................................................................... 56  
4.6 EPT violation monitoring ..................................................................... 57  
4.7 Shell Script to Calculate vCPU Efficiency ........................................... 58  
4.8 Processing rate Gaussian density estimation ....................................... 63  

5.1 Job Delayed Hours due to Retries ....................................................... 69  
5.2 I/O and CPU Utilization of 5 Somatic Variant Calling Jobs Running in VM ................................................................. 71  
5.3 Task-based Scheduling Model .............................................................. 72  
5.4 Prefecthing Data Before Computation Optimization for Tasks ............ 74
LIST OF TABLES

3.1 Synthesized Statistics Database Scheme Overview .................................. 37
4.1 Read latency break-down ........................................................................... 55
4.2 vCPU Efficiency ......................................................................................... 59
4.3 Pros and cons of five mitigation methods .................................................. 60
4.4 Rebooting VM mitigation ........................................................................... 61
4.5 Memory defragmentation ............................................................................ 62
4.6 Running on bare metal ............................................................................... 63
4.7 DNA Alignment Pipeline Experiments ......................................................... 64
5.1 Basic Statistics for Jobs of Somatic Variant Calling Pipeline ....................... 68
5.2 Simulated Execution for Somatic Variant Calling Workflow Jobs ................ 73
5.3 Simulated Execution for Bamfasq-align Workflow Jobs ................................. 73
5.4 Prefetching Optimization for Tasks When the VM is Fresh or Aged ............ 75
5.5 Somatic Sniper Multiple Task Experiment .................................................. 77
ABSTRACT

The Genomic Data Commons (GDC) is a data platform for managing, processing, analyzing, and sharing cancer genomics data. The data processing component of the GDC is called the GDC Pipeline Automation System (GPAS). GPAS currently uses an on-premise cluster that uses virtual machines (VMs) and bare metal machines to run multiple bioinformatics pipelines.

The GPAS has been used in production for over two years and valuable pipeline statistics are scattered in multiple databases across the platform. This dissertation presents a platform-wide statistics collecting service for the GPAS, and based the synthesized statistics, several performance issues have been identified and investigated.

The first performance issue examined is that jobs on VMs exhibit highly varied performance. In particular, there can be a very long tail, with some VMs taking significantly longer than others to execute the same jobs. Through an analysis of jobs statistics and traces, we find that the root cause is the virtual machine memory management layer in the VM hypervisor. When the layer is overwhelmed by intense searches for memory mappings from virtual machine to the physical host, it causes the performance of the VM to degrade.

The second performance issue examined concerns job scheduling. Through an analysis of production statistics, we find that GPAS’ overall work progress can be delayed by days even if only a small percentage of jobs fail. A few other drawbacks of the current simple job scheduling model have been listed with evidence in the dissertation. A more sophisticated task-based scheduling model is proposed in this dissertation.

Lastly, a thorough literature review is presented in this dissertation towards a vision for the GPAS with further improved pipeline performance.
CHAPTER 1
INTRODUCTION

Bioinformatics is an interdisciplinary subject that develops and uses methods and computer programs to help human understand biological data. According to the report [80], biological databases are doubling in size every 15 months. Biological data is usually processed with one or more bioinformatics pipelines or workflows, and a number of distributed computing platforms have been developed and deployed to run these workflows. Since bioinformatics workloads are often both data-intensive and compute-intensive, and since the computing platforms often use virtual machines, a number of performance related issues arise.

This dissertation is concerned with the bioinformatics pipelines and workflows used by the Genomic Data Commons [41] (GDC), which has processed over 2.5 petabytes of genomic data for over 80,000 individuals using a system called the GDC Pipeline Automation System (GPAS). The GPAS uses a mixture of virtual machines and bare metal machines in a large on-premise cluster.

With pipeline execution related data and statistics scattered in multiple databases across the GPAS, it was never an easy job for the GPAS team to get an overview and in-depth analysis on the performance of pipeline jobs. This dissertation presents a statistics synthesizing service (biosyncdb) and creates an all-in-one database (res_biodb) to store all the related statistics across the platform. Based on the statistics in the database, a report on the overall system status and performance since the GPAS officially launched is presented.

Further, inspired by the job statistics, this dissertation reveals a significant VM performance problem that surfaced in the GPAS. We noticed that a significant number of jobs in the GPAS, exhibited much slower performance compared to other jobs of similar types, up to 1,000% degradation. Through an in-depth investigation and experiments, we speculated that the problem could be related to memory fragmentation of aging VMs, causing frequent TLB misses and hurting performance of jobs.
Moreover, this dissertation also reviews the job scheduling model that GPAS provides and finds that the current scheduling is inefficient because job failures bring high extra costs, parallel running jobs causes computing resource under-utilization and contention. A task-based scheduling model is presented and evaluated through simulation and small-scale experiments. This dissertation also discusses further topics in terms of the challenges and requirements to build an efficient task-based scheduling for the GPAS.

Lastly, combined with a thorough literature review on others’ experience of building a bioinformatics application/pipeline platform, this dissertation presents future research possibilities and a vision for the GPAS with further improved pipeline performance.

In summary, the contributions of this dissertation are:

1. A statistics synthesizing service is created for the GPAS and an all-in-one statistics database is created. The production statistics collected from the GPAS in this database facilitates most of the topics in the dissertation and it is invaluable for future research.

2. This dissertation shows how real bioinformatics workloads can cause “aging VMs” after several days and performance degradation by up to 1,000%. Several possible mitigation scenarios are evaluated.

3. The job scheduling model in the GPAS is carefully examined and discussed, and a task-based scheduling model has been proposed with evaluations and discussed.

4. This dissertation presents a literature review on bioinformatics application/pipeline platform and gives a vision for the GPAS with further improved pipeline performance.
CHAPTER 2
BACKGROUND

This chapter covers comprehensive background information regarding to bioinformatics pipeline platforms. The topics would range from bioinformatics, scientific computing, high performance computing to bioinformatics application and pipeline computing frameworks, virtual machine technologies.

2.1 Overview of Bioinformatics

2.1.1 Brief Introduction to Bioinformatics

As biology entered molecular era since mid 1900s, biologists have been inventing and improving their methods of conducting experiments, acquiring genetic data, and analysis from molecular scale. To be more specific, with the understanding of molecular genetics, biologists start to conduct studies focused on DNA and/or RNA samples.

![Cost per Human Genome V.S. Morre’s Law](image)

Figure 2.1: Sequencing Cost Per Human Genome v.s. Moore’s law
Since the invention of DNA sequencing in 1977, tremendous volumes of data has been produced. According to an article [80] in 2001, biological databases were doubling in size every 15 months. Over the decades, the cost for whole human genome has been dropping more than exponentially, as shown in Figure 2.1\(^1\). Comparing to the famous Moore’s law\(^2\) which essentially depicts the cost dropping for increasing computing power over the years, the cost and efficiency of sequencing human genome has outpaced the advancement of computing power. The year 2003 marked the completion of the human genome project, and generation of biological data has further been accelerated thanks to the advancement of sequencing technologies including next-generation sequencing (NGS) [110], and the raw data collected by NGS machines are called sequence reads.

The biggest challenge brought by the surge of biological data is how to achieve high-speed and cost-effective analysis. Utilizing computers is ideal because computers make it possible for people to handle large amount of data and acquire insights among the data efficiently. As a result, in the recent years, a wide range of bioinformatics methods, techniques, algorithms and tools have been developed to extract information and generate human understandable reports from the experimental data [64].

The aims of bioinformatics are three-fold [80]. First, at its simplest bioinformatics organizes data in a way that allows researchers to access existing information and to submit new entries as they are produced; the second aim is to develop tools and resources that aid in the analysis of data; the third aim is to use these tools to analyze the data and interpret the results in a biologically meaningful manner.

The process of bioinformatics analysis for NGS data can be divided into three phases [64], where the first phase is to obtain raw sequence reads from sequencer and pre-process raw data, and the second phase is to assemble the reads to human reference genome, and

---

1. Figure is reproduced from Wetterstrand, K.A. DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP). Available at: [www.genome.gov/sequencingcostsdata](www.genome.gov/sequencingcostsdata)
2. [https://en.wikipedia.org/wiki/Moore%27s_law](https://en.wikipedia.org/wiki/Moore%27s_law)
lastly the third phase is to produces human understandable interpretation for the results of
the second phase.

Figure 2.2: General Process of Bioinformatics Analysis

Take DNA sequence reads as an example, as shown in Figure 2.2, before the data can be
used for analysis, it must be pre-processed to be “cleaned up”. Various tools and packages
can be used for data pre-processing including the Genome Analysis Toolkit (GATK) [83].
The chosen methods of pre-processing data can vary from institute to institute, resulting in
slightly different “clean” data even thought the raw sequence reads are for the same purpose.
This difference could affect subsequent analysis as well.

Once the data has been “cleaned”, it needs to be assembled to the reference, where essen-
tially the sequences are aligned to the reference genome using tools such as Burrows-Wheeler
Aligner (BWA) [73]. After the alignment, in the third phase, one needs to identify variants
which are the locations on the aligned sequences that have different nucleotides from the
reference genome. The procedure to find variants is named “variant calling” in bioinforma-
tics, and a large number of tools can be used including GATK, MuSE [35], MuTect2
[24], VarScan2 [61], SomaticSniper [69], etc. Different tools implement different algorithms
and potentially give different results in terms of variant types and confidence. The variants called from the reads need to be annotated so that human can understand, using tools such as SnpEff [25]. And finally researchers need to conduct further analysis to compare the results between cases and controls.

Another key concept in bioinformatics is experiment strategy, which identifies the type of genomic sequences, e.g., DNA sequencing v.s. RNA sequencing. In spite of the fact that the raw data acquired from the first phase are just characters representing nucleotides, a different experiment strategy such as RNA-sequencing does require another category of tools. Other strategies including targeted or genome sequencing may have different tools or reference database during the study.

In addition, other utility tools would be used in different occasions. Examples include Biobambam [114], SAM tools [74], FastQC [12], Picard Tools [4], for different purposes such as data converting, data viewing, quality control, metrics collecting.

Since the tools are developed by different groups of researchers, written in different languages, serving different purposes, those tools’ workload characteristics such as parallelization capabilities, input/output ratio are quite different too. With wider adoption of SSD storage, researches have shown that not all tools have significant performance improvements when storage is switched to SSDs [70], because some tools only support single thread execution, and they may have been optimized for spinning disks and may not benefit from SSDs by design. Simple benchmarking of some bioinformatics tools will be discussed in a later section.

In short, bioinformatics comes from the needs of handling large volumes of data in a cost-effective, high-speed, high-throughput manner. A variety of analytic tools have been developed to serve all kinds of purposes. Those tools largely contribute to enhancing the understanding of genomic data.
2.1.2 Bioinformatics Pipelines

The typical methodology of bioinformatics involves multiple phases and they are primarily running computer programs other than the first phase of gathering experimental data. To better automate and repeat the process of a study, one needs to arrange all required computer programs in a script-like format so that all the computation tasks can be executed automatically in order, in a bioinformatics pipeline or workflow. In the context of bioinformatics, the terms “pipeline” and “workflow” refer to the same thing and are often used interchangeably.

![Directed graph Presentation of a Pipeline](image)

Figure 2.3: Directed graph Presentation of a Pipeline

To be precise, a pipeline represents a set of computing tasks in a specified order. The output of one task may be the input of another task, whereas some tasks may be independent. Depending on data dependencies of tasks, a pipeline can also be presented as a directed graph as shown in Figure 2.3, where each vertex is representing a computing task, and the edge between two vertices represents the data dependency. Usually, two non-adjacent vertices imply that they could be executed in parallel. It is common that a task in a pipeline references another pipeline. Even though the referenced pipelines could be executed on their own, we call them “sub-pipelines” in the presence of the parent pipeline. We consider all
sub-pipeline executions as part of the ultimate parent pipeline.

To describe a pipeline, there are several emerging workflow language standards. Two of the more popular ones are the Common Workflow Language (CWL) [11] and the Workflow Description Language (WDL) [5]. Specific execution engines have been developed to execute pipelines written in a certain language. CWLtool and Cromwell are the execution engines for CWL and WDL respectively. Both are sufficient to run pipeline jobs. But to fully utilize computing resources on a platform, they rely on designs and policies implemented by the platform. In addition, other third-party execution engines support CWL or WDL, but may not support up-to-date language standards.

Bioinformatics pipelines are data-driven, i.e. whenever a task’s inputs are ready, the task can be performed. Thus, if inputs for certain tasks are intentionally not provided, those tasks will be skipped during pipeline execution. Often, researchers pack all possible tasks into one pipeline, and they can perform slightly different analyses using the same pipeline by using different inputs, as shown in Figure 2.4.

A pipeline job is a pipeline execution that processes a specific set of input data. Thus, jobs are distinguishable by the input data and its pipeline definition. Jobs are expected to perform similarly if they come from the same pipeline, because the computing tasks are similar. However, due to the data-driven nature of pipelines, input data also affects job performance. As shown in Figure 2.4, in addition to data itself, input data involves a few characteristics of which project and experiment are the most important. Project stands for the source (institutions or organizations) of input data, and data from the same project are usually similar. Experiment strategy, e.g. DNA sequencing or RNA sequencing, determines the work required for preprocessing and other computing tasks. As a result, jobs are expected to have similar performance when they come from the same pipeline, and their input data project and experimental strategy are the same. Performance for jobs is typically measured by data processing rate which is the amount of data per unit time that a job can process.
Note that bioinformatics pipelines are fundamentally different from instruction pipelining\(^3\) in computer science contexts. However, executing bioinformatics pipeline jobs on a distributed cluster shares many similarities with instruction pipelining. So far, bioinformatics pipeline scheduling is not as sophisticated as pipelining is, because of the complexity of mixing up all different computation tasks. We will explain current popular pipeline job execution scheme in a later chapter.

### 2.1.3 Genomic Databases

As genomic data creation has reached a phenomenal rate, it becomes invaluable if all the variant information extracted can be shared by all related clinicians and researchers. Bioinfor-

\(^3\) https://en.wikipedia.org/wiki/Instruction_pipelining
mathematics today not only includes computation tasks, but also facilitates genomic data sharing to stimulate the research in molecular biology. This brought up the idea of creating genomic databases, so that all the knowledge and organizations of stored genomic data would be available.

The models of genomic databases are chosen in relational databases, plain text websites, and flat-file databases. Each option has its own advantage and drawbacks: relational databases are the most efficient but they require specialized software and proficiency of computer skills; plain text websites are the simplest, but there is no formal function of query instead of simple web page search; flat-file databases have a modest querying capacity and do not require too much computer skills.

In addition to the storage that a genomic database provides, it also provides indexing and searching functionalities on the stored data, e.g., one will be able to search for a certain variant on a genome in the database without any extra help from other software. However, genomic databases are not associated with how the data is produced or computed.

\section*{2.1.4 Data Commons}

The increasing sizes of genomic data today make it harder and harder for researchers to access, manage, and analyze [45], because they need to find a platform to do computation for those data. The research community has began to work on creating large-scale distributed computing platforms, as a result of which a number of well-known platforms such as Taverna [94], Galaxy [8], Toil [115] have arisen. However, one platform can’t fit all needs [27], different aspects of research require different ways on handling data, thus systems need implementing differently.

Although it is hard to build a uniform platform for all types of bioinformatics study, a uniform platform is inevitably beneficial for the ease of data management and research. The crucial part is data harmonization which enables one single platform to process data in differ-
ent ways. “Data Commons” is introduced, aiming to co-locate data, storage and computing infrastructure with commonly used software services, tools and applications for analyzing and sharing data to create a resource for the research community [40, 116]. It eliminates the burden of purchasing and managing local storage or clusters by letting researchers submit or download data, and analyze data “in-place”.

Genomic Data Commons (GDC) [41] [54] is an example of data commons, which starts with biomedical and genomic data. The data processing component, GDC Pipeline Automation System (GPAS), is the large-scale computing platform that accesses data from GDC and conduct analysis. In addition, there are already a number of biomedical data commons implementations including Bionimbus PDC [45], Sage Bionetworks [37], the Global Alliance [1].

2.1.5 Bioinformatics Pipeline Platforms

Bioinformatics study heavily relies on the computation part, and the complexity of bioinformatics pipelines requires a high performance and robust platform to ensure successful executions of jobs.

Taverna [94] is an early tool designed specially for bioinformatics workflows published in 2003. Along with Taverna, a new workflow language SCUFL was created. The primary goal was to satisfy bioinformaticians’ needs to design workflows from web services. Much of the effort has been put in to build a reliable collection of components to achieve that goal.

Similarly, Galaxy [8] provides the capability for researchers to design workflows through a web-based interface. Besides, it utilizes a built-in light-weighted job running system, and a number of other workload managers including SLURM [121], HTCondor [79], Apache Mesos [57] and Kubernetes [19] to schedule and manage job executions. It has also been developed into different versions to run on major clouds including Amazon Web Services and Microsoft Azure.
Toil [115] is a portable, open-source workflow software that can run workflows on a large scale in cloud or high-performance computing environment. Toil relies on Apache Spark to rapidly deploy services and schedule jobs. One special scheduling policy that Toil provides is file caching and data streaming for jobs. Toil is also able to run on multiple cloud services.

A recent trend is decoupling pipeline execution engine from the computing resources. That is, execution engines for workflow language such as CWL, WDL, NextFlow [32] provide basic execution capability, while computing resource such as VMs management, job submission, job scheduling reply on external services. Cloud providers such as Google Cloud Life Sciences have started to provide tailored cloud service for life sciences to support computation-intensive bioinformatics research.

With the advancement of solutions to executing pipeline jobs, the ultimate goal remains the same, which is to achieve high-throughput and cost-effective computing.

2.2 Technical Review of Bioinformatics Pipeline Platform

Bioinformatics research relies on huge amount of computing resource because of the huge data size and long analysis computation. Various institutions and companies have created a number of platforms for high throughput bioinformatics pipeline execution.

2.2.1 Abstract of a Bioinformatics Pipeline Platform

The general architectures of bioinformatics pipeline platforms are similar. As shown in Figure 2.5, from top to bottom, depending on implementation, the platform will provide a user interface through web app or command line tool. Essentially, the user interface invokes a pipeline execution engine to execute the pipeline job.

At the second level, the pipeline execution engine must know the pipeline specification which may be written in CWL, WDL, NextFlow, etc. Currently, one workflow language may only be best supported by one corresponding execution engine, hence researchers usually
choose the language and execution engine at the same time. Users give parameters to the pipeline execution engine including information such as input data file path and other parameters so that a job can be created. As bioinformatics data tends to be large, the data storage is usually large local file system, or remote system hosted in the cloud.

The interesting part in a bioinformatics pipeline platform is the execution model that the execution engine provides for tasks. Some engines aim to use unmodified tools, so they may provide limited parallelized execution model for the tasks in a pipeline. And others may be implemented with more advanced frameworks such as MapReduce [30] and Spark [122], and this requires that all tools must be re-implemented with the framework, which may not be an easy job for most tools.

Last but not the least, at the bottom layer, the platform must provide options for the execution environment. Most of the platforms are built with distributed system in mind, and get support from many HPC infrastructures and commercial clouds. Some platforms also provide the option to run locally. The key difference between platforms is the degree to which a platform integrates the pipeline execution with the underlying execution environment. A higher-degree integration means the platform itself has more control over the job scheduling on its execution environment and the execution environment can be specially optimized for
2.2.2 Bioinformatics Pipeline Execution Model

As we have explained before, a bioinformatics pipeline is essentially a set of computations that are arranged in a certain order. There have already been hundreds of standalone computation tools developed by various bioinformatics researchers, and the purpose of pipelines is to organize those tools together so that researchers can create a pipeline of tools, and execute and reuse the same computations.

The most common and simplest execution model for pipelines is that the pipeline just organizes unmodified tools, and the pipeline execution engine starts the requested tools directly in order, such as NextFlow, CWLtool, cromwell. The biggest advantage is its simplicity and any bioinformatics tools can be put in a pipeline without any modifications. However, it is poor in scaling up workloads and maximizing parallelism, because it is hard to figure out the ways to optimize for all the various tools individually and in combination.

Other pipeline engines may implement with MapReduce framework or Spark framework:

- MapReduce is a classical distributed task execution model where a job is split into independent small-sized tasks through mapping, and results from tasks get synthesized into final results through reducing. GATK has plenty of tools and small pipelines implemented with MapReduce.

- Spark is a distributed computing framework where more transparent data and job partitioning are provided through resilient distributed dataset (RDD). Since RDD will put all work data set into memory, executions in Spark are much faster. The adoption of MapReduce and Spark would require new tools to be written in those framework, or existing tools to be rewritten.

Other than the framework abstractions, we also introduce three early works of bioinfor-
mathematics pipeline platform:

- Galaxy [8] provides a web user interface so that users can develop workflows easily with pre-provided tools. Galaxy adopts the simple model there tools are executed in separate processes. Those tools are not necessarily same with most popular standalone tools. But Galaxy does implement a standalone job scheduler to avoid long waiting time for jobs because it is built on an HPC cluster.

- ADAM [92] is a research project that leverages Apache Spark and Parquet to achieve high speed up for bioinformatics pipelines. However, most of the analysis tools must be rewritten to be used in Spark framework, which is a huge and difficult work.

- GESALL [106], on the other hand, tries to improve pipeline’s performance without modifying the tools by “Wrapper Technology”. Essentially, they achieve application level parallelism by partitioning input data properly. In three evaluated tools, performance is increased by a large margin, but it also creates slight differences in the final results when compared to original execution model of the tools. Due to the limitation of evaluated tools, it is possible that larger differences may appear for other tools. Besides, under most of contract-based scenarios, it is not realistic for the platform to create different results that the other party may not expect.

To summarize, the design of a pipeline platform and adoption of pipeline execution model must consider the numerous existing analysis tools and the requirement for programming skills of bioinformaticians. Most of the analysis tools are mature and difficult to be correctly re-implemented with another framework. Thus, the stress in bioinformatics pipeline platform is more focused on how to use existing, unmodified tools to easily create a large pipeline execution environment. The workflow languages such as NextFlow, CWL and WDL can all easily pack unmodified tools together and execute on a cloud platform such as Google Cloud,
Amazon Web Services, etc. And to improve the performance of pipelines, it relies more on how to utilize computing resources and scheduling jobs more efficiently.

### 2.2.3 Bioinformatics Pipeline Job Scheduling

Although there are numerous options to build a platform with various pipeline execution engine and the execution environment, in this section, we focus on the type of platforms that build with NextFlow, CWLtool or cromwell, and use a cloud service as the execution environment.

In most cases, cloud services aim to reduce the difficulty for bioinformaticians to create and manage computers in the form of virtual machines. They provide cost-optimized VMs for long running bioinformatics jobs, and batch mode preemptible VMs for fault-tolerant workloads.

As a result, the development and optimization of bioinformatics pipeline engine and computing environments is often separate. There are two strategies for scheduling jobs on a cloud-based cluster. One strategy is to achieve task parallelization on a single worker node for a single job, and then run multiple jobs on the same node to maximize resource utilization and parallelism. The other strategy is to make a single job utilize multiple nodes where independent tasks of the job can run in parallel on separate worker nodes.

In the first strategy where tasks are parallelized on the single node, the pipeline execution engine is responsible for executing the job, and achieving possible parallelism for tasks within the job. And the platform uses APIs of the execution environment to specify computing resource requirement for the job, and the job will be submitted to a worker node transparently. In this case, the computing efficiency largely relies on the implementations of pipeline execution engine. At the time of writing, stable version of CWLtool only supports serialized execution of a pipeline job. CWLtool beta version, cromwell and NextFlow support tasks parallelization within a job.
As shown in Figure 2.6, the pipeline execution engine is able to parallelize as many tasks as possible that use different number of threads within one job. However, due to lack of ability of resource management in a fine granularity, as shown in Figure 2.6, the job would under-utilize CPU resource during the execution of Task 1 if it runs on a 2-CPU VM, while oversubscribe CPU resource during the execution of Task 2 and 3. This would add some difficulties to scheduling when more jobs are added to the same node.

Alternatively, the execution engine can choose to execute one task at a time, which is the same with what the stable version CWLtool supports, thus overall job CPU limit is easy to be set and guaranteed, and appropriate VMs can be allocated for jobs.

To scale up jobs on a large-scale computing cluster, an external job submission and scheduling service can be built to distribute jobs across all the available nodes in the cluster, with the consideration of resource availability on individual node and resource requirements of jobs.

The other strategy depends on the integration with cloud providers that pipeline execution can provide. NextFlow and cromwell have implemented mature integrations with Google Cloud, Amazon Web Services, etc. By providing the configuration or requirement of computing cluster, users can have the pipeline execution engine automatically scale up and down according to current resource requirement from running tasks. As of now, NextFlow and cromwell only support managing one job on one cluster at a time. As a result, although
the tasks of a job are parallelized across an elastic computing cluster, there is no easy way to scale up more jobs on the same cluster.

So far we have only talked about scheduling under the limits of CPUs, when we add more constraints such as memory, I/O intensity, it will become more difficult to achieve efficient job scheduling. As a matter of fact, successful execution and correctness weigh much more than efficiency to the team of the GPAS. Thus, in the GPAS, more than enough computing resource is usually allocated for jobs, ensuring that jobs can complete successfully.

2.2.4 Frameworks for Bioinformatics Tools

Numerous tools have been created to suffice the needs of bioinformatics research. As we mentioned before, more platforms tend to support unmodified bioinformatics tools wrapped in pipelines instead of rewriting their own implementations, bioinformatics developers start to look at more advanced programming frameworks to improve the performance of their tools.

Until now, most mainstream tools are implemented in traditional multi-threaded model in C, Java or Python, such as BWA. Due to the design of algorithms, the benefits from multi-threaded parallelization are not the same across different tools.

Research has tried to turn some traditional tools into ones with more advanced frameworks, such as BWA is turned into SparkBWA [7]. GATK is a collection of useful genome analysis tools, some of which are built with MapReduce in mind, and some others are now tested with Spark framework. However, there are still a number of other tools that can’t be re-implemented within Spark framework.

The complexity brought by implementing tools with frameworks with more scalability is that one needs to find a way to utilize a highly scaled tool on a separate cloud when it is just one of the many tasks in the pipeline job. For tools in Spark framework, one needs to set up a Spark cluster and configure the job for tasks to use that Spark cluster, and it is
likewise to tools in MapReduce framework.

This would add another layer of problem in managing and utilizing another Spark or MapReduce cluster, which can be rather difficult for current implementations of all the pipeline execution engines. More often, those tools would run locally just to utilize the multiple CPUs when they are embedded in a pipeline.

### 2.3 Organization of the GDC and GPAS

#### 2.3.1 Organization of the GDC

The main purpose of the GDC is to serve as a data warehouse where users can conveniently submit and download data. To build a robust and high performance data service, a lot of design decisions are focused on improving data querying performance and data reliability.

![System Organization of the GDC](image)

**Figure 2.7: System Organization of the GDC**

Figure 2.7 shows the organization of the GDC. From top right, as the GDC is mainly servicing external users, it adopts F5 application services to guarantee security, firewall, load
balancing, etc. APIs and a web portal are provided by the GDC so that users are able to submit and download data securely from the GDC. Submitted data is ultimately uploaded to S3-compatible storage, such as Cleversafe and Ceph in the GDC. There are two important design decisions that keep data service from the GDC reliable and fast.

**Graph-oriented data model and Data build services:** The data model is designed to maintain the consistency, integrity, and availability for both existing data and metadata, while accommodating data for multiple ongoing or new projects, as well as providing reliable query capability for both external and internal users.

To meet these requirements, the GDC implements a flexible but robust graph-oriented metadata stores with PostgreSQL, an index service for data files with Signpost, and indexed document stores for API and front end performance in Elasticsearch facilitated by a data build service, ESBuild.

![Graph-oriented Data Model in the GDC](image)

**Figure 2.8: Graph-oriented Data Model in the GDC**

Data entities for Bioinformatics projects are so complicated that the relationships between
different entities are hard to maintain in a traditional relational database. Thus, the graph-oriented data model is created as shown in Figure 2.8. The figure shows a small portion of the entire model, but it is clear to see the relationship between entities. It keeps the relations in the edges which are generic and are transparent to database users, thus the model is flexible and extendible.

The true data files that contain sequencing data and analyzed results are stored in the Cloud (Cleversafe and Ceph), and they are represented as a node with an UUID in the graph data model. Signpost keeps the mapping of UUIDs and the data files’ true locations in the Cloud.

```
file
  ↓
related metadata
  ↓
related files
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
  ↓
case
  ↓
case related information
```

Figure 2.9: Elasticsearch Data Model for the GDC

Simple and flexible as the graph data model is, the downside is poor query performance in PostgreSQL, especially when the website needs to present a whole bucket of information from multiple data entities in the graph data model. As a result, Elasticsearch is used to provide fast queries for metadata for users from website. ESBuild, as a data build service, is thus created to traverse the graph data model and convert the graph model into flat JSON to be indexed into Elasticsearch. To support highly integrated and flexible data queries with fast performance, the Elasticsearch JSON schemata are designed in a highly denormalized fashion, with related data entities pre-joined via subdocument embedding. File-centric and
Case-centric indexes are built to allow users to search and retrieve files and participants respectively. Figure 2.9 shows a representation for file-centric index built by ESBuild. File related metadata is joined from PostgreSQL by ESBuild and embedded directly in the same document with file. Besides, other related files are also embedded as subdocuments. Thus, when the index in Elasticsearch is queried, it avoids a lot expensive join operations in the relational database.

Currently, the GDC graph database takes up 62 GB on disk, thus the ESBuild process can be resource and computation intensive. There are a lot of performance improvements for ESBuild have been done to enable fast caching and project-based parallelization of the process. This is discussed in the next section.

**Data backup and recovery:** As the data is important to users, the other major responsibility is to keep those data safe. The GDC currently stores all data one copy on-prem, one copy in Amazon Data backup services (S3/IA and Glacier), and one copy on BlackPearl tapes.

Frequently accessed data is maintained in Amazon S3 Standard Service, and infrequently accessed data is maintained in Amazon S3 Standard IA, retired, deprecated, or archived data is maintained in Amazon Glacier. The GDC uses AWS Object Lifecycle Management to manage rules for moving data objects between storage tiers.

### 2.3.2 Organization of the GPAS

As shown in Figure 2.10, the system consists of a few micro services, implemented by both community open source software and in-house software.

1. **Cloud Storage:** Cloud storage is the data warehouse where the majority of raw data in the GDC is stored. GPAS uses cloud storage to conveniently download and process raw data and upload analyzed results. The GDC is designed to be S3 compatible, so users can select their favorite S3 data services. In GPAS, data is only downloaded through a download
client when computing workflows in the system are created to analyze data. Data size for a analysis job ranges from a few GBs to hundreds or even thousands of GBs. Currently, in the GDC, the total data volume stored in cloud storage is 7.2 PB including both public data and internal unpublished data.

2. **Metadata Services**: This part is simplified in the figure, but they are the bridge between the GPAS and the GDC. Metadata database for the GPAS is created through a few steps. Firstly, it clones the database from the GDC’s database, and later the cloned database is reduced and only necessary nodes that pipeline jobs need are kept. This final database is also called “BioGraph” in the GPAS. Same as in the GDC, metadata is stored within a graph Metadata does not have a critical impact on workflow performance.

3. **Job Creation Services**: Job creation services mark the starting point of a job’s life. It consists of two components: “Work Creator” and “BioJSS” (Bioinformatics Job Scheduling Service). Work Creator queries the BioGraph for new jobs, and it combines data nodes and pipeline nodes to create CWL jobs. BioJSS assembles new jobs and hands them to the cluster manager. When jobs are submitted to the cluster, BioJSS periodically polls cluster
manager to get job status and records it in database.

4. **SLURM**: GPAS chooses SLURM as the cluster manager for VM Pool. Although SLURM itself has its own scheduling algorithms, it is merely a job distributor in the GPAS, with little control over the scheduling of jobs.

5. **VM Pool**: Currently in the GPAS, the VM Pool is on-premise, and is managed by OpenStack. The policy of VM allocation is simply one VM per physical node. Almost all pipeline jobs until recent were executed in VMs. The VM Pool includes VMs residing on a variety physical nodes, with CPU ranging from 40 to 48 cores, memory ranging from 500 to 900 GBs, and storage ranging from hard disk RAID to SSD RAID.

### 2.4 Computing in Bioinformatics Platforms

The large size of bioinformatics data and the large volume of bioinformatics pipeline jobs requires the platform to provide tremendous computing power. Thus, a large-scale high performance computing cluster is only used to support the bioinformatics platform. It is important to understand the type of computing for bioinformatics pipeline platform.

#### 2.4.1 Computing Paradigms

Modern computers have provided powerful computation capabilities, however, complex problems in various fields such as science, engineering and business and etc. require tremendously more computation power. Clusters of computers are deployed for this purpose. Computations in this scenario will try to utilize the computing power from multiple computers to finish complete workloads.

Depending on the type and time required by a computation job, three paradigms are proposed:

- **High performance computing (HPC)** classifies computations that require large amounts of computing power for their workloads, and use intensive communications
between each computation task on distinct computers via Message Passing Interface (MPI). These tasks are considered to be tightly coupled and usually they can be complete in a short period of time (within seconds). Thus, the latency of communications via MPI is crucial. The performance of applications in HPC are measured by floating point operations per second (FLOPS).

- **High-throughput computing (HTC)** [79] classifies computations that also require large amounts of computing power. However, the tasks for a computation job are usually loosely coupled or independent, and require long periods of time. Scientists in HTC are more interested in number of floating point operations per week or per month that they can extract from the underlying computing cluster, rather than the FLOPS that the cluster can provide.

- **Many-task computing (MTC)** [101] is proposed to bridge the gap between HPC and HTC, and it is aimed for computations that consist of a larger number of tasks, each of which uses relatively short per task execution times (seconds to minutes) while is data-intensive (tens of MB of I/O per CPU second). Tasks in MTC can be individually scheduled on many different computing resources, and their performance is measured by FLOPS, tasks per second, MB per second and etc.

It is important to appropriately classify an application into one of the three computing paradigms, because benchmarking and metrics are quite different for them. However, since the applications from all the three paradigms share the same requirement for computing resources of their underlying environments, the discussion for computing environment is usually in the field of HPC.
2.4.2 HPC Computing Environment

HPC was initially introduced as the form of supercomputers that have a high level of performance as compared to a general-purpose computer. Supercomputers are powerful but expensive, thus they are only suitable for some specific applications.

Grid computing, distributed computing, and cloud computing were then introduced to provide more affordable and flexible computing power. The boundaries in between are, however, blurry in some cases. Grid computing environment is usually built for applications that are embarrassingly parallel [33] and do not need communications between the parallel running tasks. Distributed computing [14] aims to achieve more control over the assignment of tasks to distributed resources. Cloud computing, however, attempts to provide HPC-as-a-service through virtualization technologies so that HPC users can enjoy the benefits of scalability in cluster size, resources on-demand, and inexpensive cost. A cluster of VMs acquired from a cloud platform can be easily used for embarrassingly parallel applications as in a grid computing on-premise cluster, or for tasks that need more control over assignment as in a distributed computing on-premise cluster, through installing different stacks of softwares. Moreover, because of the features of security and flexibility that VMs provide, more on-premise clusters have been deploying worker nodes with VMs, just as in the VM pool in the GPAS shown in Figure 2.10.

Recent research interests on cloud-based HPC computing environment lie on the performance of a cloud provider can provide comparing to a traditional on-premise cluster. Because of the probable overhead brought by VMs and performance impact brought by multitenancy which cloud providers commonly use, performance variation and predictability in the cloud has been studied [71]. Besides, multiple studies [53, 63] on Amazon Web Services and Microsoft Azure have proved that only the computation-optimized instances provided by the cloud providers, instead of general-purpose instances, are suitable for building a HPC computing cluster. And the best cloud solution for a HPC platform varies from case to case.
in terms of the applications to run on the platform. Besides, hybrid environments where workloads can run on on-premise cluster and can also utilize on-demand resources from a cloud platform becomes a trend in the industry. There come more challenges in order to get the best performance from the unknown underlying of the cloud and build a cost-efficient hybrid environment [90].

### 2.4.3 Bioinformatics Applications in High Throughput Computing

Because of the long running time for bioinformatics applications, bioinformatics applications are usually regard as in the category of high throughput computing.

HTCondor [79] was the first framework implemented for HTC purposes, and it stresses on the amount of computing power that the applications can utilize in the platform. The key mechanism in HTC is an efficient algorithm to satisfy resource requirements from tasks in a distributed environment [102]. With efficient resource management at the center place of HTC, resource monitoring [56] in HTC becomes important. A more recent HTC system, HTCaaS [60] expands the resource management to a combination of grids and clouds.

It has been pointed out that HTC is very helpful for bioinformatics applications that tend to take months to complete [113], and to achieve high data throughput for bioinformatics is a very complex problem. While the research on bioinformatics in the HTC context is very few, HTCondor has been modelled into a few systems for bioinformatics computing solutions [112, 59]. The Discovery Net system [105], and galaxy [8] platform are also solutions that try to bring HTC to bioinformatics.

### 2.4.4 Pipeline/Workflow Scheduling in the Cloud

Pipelines or workflows, as a much more complicated workloads than single applications, raised an interesting subject of scheduling in the cloud where underlying hardware status is usually unknown.
The performance fluctuations on the VMs provided by the cloud has attracted considerable attentions [119] because it is difficult to guarantee user-perceived performance. Thus, workflow scheduling needs try to become fluctuation-aware and make predictive decisions [75].

With the cloud allowing users to pay only for what they use, schedulers have to find a balance between time and cost. Majority of research in scheduling are trying to minimize the cost while meeting a deadline constraint [13, 103]. Thus, it is also important to make accurate predictions or estimates for workflows, and various approaches [49, 55, 88] have been adopted. Another category of scheduling proposals focus on reducing costs to meet quality of service requirements for workflows [84, 81, 100, 23]. Besides, a large number of scheduling algorithms [48, 118, 103, 120] are task-based to reach finer granularity scheduling results.

For more comprehensive taxonomies on scheduling algorithms for scientific workflows, please refer to these two surveys [104, 47].

2.5 Virtual Machine Technology Background

2.5.1 Overview of Virtual Machines and VM Hypervisors

Large as a computation platform can be, people usually adopt virtual machine technology for rapid deployment, fast development and test, security requirement over a huge number of bare metal machines.

As shown in Figure, the architecture of VM includes the VM itself, a VM manager software, and host hardware. The VM manager software is often called VM hypervisor which is responsible to isolate VMs from each other, distribute hardware resources to individual VMs, and help execute instructions from VMs.

Virtual machine technology itself is a large topic and we don’t want to dive too deep
into it. Instead, we will focus on VM hypervisors and VM memory management mechanism since they are most concerned with VMs’ performance.

Based on the layer where VM hypervisors stand, they are often classified into two categories. Type 1 hypervisors run directly on the system hardware, create virtualized resource for guest operating systems directly on hardware. While type 2 hypervisors run on a host operating system that provides virtualization services, such as I/O device support and memory management.

Type 2 hypervisors require a host OS to be installed first on the hardware, and they manage resources with assisted from the host OS. The advantages of this type are it supports a wide range of hardware thanks for that the host OS creates an additional layer of resource abstraction, and the installation of hypervisor and VMs are much easier. However, VMs’ performance is reduced in a degree because the additional layer of host OS, besides, the stability of VMs largely depends on the stability of host OS.

On the other hand, type 1 hypervisors directly run on hardware, the distance between VMs and the host’s hardware resources is small, performance of VMs managed by type 1 hypervisors are usually higher.

Type 1 hypervisors include VMware ESX, Microsoft Hyper-V, and many Xen variants, and type 2 hypervisors include VMware Workstation, VirtualBox, etc.

A popular hypervisor choice is Linux’s Kernel-based Virtual Machine (KVM) because it ships with every Linux distribution. KVM can be classified to type 1 because it effectively convert the whole host operating system to a hypervisor. However, virtualized resources for VMs are still derived from the abstraction of host OS, so KVM can also be classified as type 2.

In this paper, we classify KVM as a type 2 hypervisor due to its memory management mechanism as would be explained in the next section. KVM maintains a two-level memory address mapping from guest OS to host OS to hardware. However, type 1 hypervisors such
as Xen uses direct paging that directly mapping guest OS memory address to hardware memory address, which effectively reduces the cost of memory access for VMs.

2.5.2 Important VM technology: Second Level Address Translation

There are many notable technologies in VM to improve VM's performance and security. As the context in this paper is mainly concerned with bioinformatics workloads that often require large amount of memory, the performance of VM memory management is vital to the performance of workloads. For type 2 hypervisors, such as KVM that OpenStack relies on, memory access performance is optimized by Second Level Address Translation (SLAT). The implementation of SLAT in Intel processors is called “Extended Page Table (EPT)”, and we will use “EPT” directly since Intel processors are commonly used in GPAS cluster.

For an application to work properly in operating system, operating system ensures the translation between application’s memory address space (Virtual Address, VA) and machine physical address (PA) space. Virtual machines add an additional guest operating system (Guest OS) layer to the stack, with virtual machine hypervisor taking the layer of application on physical machine, and application that needs to run being put inside the virtual machine. Virtual machine hypervisor keeps application from being aware of its environment, and takes care of translating application’s VA (GVA) to Guest OS’s PA (GPA), then translating GPA to the address in virtual machine manager’s virtual address space on the host (HVA), and finally translating it to true physical address on the host (HPA) assisted by Host OS.

This two-layer abstract introduces three translation mappings for each memory access. Initially, VM hypervisors use software-based memory virtualization by implementing shadow page tables that store the mappings from GVA to HPA for each guest process. This approach does not introduce address translation overhead for normal memory accesses in the virtual machines after the shadow page tables are set up, because the translation look-aside buffer (TLB) on the processor caches GVA to HPA mappings. However, since the hypervisor
needs to maintain a shadow page table for each guest process, the memory requirement is higher. Besides, the overhead and implementation difficulties that come from keeping synchronization between shadow page tables and guest page table are also increased.

Figure 2.11: VM Memory Access with Extended Page Table (EPT)

In order to increase memory performance further for VM in modern computers, Intel and AMD both implemented their solutions for second level address translation (SLAT) with Intel VT-x since 2005 and AMD-V since 2006, respectively. This approach is hardware-based memory virtualization. For Intel, Extended Page Table (EPT) is introduced so that host OS builds this table to store GPA to HPA mappings. And Intel processors use tags on Translation Look-aside Buffer (TLB) to avoid flushing the whole TLB for each context switch, thus EPT entries can be stored in the TLB without frequent flushing penalties. During virtualization, VM can access the TLB directly. As shown in Figure 2.11, EPT entries in the TLB cache GPA to HPA mappings for the guest, so when guest OS is accessing memory, as long as the mappings are in the TLB, virtual CPUs can directly handle the address translation.
for the VM, thus avoiding switches between VM and host, shown as (1) in Figure 2.11, also avoiding synchronizing address translations between mappings and guest page table.

However, when guest OS is trying to access a GPA that does not have a mapping in the TLB, a TLB occurs and the system needs to switch from VM to host, which is referred as a VM exit event and is expensive, shown as (2). The host OS intercepts the GPA, and looks it up in the EPT in virtual memory management unit, shown as (3).

Such a TLB miss that occurs in (2) is also referred as a EPT violation. The host must do extra work to look for the mapping. This mechanism is easy to achieve because their is only one table for one VM, compared to per guest process shadow page table mentioned earlier.

This mechanism works well with good spatial locality workloads on relative small memory as TLB size is limited. However, if the locality is bad, in the worst case, accessing a GVA can cause 24 table accesses as a total in guest and host in 4K-page setup. Because both guest page table and host mapping table are implemented in 4 levels, a GVA needs to access four extra GVAs for each level of its guest page table. In the worst case, each of the extra GVA causes a EPT violation, then each of them will cause the host to walk four levels of the mapping table. For the case of KVM implementation for EPT management, only one lock is designed for the virtual MMU mapping table, so when multiple processes in Guest OS are causing EPT violations, there will be serious contention on the lock.

2.5.3 Related Evaluations and Researches on SLAT

Through the years, there are handful evaluations based on benchmarks for the VM performance with the adoption of SLAT, or Intel’s EPT. And they are mainly from virtualization vendors such as VMware.

In 2009, an evaluation [21] conducts comparison between software-based memory virtualization (shadow page tables) and hardware-based memory virtualization (EPT) memory virtualization. Benchmarks that are used in the evaluation include Kernel Microbenchmarks,
Apache Compile, Oracle Swingbench, Specjbb 2005, Order-Entry benchmark, SQL Server Database Hammer, Citrix XenApp. EPT is found to have significant performance increase over shadow page tables except SPECjbb2005 which is an industry-standard server-side Java benchmark. It is because Java’s usage of the heap and associated garbage collection increases TLB misses. Huge page can be used to reduce TLB misses and it is found to increase performance in the evaluation.

Another performance analysis [22] in 2013 finds a considerable performance drop for EPT comparing to native performance on a bare metal node. The workloads evaluated includes OLTP workload, Hadoop, and latency-sensitive applications. And performance is evaluation with hardware event counters, and cycle-accurate measurements. The evaluation finds that most of the virtualization overheads are small, but EPT costs indeed bring a large overhead to virtual machines.

A more recent study [85] in 2016 analyses memory load and cycles in hardware events. CPU-bound benchmarks and Database Benchmark are evaluated in this study. This study finds that the performance impact of TLB misses from EPT is not severe thanks to the increasingly larger TLBs and caches on modern CPUs. Yet another study [111] in 2016 finds that performance increase from EPT is almost negligible comparing to software-based solutions, through the SPEC CPU2006 [46] benchmarks.

However, in our experience, we find that there indeed is no obvious performance impact in the short term, however, with VM living longer, the impact becomes harder to ignore.

All the evaluations among other researches [51, 89] agree that adoption of huge page can reduce TLB misses. However, with the memory size of a server and memory usage of applications both increasing, the demand for a even larger page size is emerging as well. It is found that big-memory workloads that use a wide range of memory such as databases, in-memory caches will still suffer from TLB misses even with huge pages [18].

Besides, the discouragements of using huge pages for VMs are because it amplifies memory
allocation requests when VMs are using 4K pages, and it defeats page sharing thus huge pages must be broken into small pages [43]. This is one of the reasons that lead to the problem of page splintering. Some of the research in huge pages are solving this problem that may mitigate the benefits of using huge pages [99, 98]. However, the study [85] finds little impact from page splintering. And other researches focus on how to make huge pages more flexible by tracking page movements [96] dynamic page sizes [9, 76, 39, 42, 95] dynamic huge page based memory allocation and ballooning [109, 72, 50] and even fundamental redesigns [67].

The most effective but also the hardest approach is to reduce TLB misses. Some of the researches focus on introducing translation cache structures [20, 15], limited direct paging for virtual machine [18, 38]. And other researches [16, 10, 58] try to implement a speculation mechanism to reduce TLB misses.
CHAPTER 3

PIPELINE PERFORMANCE SUMMARY FOR THE GPAS

3.1 Overview

The organization of the GPAS intuitively scatters job execution related information across multiple databases in the GPAS, which makes it uneasy job for the team to review the system status and performance on the GPAS.

This chapter presents a platform statistics synthesizing service biosyncdb that pulls statistics into an all-in-one database. Based on this database, the current status of pipeline performance and system achievements are illustrated.

This database serves as the most critical role in this dissertation and it is invaluable for future researches on the GPAS’ performance.

3.2 Platform Statistics Synthesizing Service biosyncdb and Statistics Database res_biodb

In §2.3.2, the organization of the GPAS is described and a major drawback of the loosely coupled components is that data related to pipeline job performance is scattered in multiple databases. There are three databases and one data service existing in the GPAS:

1. **Graph-oriented database in PostgreSQL**: Relationships between projects, pipelines/workflows, jobs, input data, output data are recorded in the representation of edges. However, no further information than UUIDs or names is stored.

2. **BioJSS Database**: as mentioned in §2.3.2, the “BioJSS” service is handles the scheduling and dispatching of jobs, and it records important information of input/output documentation, pipeline configuration and job status at its backend database. However, it replaces failed job records if the job is attempted again.
3. **SLURM Database**: SLURM database records all necessary information for a job in terms of running status, resource allocation, submit time, start time, end time, and etc. Besides, it keeps failed job records as well.

4. **File Indexing Service**: explained in §2.3.1, Signpost serves as an index service to provide necessary information for a file if the file’s UUID is given, such as file name, downloading url in the cloud storage, and data size.

Naturally, it is impossible to measure the performance of a system without knowing the data size that it processes. Besides, as explained in §2.1.2, performance of jobs is only comparable if the jobs are in the same projects, of the same pipeline, and processing data with the same experimental strategy. Thus, to assess the performance of GPAS, the data from multiple sources must be combined.

One of the major contributions in this dissertation is that a platform-wide statistics synthesizing service (`biosyncdb`) is created, and a statistics database (`res_biodb`) is thus created.

Table 3.1 shows an overview of the database scheme for the `res_biodb`. The algorithm of `biosyncdb` is very straightforward, and it pulls data from the database as shown in `Source` column for each table in the `res_biodb` in order. The performance of `biosyncdb` is constrained by two factors:

1. The APIs provided by the graph-oriented database developed by the GDC are limited and do not allow to perform arbitrary JOIN operations. Thus, in order to get information such as how many jobs are using the same input file, instead of querying file table and joining with job table, `biosyncdb` can only traverse all the job nodes and build a map to store the affiliation between files and jobs.

2. The backend database of file indexing service is not exposed for security considerations. And the query `http` point is also used for job execution purpose in the GPAS. In order
<table>
<thead>
<tr>
<th>Table name</th>
<th>Purpose</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>Program and project compose the project id</td>
<td>1. graph-oriented database</td>
</tr>
<tr>
<td>project</td>
<td>As above</td>
<td>1. graph-oriented database</td>
</tr>
<tr>
<td>workflow</td>
<td>Workflow name, UUID information</td>
<td>1. graph-oriented database</td>
</tr>
<tr>
<td>workflow_biojss_info</td>
<td>Additional git repository information</td>
<td>2. BioJSS database</td>
</tr>
<tr>
<td>job</td>
<td>Job UUID, workflow affiliation</td>
<td>1. graph-oriented database</td>
</tr>
<tr>
<td>job_graph_info</td>
<td>additional information on experimental strategy, project affiliation.</td>
<td>1. graph-oriented database</td>
</tr>
<tr>
<td>job_biojss_info</td>
<td>additional information on job input/output document, etc.</td>
<td>2. BioJSS database</td>
</tr>
<tr>
<td>datafile</td>
<td>UUID, data size, etc.</td>
<td>1. graph-oriented database,</td>
</tr>
<tr>
<td>datafile_job</td>
<td>Job and data file affiliation</td>
<td>4. file indexing service</td>
</tr>
<tr>
<td>job_slurm_info</td>
<td>Job states, submit time, start time, end time, and resource allocation</td>
<td>3. SLURM database</td>
</tr>
<tr>
<td>job_metrics</td>
<td>SLURM recorded average resource utilization metrics</td>
<td>3. SLURM database</td>
</tr>
<tr>
<td>job_to_job</td>
<td>Dependence between jobs</td>
<td>1. graph-oriented database</td>
</tr>
<tr>
<td>workflow_to_workflow</td>
<td>Dependence between workflows</td>
<td>1. graph-oriented database</td>
</tr>
<tr>
<td>last_sync</td>
<td>Last timestamp that the service runs incrementally</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Synthesized Statistics Database Scheme Overview

not to swarm the query point, biosyncdb limits the frequency to query information such as data size from the file indexing service.

Similar to ESBuild in the GDC, it also takes longer and longer time for biosyncdb to synthesize all the data. To optimize the performance, biosyncdb synchronizes data in batches and only checks newly changed data since last synthesizing. Currently, biosyncdb runs once a day and it takes around one hour to finish synthesizing the new data since last day.

Currently, res_biodb stores all the available and necessary information for more than
904,484 data files, 362,400 job attempts (including successes, failures, test attempts that are not presented in the graph-oriented database), and the total size of the database is merely 2.3 GB. This database is valuable because most of the remaining discussion in this dissertation is based on the data stored in `res_biodb`, and it provides enough data to get an overall impression on the status of the GPAS. Besides, other unused data such as `job_to_job` and `workflow_to_workflow` is valuable to future research in the relationship between jobs and workflows.

### 3.3 Job Performance (*processing rate*) on the GPAS’ VM Cluster

#### 3.3.1 Job Performance Definition: *processing rate*

The majority of the jobs in the GPAS process a large amount of input data and take a long time to complete. Thus, a simple metric for understanding job performance and its degradation is the processing rate:

\[
\text{processing rate} = \frac{\text{job execution time}}{\text{input data size}}
\]  

The unit of *processing rate* is seconds/GB or hours/GB, thus a larger *processing rate* value means the performance is worse (as more time is needed for processing one GB input data).

#### 3.3.2 Performance in the GPAS and Performance Tails

In this section, we present *processing rate* for projects that have the most pipeline jobs.

In this section, an overview of the job performance in the GPAS is given and an critical issue of performance is discussed.

Shown in Figure 3.1, Gaussian kernel density estimation are applied to the *processing rate* for four pipelines. As expected, the distribution of *processing rate* roughly complies
with a Normal/Gaussian distribution. However, what concerns the most is that the slowest processing rate is abnormally bad, as shown to the far right of the graphs.

In the Figure 3.1, the vertical line shows where the 95th percentile processing rate resides, and to the far right of the vertical line of each graph, there are long “tail” reaching to even longer time required to process one GB input data. We refer this as “tail processing rate” for a pipeline. Jobs that suffer from tail processing rate tend to contribute more to the total execution time for all the jobs for a pipeline. In the case of these four pipelines, jobs that are slower than 95% of all the jobs, contribute to the total execution time for all the jobs by 11.4%, 15.2%, 22.2% and 13.8%, respectively. In the worst case, tail jobs in another pipeline contribute 53.1% for the total execution time.

This tail processing rate issue will be discussed in detail in Chapter §4.

One of the methods to reduce the tail effects is to run jobs on physical hosts directly instead of using VMs. This type of computing nodes are called as “bare metal nodes”. Since June 2019, the GPAS has started to run a small portion of jobs on bare metal nodes, as it has been proved that bare metal nodes are more reliable in terms of providing stable performance.
for jobs. Figure 3.2 provides more intuitive comparison of processing rate between jobs on VMs and jobs on bare metal nodes, where jobs on VM do have longer tails than jobs on bare metal nodes. Jobs on bare metal nodes that are slower than 95% of all the jobs, contribute to the total execution time for all the jobs by 7.1%, 6.3%, 6.8% and 7.4%, respectively. However, jobs of the same pipeline that are slower than 95% on VMs contribute 8.0%, 24.8%, 6.2% and 10.4%, respectively. For the pipeline mentioned earlier where slow jobs on VMs contribute 53.1% of execution time, the jobs of this pipeline that run on bare metal nodes contributes only 8.1%. More details will be discussed in Chapter §4.

Figure 3.2: Processing rate (Seconds/GB) Comparison for Jobs on Bare Metal Nodes and VMs

Since the main goal of GPAS is to achieve high throughput for pipelines, the most urgent task for us is to eliminate the tail jobs. One approach that we have started to adopt is distributing jobs on the bare metal nodes directly. We are also actively exploring other options to run jobs more efficiently in the VMs.
3.3.3 Variation in Non-tail Performance

As shown in the previous section, jobs on bare metal nodes exhibit much more stable performance than those on VMs (at least for the VMs with current configurations in the GPAS), and the distribution of processing rate on bare metal node is more centered to the mean processing rate, and 95th processing rate is more close to the center as well. However, there still exists variation. This section discusses some factors that correlated with the variation of processing rate. All the statistics are collected from bare metal nodes to avoid interference from unstable VM performance.

With a linear regression and significance test analysis, it is found that among other parameters that are known before a job starts such as number of input files, number of CPUs allocated, etc., input data size exhibits the most correlation with processing rate. Limited by the small number of jobs running on bare metal nodes, there are only a few pipelines that are suitable for regression analysis, and four pipelines are discussed in this section.

To better illustrate the correlation between input size and processing rate, jobs are sorted by their processing rate, and are divided into 20 buckets according to the percentile of processing rate. Average input size of each bucket is shown in Figure 3.3, 3.4, 3.5. There can be either positive or negative correlation existing between input size and processing rate for different pipelines. Definitions of the pipelines have been investigated to help explain the correlations. Hence, at least three principles are concluded in this analysis:

1. **Positive correlation** (larger input size means slower/larger processing rate): As shown in Figure 3.3, processing rate is positively correlated with input size. At the same time, according to the graph to the right, processing rate of this pipeline is generally very large, which is more than 1500 seconds/GB. Indeed, the workloads in this pipeline contain applications to align sequences to a human genomic database, which is string-searching in a nutshell. Thus, it is common for the pipeline to have $O(N^2)$
computation complexity, and it exhibits positive correlation between input data size and processing rate.

2. **Negative correlation** (larger input size means faster/smaller processing rate): On the contrary to the previous one, Figure 3.4 and Figure 3.5 show that processing rate is negatively correlated with input size. It brings to the attention that processing rate in these two pipelines are much faster than the jobs in the previous one. By inspecting the definitions of these two pipelines, we find that these two pipelines are designed to do simple work where they download and extract files. With a high bandwidth network and I/O, processing rate is higher. However, since the execution time for such pipeline jobs is short, the overhead coming from spawning up containers and other components can not be ignored. As a result, smaller input size leads to worse processing rate.

3. **External factors interference**: In addition, there are some cases that may not comply with previous principles. In Figure 3.6, it shows two different distribution for job processing rate of this pipeline. Indeed, in the significance test, jobs in this pipeline shows a much higher p-value (0.038), indicating there is only 96.2% probability that processing rate is correlated with input size (for other pipelines, the probability is larger than 99.999%). By inspecting job logs, it is found that jobs at the right peak (slower jobs) spent significantly longer time in downloading data. It implied that there might have been network congestion in the cluster during that time. Hence, in rare cases, we need to take external environment into account to understand the processing rate of jobs.

To conclude, by statistical analysis of the jobs, although there are some degrees of variation, we find that processing rate for pipelines is highly correlated with a job’s input size. And, the exact correlation is depended on the workloads associated with the pipeline. Sometimes, the processing rate might be interfered by external factors during some parts in the
job. It is important to understand the definition of the job so that good explanation can be given.

Chapter §5 will discuss the importance of defining finer-granularity pipeline scheduling mechanism so that it will be easier to model processing rate of jobs.

### 3.4 Overall System Status

Thanks to the data collected by biosyncdb, here we present an overview on the system status of the GPAS, which has not been concluded before.

#### 3.4.1 Sustainable Data Processing

Starting March 2018, the GPAS has been devoted into large-scale bioinformatics pipeline computing. Figure 3.7 and 3.8 show the accumulated input consumption and output production over the months. The first few months included development, testing and small-scale
3.4.2 Sustainable Computing Power

Figure 3.9 shows the accumulated computation time that the GPAS has utilized in the cluster. Over the two years, 200 job computing years have been utilized for computation task.

A large room for improvement on computing power utilization exists due to the design of job scheduling in the GPAS. As shown in Figure 3.10, the CPU allocation exposes a lot
of inefficiencies in the current design of the GPAS. This is resulted from various reasons. Current practice in the GPAS is that it never allocates all the CPUs to the jobs on a VM. This is to ensure that there is no contention on CPUs because most of the applications in bioinformatics are computation-intensive. The other reason is that jobs in the GPAS are contract-based. In most cases, pipelines must get permission before they can be submitted into the GPAS.

More details will be discussed in Chapter 5 on what measures can be adopted to improve the efficiency of job scheduling.
Figure 3.9: Total Computing Time by the GPAS in Years

Figure 3.10: Monthly Average CPUs Allocated in GPAS

3.4.3 Monthly Job Execution Details

Figure 3.11 shows the number of job attempts for each month, and it shows dramatic changes in number of job attempts over the months. This supports the variance in monthly CPUs allocation in the previous section. Admittedly, current implementation of the GPAS still needs improvements on finer-granularity resource management. Also, the team is actively exploring the possibility of running non-priority jobs in the background to improve resource utilization.

Figure 3.12 shows the success rate for jobs in each month. The GPAS managed to maintain around 90% success rate for most of the months. However, sometimes it might meet issues that interrupted production jobs or test jobs, which is also reflected in the
Starting June 2019, it is found that bare metal nodes can provide much more stable performance than the VMs, thus the GPAS phased out a small portion of VMs and used bare metal nodes directly. The green bold line in the figure shows that the success rate on bare metal is relatively high. Since jobs that run on bare metal nodes are in a small number, the overall success rate is still dominated by jobs that run on VMs.

3.5 Summary

This chapter presents the all-in-one database (res_biodb) created to facilitates the performance study on the GPAS, and a definition to measure job performance is given. Based on the data collected in res_biodb, this chapter presents various metrics that can be acquired:
• An issue is exposed that some jobs that run on VMs in the GPAS exhibit abnormally poor performance (tail performance).

• Variation in non-tail performance is studied and the input size is found to be the most significant factor correlated with the job performance.

• A system status overview on the two-year production of GPAS is presented to showcase the achievements and identify space to improve.
CHAPTER 4
PERFORMANCE IMPLICATIONS OF AGING VMS IN THE GPAS

4.1 Overview

As briefly discussed in §3.3.2, some jobs that run on VMs exhibit tail \textit{processing rate}. Since the workloads in the GPAS are I/O intensive, memory consuming, and most importantly long running (the longest job takes 39 days), this combined nature makes the issue hard to troubleshoot. This chapter expands the discussions and shows the steps we take to pinpoint the root cause and evaluate fixes to this issue.

4.2 Motivation

It brings to attention that some jobs exhibit tail performance.

![Figure 4.1: CDF of processing rate](image)

Figure 4.1 shows the CDFs of \textit{processing rate} of jobs on VMs and bare metal nodes. There are 796 and 247 jobs on VMs and bare metal, respectively, and all the jobs run the same type of pipeline ( “variant-filtration.pindel”), hence all the jobs should observe similar performance. In Figure 4.1 a, the nearly vertical solid line represents \textit{stable processing rate} of jobs on bare metal nodes. However, the dashed line shows that \textit{processing rate} on VMs is generally worse than \textit{processing rate} on bare metal nodes. While this is expected, the issue
lies in the tail performance especially at high percentiles. According to the zoomed graph in Figure 4.1b, starting from the 80th percentile, processing rate on VMs is 3 times worse than bare metal nodes, and in higher percentiles, the performance gets worse by an even larger magnitude.

We also would like to note again that the jobs in Figure 4.1 are jobs that come from one type of job pipeline ("variant-filtration.pindel") that generally only spends not more than 40 seconds/GB. However, there are other more CPU/memory-intensive pipelines that only process data 1000 ∼ 3000 seconds for each GB. The problem raised in this paper becomes worse for these type of pipelines.

To show that the behavior in Figure 4.1 is not because of degraded machines or hardware, we take several six VMs that run on six different machines and show the statistics of the job performance on these 6 VMs in Figure 4.2 in a boxplot. In every VM (e.g., VM1 on machine1), users can run the same job pipeline repeatedly. As shown in the figure, the processing rate in a single VM varies (even though the same pipeline on the same machine) and the performance across the VMs also varies (even though the VMs have the same configuration).

![Boxplot of performance on six VMs with the same configuration](image)

Figure 4.2: Processing rate variance on VMs
4.3 Investigation on the Tail Performance and Aging VMs

In order to understand the root cause to the issue, we conducted a set of experiments that included micro-benchmarks and real workloads. All experiments were conducted on a pair of VMs with the same virtual hardware setup and the same software setup, except that one of the VMs was cold-restarted before the experiment. We refer this cold-restarted VM as Fresh VM. The other VMs (Aged VM) had been running for a few days and were selected for close monitoring after slow jobs were observed.

Each VM was the only tenant on its host and configured to have 40 vCPUs, 226GB RAM and 2.5 TB storage. The hosts were equipped with two 2.20GHz 12-core 24-thread Intel Processors Xeon E5-2650 v4, 504GB RAM and 7.3 TB SSD RAID-5 storage. Linux-4.4 kernel, librunt 1.3.1 and OpenStack Nova 13.1.4 were installed for virtual machine support.

4.3.1 Application-Level Measurement

In GPAS, we noticed that jobs started running slower on VMs that have been running for several days. We ran a simple application to reproduce this. To simplify the experiment, we broke down a widely used pipeline in GPAS (Somatic Variant Calling) and only selected one of the tasks, a Java application called VarScan2 [61]. Job pipelines in GPAS are spawned as multiple processes, hence VarScan2 can run as multiple processes.

Since the input data for all the tests are the same, here we do not use processing rate, but instead execution time as the performance metric. A higher execution time implies worse performance (the same as in processing rate).

We define an “n-process VarScan2 task” as a task that uses n VarScan2 processes to process the input data. We define a “test” as an experiment that concurrently runs 5 tasks of 8-process VarScan2 on a single VM. The input data is replicated by 5 times and each task performs the same computation on the same content but distinct replicas of the data. After a test completes, we measure the average execution time of the 5 tasks in the test and record
that, and then repeat the test until 5 days have elapsed.

![Graph showing execution time vs VM uptime](image)

Figure 4.3: VarScan2 experiment

Figure 4.3 shows the average execution time of the tasks across several days. Every y point represents a test; the y value of a point shows the average execution time of the 5 concurrent tasks in the test. We can observe that between day 0 and 1.6 (marked by line B), the execution time of the test is relatively fast with low variance. However, after approximately 1.6 days, the performance starts to degrade. We observe that VarScan2 tasks perform normally on a Fresh VM, but perform much worse on an Aged VM.

### 4.3.2 Kernel-Level Measurement

To understand the slow performance in an Aged VM, we conducted further experiments including micro-benchmarks and in-kernel measurements in the Aged VM.

We use sysbench [62], a configurable multi-threaded benchmark tool that provides a variety of tests for benchmarking CPUs, multi threading, memory operation, and file I/O performance. We performed many varieties of experiments (not shown here for space) and found that most benchmarking results do not reveal much difference between an Aged VM and a Fresh VM (not shown), except for file I/O performance. Not only does the I/O throughput show different results; but, more interestingly, the monitored CPU utilization on Aged
VM and Fresh VM are quite different when compared to that of the host OS.

A common way to monitor CPU utilization is calculating the difference of accumulated values in the pseudo file system (/proc/stat). Simply speaking, the values represent how many time slices have been used for each type (user, system, etc) and for each CPU. There are many tools that profile CPU utilization including top and scollector. We use the latter as it collects and saves raw data, and we can use other tools to calculate and visualize statistics in desired ways.

![Figure 4.4: CPU utilization of sysbench](image)

Figure 4.4 shows the comparison of CPU utilization collected from the host OS versus inside an Aged VM. In the VM, we ran a sysbench file I/O test (on an SSD). It is worth noting that there are two lines in the figure (UtilRaw and UtilDrv) representing two ways we calculate CPU utilization. UtilRaw is the raw CPU utilization number (in %) that scollector outputs in every second. In addition to the raw CPU utilization, scollector also outputs more detailed information, including cpuUserSlices, cpuSysSlices, and cpuIdleSlices. We define UtilDrv (for derived) by summing the user and system time slices and dividing by all slices ((cpuUserSlices+cpuSysSlices)/allSlices). Again the difference is that UtilDrv simply sums the cpuUserSlices and cpuSysSlices without considering the idleSlices. The figure also shows another line sys, which represents cpuSysSlices divided by all the slices (in %).

We make the following important observations from the figure.
1. System (sys) CPU utilization is high on the host and equal to the overall CPU utilization. A similar observation can be found in the Aged VM. We consider this abnormal because the workload is I/O bound.

2. At the peak utilization, the CPU utilization observed on the host is 82%, while it is only 27% on the Aged VM. Note again that there is only one VM on the host and no other heavy workloads running on the host. This implies that the hypervisor works intensively, a hidden CPU overhead.

3. On the VM, there is a gap between the two ways we measure CPU utilization (the gap between UtilDrv and UtilRaw). Normally, these two lines should overlap as in the host-level measurement (left graph). What happens here is that scollector assumes that when a process gets a time slice (10ms), it always gets the entire time slice. However, with our method, UtilDrv, it shows there is a “loss of time” due to the hidden overhead in the hypervisor. CPU time that is supposed to be used for tasks in the VM was used for other system tasks in the host.

4. On a separate measurement on a Fresh VM (not shown for space), we found no such high system CPU usage nor a gap between the two lines.

4.3.3 Memory Fragmentation

A major problem of aging resources is fragmentation. We started suspecting there was a memory fragmentation problem where sequential guest pages were not mapped sequentially on the physical pages. To get more evidence, we ran concurrent processes and analyzed read operations. Roughly speaking, a file read operation goes through five steps: 1) check the page cache to see whether the data is already in memory; 2) conduct a synchronized read request at the file system level; 3) send out an asynchronous read request at the block level; 4) wait for completion; and, 5) copy the data from the kernel to the user space. Step (5)
represents the most memory-intensive step among all the steps.

We conducted an experiment that records the time spent in each of the steps. We set up four tests. The first test runs a single 1-process VarScan2 task, and the other four tests run 1, 4, and 5 (concurrent) 8-process tasks, respectively.

<table>
<thead>
<tr>
<th></th>
<th>1 × 1</th>
<th>1 × 8</th>
<th>4 × 8</th>
<th>5 × 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steps #1-4</td>
<td>&lt; 1</td>
<td>&lt; 2</td>
<td>&lt; 12</td>
<td>&lt; 20</td>
</tr>
<tr>
<td>Step #5</td>
<td>5</td>
<td>92</td>
<td>290</td>
<td>512</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
<td>94</td>
<td>306</td>
<td>552</td>
</tr>
</tbody>
</table>

Table 4.1: Read latency break-down

Table 4.1 shows the average time every process spends on each I/O step. Note that most of the time is spent in Step #5 (memory copying). With just 1 process (1x1), a process only takes 5 seconds in total for memory copying. With 40 processes (5x8), every process now takes 512 seconds in step #5, which is roughly a 100x slowdown. We note that there is other contention in the SSD (as can be seen in steps #1-4); but, even with 40 processes, the I/O waiting time is not as severe as the slowdown from memory-copying. We also note that we have a 48-core machine, hence CPU contention should be almost negligible with 40 processes.

4.3.4 The Root Cause: EPT Violation

As our experiments all point to memory management overhead, it is time for us to quantify the root cause. After considerable investigation, we found that the root cause resides in the extended page table (EPT), which is a technology invented to increase virtual memory performance for VMs. The use of EPT by the hypervisor is designed to be transparent to VM users. To our knowledge, EPT is used by only certain hypervisor implementations, such as Linux KVM. In a nutshell, EPT serves as a page table that stores the mapping between the VM memory address and the host physical memory address. Modern CPUs use the translation look-aside buffer (TLB) to store a small subset of the EPT entries. Whenever
there is a TLB miss, an *EPT violation* occurs, which causes the hypervisor to interrupt the VM to handle the violation.

Figure 4.5 shows an experiment with 5 jobs using 30 cores running repeatedly on a fresh state VM for ten days. Figure 4.5a shows the average of number EPT violations (in millions) observed in every 5 minutes. The figure clearly shows that the longer the VM has been running, the higher the number of EPT violations. Figure 4.5b confirms the correlation between the number of EPT violations and the job execution time.

![Figure 4.5: EPT violation](image)

**In conclusion, VM aging leads to more frequent EPT violations causing the hypervisor to interrupt the VM more frequently.** In the next two sections, we describe how to monitor and mitigate the problem.

## 4.4 Monitoring

We suggest two methods to detect VM aging: monitoring EPT violation (in the host) or CPU utilization gap (in the VM).

### 4.4.1 Host-Level EPT Violation Monitoring

One direct way to measure the problem is to count the number of EPT violations observed in the hypervisor, however the result is relative—how do we know whether the number represents a higher than normal number of EPT violations. We found another more con-
crete metric to measure this problem: *address distance of subsequent EPT violations* (which basically attempts to measure the level of memory fragmentation). For example, if two subsequent violations at time $T$ and $T+t$ are about translation misses of guest pages #100 and #2000, respectively, then the distance recorded is $1900 \times 4\text{KB}$. Essentially, we argue that when the addresses of subsequent EPT violations are farther apart, the memory tends to be more fragmented and more EPT violations will occur, causing more time to be spent managing EPT violations.

![CDF for Subsequent EPT Violation Address Distance](image)

**Figure 4.6: EPT violation monitoring**

We return to the experiment in Section 4.3.4 and this time plot the distribution of address distances of subsequent EPT violations. Figure 4.6 shows different distributions categorized based on the age of the VM age. Notice that older VMs have distinctly larger address distances. For example, in a 10-day old VM, we can see a distance of at least 50GB in roughly 40% of the time ($x=50\text{GB}, y=0.6$).

This monitoring method requires access to the host. It can provide performance alerts in advance. For example, the distance distribution in a 3-day old VM can be clearly distinguished from a fresh VM. Here, the resulting job execution time has been increased by 27% (which was hard to observed in the middle of the job). Thus, this kind of monitoring
allows us to predict job performance degradation even before the job ends. In terms of performance overhead, sampling violations in an online manner may bring an impact to system performance. However, our experiment where the trace is enabled for five minutes every ten minutes does not show a negative impact on performance.

4.4.2 VM-Level /proc/stat Monitoring

While the above method requires host-level access, we now present another method that can be done at the VM level. As explained before, the Linux kernel implements the proc pseudo-filesystem which provides an interface to kernel data structures. The /proc/stat file provides time-slice statistics across user, system, and idle processes from the time the system boots up. In our deployment, the time slice is set to 10ms.

Inspired by the “gap” shown in Figure 4.4 earlier, it is possible for users to monitor the gap at the VM level. The gap is caused by a phenomenon where a time slice in the VM is actually (and significantly) less than 10ms because the hypervisor is handling EPT violations. Thus, we can introduce a simple metric, *vCPU Efficiency*, which is the sum of all the time-slice values in /proc/stat (user, sys, and idle values) divided by the real time slices that have elapsed (since the last time we read /proc/stat). The metric should be near 100%, but when EPT violation is high, it is expected that the efficiency will be much lower than 100%. Figure 4.7 shows the simple shell script to calculate *vCPU Efficiency*.

```bash
stat_period=300
cpunum=`grep -c ^processor /proc/cpuinfo`
read cpu user nice system idle ... < /proc/stat
total_HZ_prev=$((user+system+nice+softirq+steal+...))
sleep $SLEEP
read cpu user nice system idle ... < /proc/stat
total_HZ_cur=$((user+system+nice+softirq+steal+...))
ElapsedHZ=$((total_HZ_cur-total_HZ_prev))
vCPU_Efficiency=$((ElapsedHZ/cpunum/stat_period));
```

Figure 4.7: Shell Script to Calculate *vCPU Efficiency*
Table 4.2: vCPU Efficiency

<table>
<thead>
<tr>
<th></th>
<th>vCPU Efficiency (%)</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fresh VM Heavy</td>
<td>99</td>
<td>16.0 hrs</td>
</tr>
<tr>
<td>Aged VM Heavy</td>
<td>83</td>
<td>39.0 hrs</td>
</tr>
<tr>
<td>Fresh VM Light</td>
<td>99</td>
<td>5.4 hrs</td>
</tr>
<tr>
<td>Aged VM Light</td>
<td>99</td>
<td>5.6 hrs</td>
</tr>
</tbody>
</table>

Table 4.2 shows vCPU Efficiency and the job execution time for the same experiments we ran before. Here, we select one “heavy” and one “light” application, where the heavy application uses all 6 available cores per job and the light application uses 1 core per job. We can see that for heavy workloads, there is a large difference in vCPU Efficiency and execution time. For example, when vCPU Efficiency drops from 99% to 83%, the resulting job execution time increases from 16 to 39 hours.

The disadvantage of this method is that we cannot detect VM aging unless we run a heavy application that can be impacted by the aging. Table 4.2 shows that for a light application, there is no visible difference in the vCPU Efficiency and execution time, even though the VM is already degraded at the time of running. We also want to emphasize that VM aging cannot be crudely defined by the number of days a VM has been up running. In our deployment, VMs age fast (after 3-6 days) because we ran complex bioinformatics pipelines.

4.5 Mitigations

This section describes several ways that we tried to address the problem. It is up to the system requirement and administrators to decide which mitigation technique suits them the best. Table 4.3 summarizes the pros and cons of the mitigation techniques we discuss below.
<table>
<thead>
<tr>
<th>Method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Huge Pages §4.5.1</td>
<td>Performance is close to bare metal nodes.</td>
<td>Huge page VMs are more complicated to configure, and less flexible.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Benefits of huge pages could be offset with even larger memory size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and memory usage.</td>
</tr>
<tr>
<td>Restarting VM §4.5.2</td>
<td>Performance is best after restarting.</td>
<td>The system must support job checkpointing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Longer down time</td>
</tr>
<tr>
<td>Defragmenting Memory §4.5.3</td>
<td>Performance is improved, short down time.</td>
<td>Cannot provide the best performance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Improvement is only temporary</td>
</tr>
<tr>
<td>Running on bare metal §4.5.4</td>
<td>Performance is best and sustainable.</td>
<td>Difficult to manage and maintain.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Security concerns.</td>
</tr>
<tr>
<td>Using public clouds §4.5.5</td>
<td>Based on one-week experiments, the performance is best and sustainable. Easy to manage</td>
<td>Security concerns.</td>
</tr>
</tbody>
</table>

Table 4.3: Pros and cons of five mitigation methods

### 4.5.1 Using Huge Pages

Just like a standard page table, EPT size increases as memory grows larger, hence higher probability of misses when the unit of page is 4KB. Increasing the page size (i.e., using 1MB “huge” pages) can effectively shrink the size of the EPT table. However, in GPAS this is not an easy option to adopt. Conceptually, huge page VMs are less flexible. Configuring and debugging huge page VMs in a production environment takes time. For academic platforms, administrator time is more limited compared to those in large industries. Hence, it is not easy to reconfigure and troubleshoot all the machines with a huge page configuration. In
addition, a huge page size that is “huge” enough for now cannot be a permanent solution with the increase of memory and application memory usage in the future [18]. Another way is to keep the 4KB page unit but allocate smaller VMs to reduce memory fragmentation. However, in GPAS, resource requirements differ across jobs; many of them require large VM memory.

### 4.5.2 Restarting VMs to Avoid Performance Degradation

Another method is to restart the VMs occasionally to “reset” the memory fragmentation. According to Figure 4.5, it is possible to detect EPT violations early before performance degrades significantly. With such a detection, we can decide when is a proper time to restart. We conducted a simple experiment with the same jobs as in Figure 4.5. Table 4.4 shows that restarting increase the number of jobs finished per day. Here, “proactive restart” implies restarting the VM after every job finishes (i.e., do not reuse the VM across jobs) and “slowdown-triggered restart” implies restarting when the monitored νCPU Efficiency drops below 90%. The throughput numbers in Table 4.4 might also suggest that restarting in the middle of a long job might improve its execution time. However, this requires checkpointing the job progress, which a feature that is not supported in GPAS given the complexity of bioinformatics jobs.

<table>
<thead>
<tr>
<th></th>
<th>Jobs per day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proactive restart</td>
<td>1.92</td>
</tr>
<tr>
<td>Slowdown-triggered restart</td>
<td>1.69</td>
</tr>
<tr>
<td>No restart</td>
<td>1.23</td>
</tr>
</tbody>
</table>

Table 4.4: Rebooting VM mitigation

### 4.5.3 Defragmenting Memory

The burst of EPT violations in aging VMs is essentially caused by the loss of data locality of the VM memory on the host physical memory. We can use the built-in memory de-
fragmentation tool in Linux to reorganize the memory layout. A simple experiment with VarScan2 workloads shows that defragmentation indeed helps decrease EPT violations. As shown in Table 4.5, the test runs 21% faster after defragmentation when the VM has been heavily used for 7 days. The number EPT violations during the test is decreased by 58%. However, comparing to performance of the fresh state, this method is still 44% slower and EPT violations are nearly 100 times more. In other words, memory defragmentation can be a temporary method to improve performance by a small margin, but restarting VMs leads to a better outcome.

<table>
<thead>
<tr>
<th>VM age</th>
<th>Exec. Time (s)</th>
<th>EPT violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-day</td>
<td>587</td>
<td>630636</td>
</tr>
<tr>
<td>7-day</td>
<td>1073</td>
<td>140677142</td>
</tr>
<tr>
<td>7-day, defragmented</td>
<td>847</td>
<td>58607955</td>
</tr>
</tbody>
</table>

Table 4.5: Memory defragmentation

4.5.4 Running on Bare Metal

In GPAS, the most viable alternative is to run jobs directly on bare metal nodes without VMs. The caveat is that not all research projects (jobs) can run in this mode; some research projects require strict security to be bundled in a secure VM. Thus, we make GPAS supports hybrid bare-metal and VM deployments, especially for jobs that prefer performance to flexibility and security. Recently, some jobs have been running on bare metal nodes directly. The statistics presented below are from job pipelines that have more than one hundred jobs on bare metal nodes.

In Figure 4.8, a Gaussian kernel density estimation is calculated for jobs of four particular pipelines on bare metal nodes and VMs. As shown, bare-metal jobs exhibit much less variance than VM jobs (the bell shapes of bare-metal jobs are more localized across the x-axis). Besides, bare-metal jobs also show a more superior processing rate than that of VMs. In addition, the 95th percentile processing rate is marked in the graphs; it is clear that the
processing rate tail of bare-metal jobs is relatively short.

![Gaussian density estimation](image)

Figure 4.8: Processing rate Gaussian density estimation

Among all the fifteen pipelines of which each has over one hundred bare-metal jobs, job performance has been improved a lot when compared to VMs. The 95th percentile processing rate is improved by 22–95%, the average 18–87%, and the variance by 19–100% for all the pipelines. The processing rate improvement (in %) at different percentiles is shown in Table 4.6. In summary, given these benefits, running the GDC platform in a hybrid mode (bare metal and VM supports) becomes a viable option, albeit more management complexity and overhead.

<table>
<thead>
<tr>
<th>25th</th>
<th>median</th>
<th>75th</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 ~ 78</td>
<td>18 ~ 79</td>
<td>19 ~ 81</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>95th</th>
<th>97th</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 ~ 95</td>
<td>20 ~ 97</td>
<td>28 ~ 100</td>
</tr>
</tbody>
</table>

Table 4.6: Running on bare metal
4.5.5 Using Public Clouds

The problem we unveil in this paper pertains to OpenStack/KVM deployment. Different virtualization stacks likely to employ different approaches. Many vendors [3, 2] have stated in their documents that configuring huge page memory is beneficial to VMs with a large amount of memory, but there is no concrete recommendation on how large the memory is when huge page must be configured for the VM, and it is possible for most people to ignore the necessity. On the other hand, hypervisors such as Xen use another approach to implement virtual machine memory management which directly maps guest virtual address to host physical address (called direct paging), thus they do not incur any overhead for resolving the mapping from guest to host as KVM does.

To find out if this problem appears in other virtualization stacks, we tried running jobs on commercial cloud providers. The caveat of using public clouds is of privacy and security reasons. Indeed, because of protected data policy, we cannot run the same VarScan2 experiments on public cloud clouds. Thus, to evaluate public clouds, specifically Amazon Web Services and Google Cloud Platform, below we use open access data from GDC and DNA alignment workloads. As a baseline, for one-week experiment, the minimum execution time on our VM is 10 hours and the maximum is 15.3 hours. In an aging, 4-day old VM, the experiment begins to take more than 13 hours.

<table>
<thead>
<tr>
<th>Tests</th>
<th>Min (hrs)</th>
<th>Max (hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>One on-prem VM</td>
<td>36</td>
<td>10</td>
</tr>
<tr>
<td>Amazon Web Services</td>
<td>69</td>
<td>3.6</td>
</tr>
<tr>
<td>Google Cloud Platform</td>
<td>98</td>
<td>5.7</td>
</tr>
</tbody>
</table>

Table 4.7: DNA Alignment Pipeline Experiments

Amazon Web Services develop their own hypervisor based on Xen. We rent a dedicated host (z1d) and allocate a large memory VM (12xlarge). The one-week experiment does not show any performance degradation. The minimum execution time is 3.6 hours and the maximum is 3.9 hours. As mentioned before, we suspect that the Xen direct-paging approach
successfully avoids the address translation overhead, albeit less flexibility for memory sharing.

Google Cloud Platform builds their hypervisor based on KVM [6]. We rent a 96-core sole-tenant host to avoid sharing with other users and allocate a 90-core 576GB VM on the host. We repeat the same experiment while scaling the maximum number of jobs to 14. The experiment results also does not show any degradation either. The minimum and maximum execution times are 5.7 and 6.0 hours, respectively. Although they use KVM, the same problem might not appear due to one of the following reasons: they use huge pages, they use software-based memory management instead of EPT, the CPUs they use have larger TLBs.

In summary, public cloud platforms are highly prevalent nowadays, but for many bioinformatics tasks with stringent security and privacy measures, using public clouds, although fast, is not an option. For all of the reasons stated in this section, our GDC platform resorts to hybrid VM/bare-metal supports and occasional VM restarts.

4.6 Summary

To the best of our knowledge, we are the first that conduct a prolonged performance evaluation of virtualization stack, specially the OpenStack/KVM stack for bioinformatics platforms. We have shown that bioinformatics workloads are uniquely long running and memory intensive to the point that they cause virtual memory to be highly fragmented after days of operation, causing a high increase of EPT violations and interrupts to the host, a phenomenon that appears in what we call “aging VMs”. We have presented a detailed diagnosis as well as two monitoring techniques (host and VM levels) and a range of mitigation techniques (including their pros and cons) that can be adopted by the GPAS.
CHAPTER 5
PIPELINE JOB SCHEDULING IN THE GPAS

5.1 Overview

Due to the complicated composition of bioinformatics pipeline, the GPAS uses a very simple job scheduling model that simply parallelizes jobs on a VM node, which is also referred as “embarrassingly parallel” in HPC context.

This simple scheduling model have shown drawbacks in the GPAS, causing job failures extremely costly, computing resource under-utilization and contention. These drawbacks make it a necessity to adopt a task-based scheduling model.

This chapter presents the evidences of the drawbacks in the simple scheduling model, and discusses the benefits of a task-based scheduling model, and the challenges to implement a good task-based scheduler.

5.2 Current Job Scheduling in the GPAS

As shown in Figure 2.10, the core parts in the GPAS is a job scheduling service and a VM pool managed by SLURM. SLURM is a grid-computing cluster management software featured with multiple job scheduling algorithms. Since the CWLtool does not provide APIs to connect with computing environment, the external job scheduling service plays the role to schedule jobs, and SLURM is only responsible to distribute jobs to VMs according to jobs’ requirement on computing resources. Besides, the GPAS relies on a S3-compatible storage service to acquire data through a flat-file genomics database service, and jobs’ status information is kept track of in a local relational database.

The design of job scheduling in the GPAS differs from NextFlow and cromwell, mainly because the version of CWLtool in GPAS does not support parallelized task execution for a job. In the GPAS, information of jobs are put in the graph-oriented database at first,
then the job scheduling service will pick up job information and assemble job configuration in json format and submit to SLURM in the end in a first-come-first-serve style. Thus the GPAS has the ability to schedule a lot of jobs on a large cluster. However, NextFlow and cromwell connect to a cloud computing cluster directly without any external components. Only tasks inside one job can be executed at a time on the specific cloud cluster.

The ultimate goal of the GPAS is to achieve highly coordinated jobs scheduling where tasks from different jobs can be executed on one node to maximize resource utilization. However, in the current stage, a certain number of jobs are just simply run in parallel on a node, which is more similar to “embarrassingly parallel” as introduced in §2.4.2. Besides, the GPAS always allocate the largest possible computing resource that a job may need to the job, however, the allocated resource may not always be fully utilized during the execution of the job.

5.3 Drawbacks of the Existing Job Scheduling Model

5.3.1 Extra Cost Caused by Job Failures

Other than the “embarrassingly parallel” computing fashion, the GPAS does not use job progress caching mechanism or checkpointing for jobs. And there are mainly three reasons:

1. The GPAS expects the error rate for jobs to be low, so retrying a job from start doesn’t hurt too much.

2. There can be various types of errors or reasons that terminate a job, intermediate results for a job may not be useful.

3. Sometimes a job needs to be retried on another node, so the intermediate results are not easily accessible.

Because of this design choice, it may take some extra time for a failed job to be rescheduled
and retried until the job finally completes successfully. What’s worse, it is possible for the job to fail multiple times and to be retried multiple times.

<table>
<thead>
<tr>
<th></th>
<th>VM Min</th>
<th>VM Max</th>
<th>VM Avg</th>
<th>Bare metal Min</th>
<th>Bare metal Max</th>
<th>Bare metal Avg</th>
<th>Overall Min</th>
<th>Overall Max</th>
<th>Overall Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution minutes</td>
<td>283</td>
<td>24617</td>
<td>2098</td>
<td>1227</td>
<td>8573</td>
<td>3673</td>
<td>283</td>
<td>24617</td>
<td>2134</td>
</tr>
<tr>
<td>Attempts</td>
<td>1</td>
<td>4</td>
<td>1.28</td>
<td>1</td>
<td>2</td>
<td>1.02</td>
<td>1</td>
<td>4</td>
<td>1.28</td>
</tr>
<tr>
<td>Error rate</td>
<td>926/4171 (22.2%)</td>
<td>2/95 (2%)</td>
<td>928/4266(21.75%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Basic Statistics for Jobs of Somatic Variant Calling Pipeline

Table 5.1 shows summary statistics for Somatic Variant Calling pipeline jobs. Some jobs were run on bare metal nodes to provide a baseline for the jobs running on VMs. All the jobs reflected in the table have succeeded in the end, though there might be some failed attempts. In the leftmost column, “Execution minutes” are measured during the last successful execution for a job, “Attempts” are measured by counting all attempts including the final successful attempts, and “Error rate” is calculated by the failed attempts divided by all the attempts. It’s worth noting that this way of calculating error rate differs from the way to calculate success rate in §3.4.3 in Figure 3.12. All the jobs in this table are regarded as successful jobs in Figure 3.12 because the success rate in §3.4.3 only cares about the final status of the jobs, instead of the status of each job attempt.

The statistics show that 22.2% Somatic Variant Calling pipeline jobs in the GPAS have been attempted more than once, and lengthy time has been spent in retrying jobs.

We define “Job delayed time” to the period between the time when a job was first attempted and the time when the job was successfully finished in the same first attempt or in another later attempt. “Job delayed time” can vary a lot, because some jobs may fail due to some random errors and simply by another attempt it can be executed successfully. However, other jobs may need manual investigation to fix error which brings in nondeterministic overheads.

Figure 5.1 shows the distribution of “Job delayed time” in hours for jobs on VMs and
bare metal nodes, and the majority of the jobs have been delayed by around 1 day until they can finally start to run successfully. What’s worse, there are considerable number of bad cases that are delayed by 100 hours on VMs, which significantly affects the efficiency of the GPAS. Again, the difference between VMs and bare metal nodes mainly come from the different error types from VMs and bare metal nodes.

The long delayed time exposes a critical issue in the GPAS because of the job level scheduling and neglect of job progress caching. More importantly, since a pipeline job generally takes a long time to complete, more efficient pipeline debugging methods are necessary for jobs to quickly recover from errors.

5.3.2 Resource Under-utilization and Contention

Apart from reducing the overhead of error recovery, it is crucial to improve the utilization of computing resources so the system can get high job throughput.

Based on the scheduling design of the GPAS, multiple jobs may be scheduled to run on
one VM host, and tasks within a job are executed in serialized order. It is possible that tasks would ask for different amount of resources. The current practice in the GPAS would always grant the maximum resource for a job that all tasks in this job may request. However, computing resource may not always be fully utilized for the job.

Among the computing resources that can be allocated to a job, memory and storage can only be allocated at a large volume because the real time memory and storage usage can be constantly changing. CPU resource, on the other hand, is more rigid because for a multi-threading/multi-processing task, it is more likely that it will use all the allocated CPUs during the whole execution of the task. However, the challenge of allocating CPUs efficiently in the GPAS is that not all the tasks in a job are multi-threading/multi-processing. This leads to varied average CPU utilization for different pipelines.

Another issue which probably slows down jobs is the I/O contention. Apparently, adoption of SSDs can avoid I/O contention to some degree, however, a small number of VMs in the GPAS uses spinning disks and disks are still widely used to support cheaper computation.

Figure 5.2 shows the real time CPU and I/O utilization for an experiment that runs five somatic variant calling jobs on a VM that has 48 cores, 226 GB RAM, and 1 TB hard disk storage. The experiment starts these five jobs at the same time in the VM, and 8 CPUs are allocated for each job. The whole experiment runs for around 2 days, and Figure 5.2 captures I/O and CPU utilization for the second day.

The top graph shows the CPU under-utilization for the five jobs, and system-wide CPU utilization never reaches the maximum utilization, 83.3%. For most of the time, the jobs wait for I/O to complete. Between 3:27PM and 5:31PM, there is a window with low I/O activity, but the CPU activity is also relatively low. This is because the jobs reach a task where not all 8 CPUs are used.

The bottom graph shows the I/O bandwidth. For most of the time, I/O utilization is always full, and it is mixed with read and write. However, the bandwidth of read and write
Figure 5.2: I/O and CPU Utilization of 5 Somatic Variant Calling Jobs Running in VM is much lower than maximum bandwidth of the hard drive, that is because the spinning disk cannot handle the read and write contentions from all five jobs.

In addition, graphs like those in Figure 5.2 can only be explained together with job logs which contains the timestamps for all the tasks. However, there are too many tasks in a job and too many different jobs in the cluster, making it difficult to identify the exact tasks that result in resource under-utilization and contention on a worker node.
5.4 Task-based Scheduling Model

5.4.1 Proposed Task-based Scheduling Model

As introduced in §2.2.3, some pipeline execution engines support parallelized execution for tasks, such as CWLtool, cromwell and NextFlow. However, the implementation in those execution engines only considers the execution for a single job. In §2.2.3 Figure 2.6 has already illustrated the situation where parallelized tasks may require more CPU resources than the worker has.

To achieve a better solution, we need to implement a global control to at least coordinate all tasks in scheduled jobs on a VM host, so that computing resource for individual job can be guaranteed. The key difference between the proposed scheduling and current scheduling is that a whole pipeline job is broken down into tasks. And scheduling is conducted in a finer granularity on the task level.

![Figure 5.3: Task-based Scheduling Model](image)

Figure 5.3 illustrates the proposed scheduling model that is able to schedule jobs based on tasks, and guarantees the CPU limits for individual task, as well as respects the CPU limit of the VM worker. As the figure shows, there won’t be more than 2 threads used for Job 1, even if the second and the third task could be executed in parallel if there is no CPU limitation. Secondly, during the execution of the second task of Job 1, tasks from other jobs can be executed to saturate available CPU resource of the VM.
5.4.2 Theoretical Assessment Through Simulations

<table>
<thead>
<tr>
<th>Simulated CPU Utilization</th>
<th>Old 5 Jobs</th>
<th>Task-based 5 Jobs</th>
<th>15 Jobs</th>
<th>20 Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (Minutes) Reduction (%)</td>
<td>3904</td>
<td>3852 (-1.3%)</td>
<td>10174 (-13.1%)</td>
<td>13260 (-15.1%)</td>
</tr>
</tbody>
</table>

Table 5.2: Simulated Execution for Somatic Variant Calling Workflow Jobs

<table>
<thead>
<tr>
<th>Simulated CPU Utilization</th>
<th>Old 5 Jobs</th>
<th>Task-based 5 Jobs</th>
<th>15 Jobs</th>
<th>20 Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (Minutes) Reduction (%)</td>
<td>4862</td>
<td>4839 (-0.4%)</td>
<td>13917 (-4.6%)</td>
<td>18252 (-6.1%)</td>
</tr>
</tbody>
</table>

Table 5.3: Simulated Execution for Bamfastq-align Workflow Jobs

Table 5.2 and Table 5.3 show the simulated execution time for two pipelines: Somatic Variant Calling and Bam FastQ Alignment. Estimated execution time for tasks are extracted from one job of that pipeline, and the simulation experiments simulate running 5 jobs in parallel, and 15, 20 jobs in a batch. Each job can use at most 8 threads, and at most 40 threads can be used at the same time by all the jobs. The percentage of time reductions use the execution time of original 5 jobs multiplies by 3 or 4, for 15 or 20 batch jobs, respectively. We can see that there are considerable improvement on CPU utilization and execution time when we batch more jobs.

We also notice that improvements shown in Table 5.3 are smaller than in Table 5.2, it is because the original CPU utilization is high in Bam FastQ Alignment pipeline. Thus, the improvement brought by the new scheduling depends on the pipeline design. It is reasonable that if the pipeline doesn’t have any single thread bottleneck as mentioned before, there won’t be any benefits to adopt task-based scheduling if only CPU utilization is the concern.

Now that the minimum scheduling units are tasks, it is possible to checkpoint task outputs that are original intermediate results in the current scheduling model. Thus, better
recoverability is provided.

5.4.3 Task Specific Optimizations

In addition, tasks, which are usually executing a single tool, are easier to classified by their workload characteristics. Based on tasks’ requirement on computing resources, more sophisticated scheduling algorithm is able to arrange tasks with different demands together and resource utilization can be maximized. Another prototype of this scheduling model implements scheduling optimization for Somatic Variant Calling Pipeline, where the execution of MuSE, MuTect2, SomaticSniper and VarScan2 is specially optimized.

To be precise, through benchmarking, we learned that SomaticSniper and VarScan2 are bounded equally by I/O and CPU when they are executed on disk-based storage, and MuTect2 and MuSE are simply CPU bound. As a result, we implemented a scheduling component that coordinates the execution of SomaticSniper and VarScan2 tasks by prefetching the needed data for each thread and then starting computation. Since the other two tasks are CPU bound, we just start them when there are CPU resources available.

Figure 5.4: Prefetching Data Before Computation Optimization for Tasks
Figure 5.4 shows the execution timeline for each task when we test five jobs each consuming eight threads on a 48-core VM with spinning disk as main storage. Comparing to the original scheduling, the execution time for the jobs is improved by 18.6%, and the improvement for the four major computation tasks are as much as around 50%.

<table>
<thead>
<tr>
<th>Experiment Scenario</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o prefetching, Fresh VM</td>
<td>71 min</td>
</tr>
<tr>
<td>w/ prefetching, Fresh VM</td>
<td>66 min</td>
</tr>
<tr>
<td>w/o prefetching, Aged VM</td>
<td>123 min</td>
</tr>
<tr>
<td>w/ prefetching, Aged VM</td>
<td>87 min</td>
</tr>
</tbody>
</table>

Table 5.4: Prefetching Optimization for Tasks When the VM is Fresh or Aged

Moreover, this optimization can also improve job performance when the VM degrades because of the issue discussed in Chapter 4. In an experiment that runs the same VarScan2 tasks as in the previous experiment. Table 5.4 shows the execution time for the experiment. When the VM is in fresh state, it takes 71 minutes to finish the tasks when data is not prefetched, and 66 minutes (including the time spent in prefetching data) with the prefetching optimization. Although the five minutes improvement is not a big deal, when the VM is in a slow state, it proves that prefetching optimization is extremely helpful. Running the tasks without prefetching when the VM is slow takes 123 minutes (73% slowdown), but with prefetching, it takes only 87 minutes (22%) slowdown. This is because the design of prefetching limits the number of concurrent I/Os. Thus, even though the VM is degrading performance of I/O, I/O bandwidth can remain high for single reading process.

5.4.4 Challenges of Building Task-based Scheduling for Pipelines

To implement an efficient task-based scheduling, there are a number of challenges that we need to tackle.

1. **Type of tasks:** The rule of thumb is that the scheduler must be aware of the accurate type of the tasks. As discussed in this dissertation, VarScan2 and Somatic
Sniper tasks in the Somatic Variant Calling Pipeline are I/O and computation mixed workloads, and when they are running on spinning disks, it is best to separate reading from computation. Mutect2 and MuSE tasks are computing intensive workloads, and without data prefetching on spinning disk, the computation is not obviously interfered by reading data.

2. **Storage type**: Since the effect of prefetching is different depending on the storage type, the scheduler must also take the storage type of worker nodes into account, though it is shown in Table 5.4 that prefetching on VM with SSD storage may also help improve performance.

3. **VM degradation issue**: As extensively discussed in Chapter 4, the VM degradation issue seriously affects the performance of I/O on VMs with SSDs. And we also find that the task-based scheduling can potentially accelerate the VM degradation when we make too many I/O intensive workloads. In an experiment where the scheduler tries to schedule a batch of ten jobs, 20 parallel I/O intensive tasks are started and $vCPU$ Efficiency drops to 92% during the first hour since the VM is freshly started, which is not observed before then the jobs are running in “embarrassingly parallel” model.

4. **Balance between number of parallel tasks and other computing resources**: When more tasks are parallelized, more memory and storage space are certainly require to accommodate all the tasks. Especially for bioinformatics workloads which are common to consume large volume of data, it is easy to cause task failures because of insufficient memory and storage.

5. **Efficient task designs**: The last but not the least, the design of the tasks is important, especially for the purpose of accurate resource management. Table 5.5 shows the average I/O read bandwidth and CPU utilization for running 1, 8, 16, 32, 40 Somatic Sniper tasks in parallel with only one thread allowed for each task on a VM with 48
CPUs and SSD.

<table>
<thead>
<tr>
<th></th>
<th>Average CPU Utilization</th>
<th>Average Read Bandwidth (Maximum: 2GB/s for the SSD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Task</td>
<td>5 %</td>
<td>1.0 MB/s</td>
</tr>
<tr>
<td>8 Tasks</td>
<td>41 %</td>
<td>6.6 MB/s</td>
</tr>
<tr>
<td>16 Tasks</td>
<td>76 %</td>
<td>10.2 MB/s</td>
</tr>
<tr>
<td>32 Tasks</td>
<td>87 %</td>
<td>11.0 MB/s</td>
</tr>
<tr>
<td>40 Tasks</td>
<td>89 %</td>
<td>11.1 MB/s</td>
</tr>
</tbody>
</table>

Table 5.5: Somatic Sniper Multiple Task Experiment

We note that for one task, the average CPU utilization is already more than twice the theoretical value (2.1%), and it is in similar case for 8 parallel tasks and 16 parallel tasks. For 32 and 40 tasks, it seem the CPU utilization is saturated. This is because in the design of Somatic Sniper task, there is actually an extra thread running in parallel that is not controlled by task configuration. In fact, this task uses SAMtools in parallel to convert the format of Somatic Sniper task’s input in advance. Besides, the conversion is also computing intensive, and SAMtools becomes a bottleneck causing reading bandwidth to be far lower than the maximum bandwidth of the SSD.

5.5 Summary

This chapter discusses the drawbacks of the current simple job scheduling model including high cost for retrying the whole job when a job fails, and computing resource under-utilization and contention due to lack of finer-granularity scheduling.

A task-based scheduling model is proposed and evaluated through simulation to be useful in improving computing resource utilization and job performance. Moreover, the task-based scheduling model opens the opportunities for task specific optimizations. A simple optimization that prefetches data into memory before computation is proved to be beneficial for jobs that run on slow storage, or on a VM that has degraded performance.
At last, this chapter discusses five challenges that need tackling to implement a good task-based scheduler, including identifying the computing type of all the tasks, choosing different strategy for different storage media, being aware of VM degradation issue, the balance between parallel tasks and the increase consumption of computing resources due to parallelization, and efficient task designs.
CHAPTER 6
TOWARDS A VISION FOR THE GPAS WITH IMPROVED PERFORMANCE

6.1 Bioinformatics Research in the Cloud Computing Era

Majority of recent researches in bioinformatics cloud computing fall into two categories, highly parallelized applications and building a pipeline computing platform specifically for bioinformatics. While a few works proposed new pipeline execution engine [87, 36, 77], CWLtool, cromwell and Nextflow remain as the best choices for the GPAS because almost any application can be assembled into a pipeline.

6.1.1 Bioinformatics Applications

As Bioinformatics is an interdisciplinary subject, a large number of researchers in bioinformatics are not experts in software development. Most bioinformatics applications are designed to run locally until recently cloud computing starts to play an important role.

It is never an easy job to make bioinformatics applications to exploit the distributed computing resources. More research emerges to make new applications with new computing frameworks, or to transform a classic tool to use new frameworks. A typical example is the Basic Local Alignment Search Tool (BLAST)\(^1\), and it has been actively transformed into multiple versions: mpiBLAST [28], ScalaBLAST [93], GridBLAST [66], CloudBLAST [82]. mpiBLAST is actually a HPC version of BLAST where each computing node searches a portion of the database and MPIs are used to communicate between tasks. GridBLAST adopts the “embarrassingly parallel” computing model in grid computing. ScalaBLAST makes use of a distributed database as the “shared memory” and enables the possibility of running

\(^1\) https://blast.ncbi.nlm.nih.gov/Blast.cgi
BLAST on thousands of processors. CloudBLAST combines MapReduce and virtualization, and runs on two geographically separate clusters. Similarly, a large number of alignment or sequence mapping applications have been created: CloudAligner [91], CloudBurst [107], Crossbow [68], SparkBWA [7]. What is common for this kind of applications is that the operations inside of the application are actually independent with each other. Thus, transforming these applications to perform the operations on a distributed cluster is natural.

It is also pointed out that the applications implemented in MapReduce is actually within the Many-Task Computing paradigm, which is introduced in §2.4.1.

6.1.2 Bioinformatics Pipeline Platforms

A large number of Bioinformatics Pipeline Platforms researches stress on the ease that the platform brings to bioinformaticians to create and execute pipelines, and how they integrate into cloud computing environments. These include SciApps [117], BioWorks [44], BioWMS [17], Taverna [94], Workflow-based PSE [112], Discovery Net [105] and Galaxy [65, 78]. It is common that how pipeline jobs are scheduled is not included in the literatures, but some of them use existing middleware to control computing in the cloud, e.g. BioWMS uses Hermes [26], Workflow-based PSE and Galaxy use HTCondor.

Though not often talked about in the context of bioinformatics pipeline platforms, pipeline or workflow scheduling is a critical to good pipeline performance for the platform. Parallelism is explored for bioinformatics workflows [86, 34], and it is believed that the parallelism for general scientific workflows (see §2.4.4) is beneficial to bioinformatics pipelines as well. Pipelines eventually need to be broken into tasks where a task is usually a sole bioinformatics application. What is more commonly researched is that the mechanisms and policies to provision resource for bioinformatics applications and schedule applications in the cloud environment [108, 97, 31, 52, 29].
6.2 A Vision for GPAS: Hybrid of HTC and MTC

In scientific computing, it is important to accurately classify the computing paradigm for an application. However, through the literature review of bioinformatics research, we find that it is hard to simply classify bioinformatics pipeline platform as HTC or MTC. This is because the large variety of applications and the diversity of bioinformatics algorithms. But we must admit that the finer control over smaller tasks in MTC, though requiring more sophisticated management, can achieve better performance for the system.

Thus, there are a few research directions to improve the performance of GPAS as the following:

1. After adopting the task-based scheduling model, the GPAS should choose whether the deadline-constraint scheduling is a better suit or scheduling for a lower cost is better.

2. Currently, since the tasks are mixed up in pipeline jobs, there hasn’t been any proper prediction model for the execution time of tasks or jobs. With pipeline breaking into tasks, work should be done to create a proper prediction for the execution time of tasks.

3. How to deploy MTC tasks simultaneously with other non-MTC tasks. The GATK applications, some of which are implemented in MapReduce or Spark, are commonly used in pipelines in the GPAS. It may be possible to make some applications run with MapReduce on a subset of nodes in the cluster, but other applications run in “embarrassingly parallel” on other nodes.

4. Find more methods to convert non-MTC applications into MTC. An good example in the GPAS currently is that because VarScan2 and Somatic Sniper do not support multi-threading, input files have been split so that multiple processes can be spawned to compute in parallel.
5. Develop a good middleware which is able to schedule some applications in MTC and other applications in HTC.
CHAPTER 7

CONCLUSIONS

This dissertation presents an in-depth review on the bioinformatics pipeline platform, GPAS. Starting with the creation of an all-in-one database, this dissertation opens up opportunities of looking into job statistics of the GPAS.

A significant VM performance problem is revealed in the dissertation that a significant number of jobs in the GPAS, exhibited much slower performance compared to other jobs of similar types, up to 1,000% degradation. To the best of our knowledge, we are the first to conduct a prolonged experiment with bioinformatics workloads on the VM stack.

Moreover, this dissertation also reviews the job scheduling model that the GPAS provides and finds out that the current scheduling is inefficient because job failures bring high extra costs, parallel running jobs causes computing resource under-utilization and contention.

Lastly, combined with a thorough literature review on others’ experience of building a bioinformatics application/pipeline platform, this dissertation presents future research possibilities the GPAS to further improve the performance of pipelines.

In summary, the contributions of this dissertation are:

1. A statistics synthesizing service is created for the GPAS and an all-in-one statistics database is created. The production statistics collected from the GPAS in this database facilitates most of the topics in the dissertation and it is invaluable for future research.

2. This dissertation shows how real bioinformatics workloads can cause “aging VMs” after several days and performance degradation by up to 1,000%. Several possible mitigation scenarios are evaluated.

3. The job scheduling model in the GPAS is carefully examined and discussed, and a task-based scheduling model has been proposed with evaluations and discussed.
4. This dissertation presents a literature review on bioinformatics application/pipeline platform and gives a vision for the GPAS with further improved pipeline performance.
REFERENCES


Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI ’04.


[99] Binh Pham, Jan Vesely, Gabriel H Loh, and Abhishek Bhattacharjee. Using tlb speculation to overcome page splintering in virtual machines. 2015.


