PAGE: A Framework for Easy PArallelization of GEEnomic Applications

Mucahid Kutlu
Department of Computer Science and Engineering
Ohio State University
Columbus, OH, 43210
Email: kutlu@cse.ohio-state.edu

Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University
Columbus, OH, 43210
Email: agrawal@cse.ohio-state.edu

Abstract—With the availability of high-throughput and low-cost sequencing technologies, an increasing amount of genetic data is becoming available to researchers. There is clearly a potential for significant new scientific and medical advances by analysis of such data, however, it is imperative to exploit parallelism and achieve effective utilization of the computing resources to be able to handle massive datasets. Thus, frameworks that can help researchers develop parallel applications without dealing with low-level details of parallel coding are very important for advances in genetic research.

In this study, we develop a middleware, PAGE, which supports ‘mapreduce-like’ processing, but with significant differences from a system like Hadoop, to be useful and effective for parallelizing analysis of genomic data. Particularly, it can work with map functions written in any language, thus allowing utilization of existing serial tools (even those for which only an executable is available) as map functions. Thus, it can greatly simplify parallel application development for scenarios where complex data formats and/or nuanced serial algorithms are involved, as is often the case for genomic data. It allows parallelization by partitioning by-locus or partitioning by-chromosome, provides different scheduling schemes, and execution models, to match the nature of algorithms common in genetic research.

We have evaluated the middleware system using four popular genomic applications, including VarScan, Unified Genotyper, Realigner Target Creator, and Indel Realigner, and compared the achieved performance against with two popular frameworks (Hadoop and GATK). We show that our middleware outperforms GATK and Hadoop and it is able to achieve high parallel efficiency and scalability.

I. INTRODUCTION

Analysis of genetic data is becoming increasingly critical for medical research and even practice. Trends in sequencing technologies have drastically reduced the cost of, and increased the speed of, collecting gene sequences [1], [6]. This data is now being shared aggressively through various projects, and researchers at different institutions can download and analyze the data. An example of such efforts is the 1000 Human Genome Project1, which has already produced 200TB of genome data across 1700 samples, and made it available on the Amazon Cloud Storage2.

As a large number of researchers have access to the data, the focus is beginning to shift to analysis of this data. There exist many tools [15], [24], [30], [34] and libraries [4] to ease the implementation of new data analysis programs on genetic data. One of the complexities in this area is that data formats in which sequences are stored are very specialized. Existing tools, in most cases, alleviate the need for scientists understanding these formats and coding complex algorithms.

Consistent with the overall trend in data analytics, use of parallelism is inevitable in analysis of genomic data also. Lowered response time, including facilitating interactive analysis of large-scale data, can open up many novel opportunities for researchers. Again, just like all other fields, the difficulty of carrying-out parallel implementations, especially for domain experts, is a large obstacle to widespread use of parallelism for analysis of genomic data.

The current state-of-the-art in parallel analysis of genomic data is very limited. There are many serial software suites that lack any parallelization capability [19], [23], [24], [26], [30]. General MapReduce frameworks such as Hadoop [36] can potentially be used, but even parallelization using such a framework can be a hard problem for the community [20], [29]. Particularly, developing a parallel implementation with the use of MapReduce will require knowledge of data formats and reprogramming of an existing serial program, neither of which is desirable. Load imbalance is another problem while processing genomic data, which requires nuanced solutions. Genome Analysis Tool Kit (GATK) [27] provides developers a MapReduce-like framework specific for analysis of genomic data. However, GATK does not provide distributed memory parallelization for most of its programs and its scalability is low, as we will show in Section VI.

Thus, we can see that easing parallelization of applications that process genomic data, while also achieving high parallel efficiency, is a challenging problem. In this study, we are proposing PAGE, which is a MapReduce-like middleware that allows users to continue to utilize their existing programs, but achieve parallelization with a modest additional effort. The map and reduce programs can be separate executables written in any (and even two distinct) programming languages. We can also work with any of the popular genomic data formats. The middleware divides the genome structure into regions/intervals and runs each map task for a different region, performing load balancing in the process. Subsequently, these partial results are reduced to obtain the final result(s). In order to improve efficiency, our middleware provides streaming task scheduling, in which the tasks can be scheduled dynamically with the help of a master node, and the reduce tasks do not have to wait for all map tasks to be finished.

We evaluate PAGE with 4 existing applications, which are

1http://aws.amazon.com/1000genomes/
2http://aws.amazon.com/1000genomes/
<table>
<thead>
<tr>
<th>Locations</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Bases</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>T</td>
<td>A</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Read-1</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read-2</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read-3</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>T</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read-4</td>
<td>C</td>
<td>G</td>
<td>T</td>
<td>T</td>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read-5</td>
<td>T</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig 1. Representation of an alignment file with reference bases.

VarScan [19], and three applications currently supported by GATK, which are Unified Genotyper [7], Realigner Tagger Creator [9], and Indel Realigner [8]. We compare the performance of PAGE against GATK for the three applications currently available as part of GATK, and against Hadoop for VarScan. In both cases, we show that PAGE performs significantly better. The scalability of PAGE for these applications is 12.7x, 12.8x, 14.1x and 9x, respectively, when the computing power is increased by 16 times.

II. GENOMIC DATASETS, APPLICATIONS, AND PARALLELIZATION

A. Genetic Data

Using sequencing technologies such as Illumina/Solexa, we obtain genomic data, which comprises nucleotide sequences (sequences of bases, i.e., A, T, G, or C). Subsequently, these sequences can be mapped to particular regions of the genome by a sequence alignment program [21], [25], [23]. Fasta is a common (and simple) format, where nucleotide sequences are stored in the text form. The results of a sequence alignment program is more complex, as the file comprises of reads, where each read is a sequence of nucleotides of a varying length. Each read is mapped to a region of the genome, with each mapping having features like location, bases in the sequence, strand (direction) of the sequence, quality of the bases in the sequence, existence of insertion or deletions, and others. A representation of alignment of reads with reference bases is shown in Figure 1.

Process of identifying SNPs may seem like a simple data scan problem (though over a massive dataset, because of the length of each genome). However, the errors during digitization of a sample’s genetic data and the alignment files’ nature complicates their detection. Specifically, we need to find differences in nucleotides across different samples by looking at potentially unreliable or unmatched information across different reads while also trying to eliminate false positives.

**Detecting Problematic Regions:** During the process of alignment, some regions may require special attention due to problems like low quality, or misaligned or duplicate reads. Identifying these problematic regions is important for accuracy of the later analyses. Detection of indels (insertion or deletion) [9] and duplicates are examples of this type of applications.

**Quality Improvement:** As mentioned above, issues like misaligned or low quality read sequences impact the correctness of the further analyses. Therefore, the alignment files may need to be re-processed to increase their overall quality. Eliminating the duplicates or applying realignment for the suspicious intervals are examples of these applications.

C. Parallelization of Algorithms

We examine the parallelization of the algorithms we have introduced from two perspectives: 1) parallelization of processing a single genome, and 2) parallelization when multiple genomes (from different members of the species) are processed.

**Parallelization of Processing a Single Genome:** We can group algorithms in 3 groups according to how they process a single genome, and how it affects their correct parallelization. **Locus-based Algorithms:** Each single location is considered separately and can be processed independently. Therefore, the genome can be divided into separate blocks by splitting it at any point, and the parallel execution will still be correct. **SNP Calling:** In comparing the genomes of two individuals of the same species, at some locations only a single nucleotide may vary. These single variations in the genes are called Single Nucleotide Polymorphism (SNP), which can have a significant effect on the biological difference between members of the same species. For example, it is known that only one SNP is responsible for the earwax type [38]. Therefore, detection of SNPs and analyzing their effects for diseases is crucial, with applications including personalized medicine [13], [19], [26], [35].

**Detecting Problematic Regions:** During the process of alignment, some regions may require special attention due to problems like low quality, or misaligned or duplicate reads. Identifying these problematic regions is important for accuracy of the later analyses. Detection of indels (insertion or deletion) [9] and duplicates are examples of this type of applications.

**Quality Improvement:** As mentioned above, issues like misaligned or low quality read sequences impact the correctness of the further analyses. Therefore, the alignment files may need to be re-processed to increase their overall quality. Eliminating the duplicates or applying realignment for the suspicious intervals are examples of these applications.

B. Representative Applications for Processing Genomic Data

A variety of analysis tasks are applied on genomic data stored in FASTA, SAM, BAM, or other related formats. Such analysis can include simple diagnostic tools that obtain basic statistical information, perform quality controls for analyzing errors on the genetic data, aligning read sequences, variant identification, comparison of genetic data of different samples, or others. We explain three representative data analysis tasks below.

3http://www.illumina.com/technology/solexa_technology.ilmn
4http://zhanglab.ccmb.med.umich.edu/FASTA/
5MarkDuplicates feature of Picard
6For example, rmdup feature of SamTools
problem, where a read sequence can be mapped to any region of the genome [8], [21], [23], [25].

Parallelization of Multiple Genome Processing: Most genomic data analysis tasks today involve processing of multiple genomes (i.e., from different samples or members of the same species). The algorithms involved may do so in two different ways:

Independent Processing of Genomes: The same algorithm may be applied to each sample separately to obtain a different output for each of them. These outputs could be combined together in the end. Alignment [21], determining low quality regions [9], realigning [9] and improving the quality of the files [11] are all examples of these.

For the parallelization of this group of applications, the application can be executed separately in different computing nodes. However, the challenge is load balancing. Particularly, input file sizes may vary drastically. For example, the size of the files for different samples ranges from 50 MB to 5 GB in the pilot dataset of the 1000 Human Genome Project, which is sequenced by MOSAIK7. Thus, a parallelization framework needs to address load balance problem, and hide the solutions from the user.

Inter-dependent Processing of Genomes: In algorithms for SNP calling, or those for obtaining statistical information summarizing all inputs, or finding similarities/differences of genomes8, data from multiple genomes is accessed at the same time. In other words, these applications produce a single output while processing multiple genomes.

The main challenge of these applications is that data from all samples should be available together. There can be three methods of data distribution for parallelization as shown in Figure 2. These three methods are: 1) Input files can be grouped and each group can be processed separately following by a communication step among the nodes to produce the final result. 2) The genome structure can be divided into pieces and each node is responsible for a particular region of all input files. All nodes generate a result for a region and then the results are combined. This division is suitable for locus-based algorithms because each location can be processed independently. It can work well for region-based algorithms as well, though more caution is needed while partitioning the problem. 3) The division can be done in a checker-board style, where nodes are responsible for a subset of data from a subset of samples. The advantage of this method can be a better workload balancing, but implementation can become more complex.

III. MIDDLEWARE DESIGN

A. Middleware Goals

The overall goal of the system we have built is to ease parallel implementations of applications that analyze genomic data. More specifically, we had the following goals: 1) the system should allow parallelization of a variety of algorithms, 2) the middleware should be able to work with different popular formats used for genomic datasets, and new user code written for parallelization should be independent, to the extent possible, of a specific format, 3) the middleware should allow use of existing programs, including programs in different languages, as part of sequential processing on each node - similarly, the middleware should allow use of existing tools for partitioning genomes and combining outputs from processing of different samples. 4) The system should achieve high parallel efficiency - particularly, the load imbalance problem should be addressed by the system, and without requiring user effort.

B. Execution Model

We now describe how our middleware helps execute an application that processes genomic data in parallel. Figure 3 illustrates the execution.

Map Task: The map task is typically an executable, which operates on either the entire or a part of the genome. The user defines the program to be executed, together with one or more input files that will be processed.

Reduction: The results from the invocation of each map task need to be combined or reduced together. The user has two options for the implementation of these reduction programs. In full reduction, the reduction program takes output from all map invocations and generates a single output. In partial reduction, the reduction program takes output from two processes and combines them. In this case, the middleware runs the reduction program repeatedly in a tree structure in order to obtain the final output. The partial reduction approach is useful while load balancing the work.

Genome Partitioning Method: Irrespective of the file format used (e.g. SAM [24], VCF [15], BED9, or another format), the genetic data of species has an analogous genetic structure, that is, the number of chromosomes and their nucleotides are the same. We can take advantage of this information for partitioning the workload. Instead of forming chunks with equal sizes, ignoring location information, as would normally happen with a system like Hadoop [36], dividing the genome according to location information may work better. In addition, equal-sized chunks may cause correctness issues because location information is also biologically significant (i.e. each gene exist in a certain region). Particularly, whether a chromosome can be further divided across nodes or not depends upon the nature of the application. Because of a limited number of chromosomes within a genome, our ability to scale and load balance a computation becomes limited if processing within a chromosome cannot be correctly partitioned. Our middleware

7http://code.google.com/p/mosaik-aligner
8for example diff feature of BambUtils [2]
9http://useast.ensembl.org/info/website/upload.bed.html
asks whether the partitioning can be by-locus, and if that is not the case, a by-chromosome partitioning is performed. Since the number of bases is different for each chromosome, our middleware tries to decrease the imbalance among regions by taking into account the chromosome lengths.

Another factor for partitioning is whether the processing across samples is independent or inter-dependent. If the former, we have greater flexibility in achieving load balance, whereas in the latter case, we have to divide the genome locations for a single genome.

Load Balancing Method: The genome needs to be divided into small regions according to the partitioning method. Then all of these regions should be assigned to computing nodes. Scheduling of the processing of regions, i.e., the tasks, is one of the challenges for the middleware. Our middleware provides two types of scheduling schemes, which are the static and dynamic schemes. In static scheduling, the number of regions is set to number of processors and regions are determined according to partitioning method. Each of these regions are assigned to a different processor. After all map tasks finish, the reduction phase can begin. The problem with such scheduling is that it may not work well for all cases, because of file size differences among the inputs and coverage variance of locations, as described in the previous section. Therefore, our middleware also provides dynamic scheduling. This scheme employs one of the processors as the master node and operates in a streaming fashion. The details of this scheduling scheme are explained in the next section.

C. Middleware Applicability

The execution model described above creates certain restrictions on the kind of processing that can be parallelized. First, the map tasks must be able to take interval/region parameter as input or must be able to work together with programs that are able to partition genomes into pieces such as SamTools [24]. Next, the reduction should have one of the formats we had discussed above. Finally, the algorithm should be safe to be parallelized by processing different areas of the genome independently. Formally, let $P$ be the map function, $R$, $R_1$, and $R_2$ be three regions, such that concatenation of $R_1$ and $R_2$ gives us $R$. Then the following property should hold:

$$P(R) = P(R_1) \oplus P(R_2)$$

where, $\oplus$ is the reduction function. It is easy to see that all locus-based algorithms can be parallelized by this system. For region-based and genome-based algorithms, the correctness depends upon the specific algorithm and the partitioning function used. For example, for the region-based algorithms if we divide the data only by chromosomes, the algorithm can be parallelized correctly, though with some limitations on its scalability.

IV. Middleware Implementation

Our system involves a carefully optimized implementation of basic design described in the previous section. Particularly, we use the streaming idea, i.e., the reduce tasks can start even though there are map tasks still executing, with the goal of improving workload balance and increasing performance. The pseudo-code for this scheme is given as Algorithm 1.

In order to schedule tasks dynamically, we assign one of the nodes as the master node, i.e., to manage the flow of tasks while the other nodes do the actual computations. The master node first initializes two queues $M$ and $R$, which keep map and reduce tasks to be executed, respectively. Note that if there will be separate outputs for each input, there will be a separate $R$ queue for each output file. For simplification, we assume that there will be single output. The master also initializes the ProcStat array, which keeps workers’ status information (Line 2). A worker can be in MAP, REDUCE, or IDLE status. The master node also calculates the regions of the genome according to genome division method (Line 3) and inserts these regions into the queue $M$ as map tasks (Line 4). Note that if we apply the by-chromosome partitioning, each region will be a different chromosome. Instead, if we apply the by-locus partitioning, we have $N$ equal length regions. If there will be multiple outputs, we sort the input files according to their size and put the regions of the larger inputs first into the queue in order to increase the performance.

The master node starts waiting a task request from the worker nodes (Line 6). When it receives a request, it first checks what type of task the requesting node has finished. If the requesting worker finished a map-task, the master node inserts address of the intermediate result files of that finished map task to the queue $R$ (Lines 7-8). Note that a map-task can produce multiple results to be inserted into the queue $R$. If requesting worker finished a reduce task, and if it is not the
Algorithm 1 PAGE’s Streaming Dynamic Scheduling Algorithm

1: **Master Node**
2: Initialize map-tasks queue $M$, reduce-tasks queue $R$ and processor status array $ProcStat$
3: Divide the genome(s) into $N$ chunks
4: Enqueue $N$ map tasks into $M$
5: repeat
6: Wait for a request from workers
7: if Requesting node finished a map task ($T_M$) then
8: enqueue result(s) of $T_M$ to $R$
9: else if Requesting node finished a reduce task ($T_R$) & $T_R$ is not the final reduce task then
10: enqueue result of $T_R$ to $R$
11: end if
12: if $M$ is not empty then
13: $T_M = dequeue(M)$
14: send $T_M$ to the requesting node
15: $ProcStat[requestingnode] = MAP$
16: else if final output is not produced yet then
17: $ProcStat[requestingnode] = IDLE$
18: for all IDLE nodes $I$ do
19: if $R$ has sufficient entries then
20: $T_R = entries$ dequeued from $R$
21: send $T_R$ to $I$
23: else
24: break
25: end if
26: end for
27: else
28: break
29: end if
30: until all tasks are done
31: Send end message to all nodes
32: **Worker Node**
33: repeat
34: Request a task from master node
35: if task is a $MAP$ then
36: intervals = intervals received from master node
37: map(intervals)
38: else if task is a $REDUCE$ then
39: files = files received from master node
40: reduce(files)
41: else
42: break
43: end if
44: until true

During this process, worker nodes simply execute a loop. At the beginning of each iteration, they request a task from the master node (Line 34). They can run map task (Lines 35-37) or reduce task (Lines 38-40) depending on the message they received from the master node. Once they receive the end message, they finish the execution.

V. APPLICATIONS DEVELOPED AND PROGRAMMABILITY

We implemented our middleware in C programming language with the MPI library. We implemented four applications that have different properties, leading to differences in the way they use the middleware and are parallelized. Three of these applications are from the Genome Analysis ToolKit (GATK) [27], which is another framework for parallelizing genomic applications. One of the advantages of using GATK-based applications is that we can compare performance against parallelization performed by GATK. We now explain these applications.

**Realigner Target Creator (RTC):** RTC [9] is one of programs available from GATK. RTC finds suspicious indels in the genome that may need realignment. The execution model for RTC is independent processing since each indel interval list is special for each input file. The reduction is simple concatenation of all output files, which are the list of intervals.

**Indel Realigner (IR):** IR [8] is another program available from GATK. IR takes the output of RTC, which is a list of indels to be realigned, as input, and applies a local realignment. Its execution model is independent processing, since realignment is specific operation for each alignment file. We used merge feature of SamTools for reduction of intermediate results, which are basically alignment files.

**Unified Genotyper (UG):** UG [7] is yet another program from GATK that finds SNPs and indels by applying Bayesian genotype likelihood model and generates a single VCF file. We used concatenate function of VCFTools for the reduction of the intermediate results.

**VarScan:** VarScan [19] is a commonly-used serial Java-based variant detection program, which finds variants like SNPs and indels among multiple genomes. It takes multiple alignment files and a reference file to detect variants and generates a single output file. It is a locus-based algorithm. One important drawback of the software is that it cannot process the alignment files which are in SAM or BAM formats, instead, it requires a pre-processing step in which all the alignment files and the reference genome data are piled up into a single text-based file that is called the mpileup file. Normally, VarScan does not have any parameters to process a specific region. However, we can take advantage of SamTools, which can generate mpileup files and take a parameter to process only specific regions. For the reduction, we used cat bash shell command to concatenate the output files, which are basically lists of SNPs.

In Table I, the map task commands for using these applications with PAGE, their execution models, and the reduction methods we used are shown. The bolded regionloc, inputloc, extrainputloc, outputloc texts in map task commands are reserved words of PAGE, helping to use the genomic applications and showing where to insert region, input and output file parameters in the corresponding genomic application’s execution command.
### Table I

<table>
<thead>
<tr>
<th>Application</th>
<th>Map-Task Command</th>
<th>Execution Model</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC</td>
<td>java -jar GenomeAnalysisTK.jar -nt 1 -L inputloc -R reference_file -T RealignerTargetCreator -o outputloc</td>
<td>Independent</td>
<td>cat bash shell command</td>
</tr>
<tr>
<td>IR</td>
<td>java -jar GenomeAnalysisTK.jar -nt 1 -L inputloc -R regionloc -T IndelRealigner -o outputloc</td>
<td>Independent</td>
<td>merge feature of SamTools</td>
</tr>
<tr>
<td>UG</td>
<td>java -jar GenomeAnalysisTK.jar -nt 1 -L inputloc -R regionloc -T UnifiedGenotyper -o outputloc</td>
<td>Inter-dependent</td>
<td>concatenate feature of VCFTools</td>
</tr>
<tr>
<td>VarScan</td>
<td>samtools mpileup -b alignment_file -r regionloc -f reference_file</td>
<td>Inter-dependent</td>
<td>cat bash shell command</td>
</tr>
</tbody>
</table>

### Table II

<table>
<thead>
<tr>
<th>Application</th>
<th>Data Thread</th>
<th>CPU Thread</th>
<th>GATK-Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Realigner Target Creator (RTC)</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Indel Realigner (IR)</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Unified Genotyper (UG)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### A. Programmability and Parallelization Comparison

Two closest frameworks to PAGE are Hadoop Streaming[3] and GATK. Hadoop Streaming is a utility based on the popular framework Hadoop [36]. The key feature of Hadoop Streaming is that it allows users to use executable programs and scripts as map and reduce tasks. As we have stated earlier, GATK is a comparable framework that provides parallelization at different levels. It has 2 types of threads, i.e., data and CPU threads for shared memory parallelization. Particularly, with respect to the three tools we have used, RTC can be parallelized by employing multiple data threads while UG can use both types of threads. On the other hand, there is no shared-memory parallelization for IR. In order to maintain distributed memory parallelization, GATK can work together with another software called GATK-Queue[10] which is a scripting framework for genomic analysis. GATK-Queue runs GATK on different nodes for different regions of the genome, and then gathers the intermediate results to form the final output. GATK-Queue can work together with GATK's own shared-memory parallelization. However, GATK-Queue can be used only for UG and IR, not for RTC. The parallelization capability of GATK for RTC, IR, and UG is summarized in Table II.

In the next section, we will compare performance of PAGE against Hadoop Streaming (for VarScan) and GATK (for the other three applications). Before that, however, we compare the programming effort involved with these systems.

In parallelization using GATK, one does not need to write any code for map and reduce tasks. In other words, parallelization with GATK and PAGE can be considered very comparable.

Now, focusing on Hadoop Streaming, which we used to parallelize VarScan - in order to run VarScan in parallel, we need to perform 3 operations: 1) Conversion of alignment files in the BAM format and reference file in the Fasta format into the mpileup file format 2) Running VarScan to detect SNPs, 3) Concatenating lists of SNPs to form the final output. A comparison of how these 3 operations are performed with Hadoop Streaming and PAGE is shown in Table III. As can be seen from the Table, PAGE does not require much effort because of its ability to employ SamTools (and take available options for partitioning in SamTools). In comparison, Hadoop Streaming needs to deal with data formats explicitly. We have shown two independent approaches for parallelizing an application like VarScan with Hadoop Streaming in Table III. The two approaches require comparable programming effort. In the execution times we report, the scripting method was used.

### VI. Experimental Results

In this section, we report results from a series of experiments we conducted to examine the performance of our middleware, using four genomic applications we had described earlier, and compare PAGE with Hadoop Streaming and GATK.

#### A. Experimental Setup and Parameter Configurations

In our experiments, we used a cluster where each node has 8 cores 2.53 GHz Intel (R) Xeon (R) processor and 12 GB memory. We used the genomic data from the 1000 Human Genome Project, which was downloaded from the Amazon Cloud Storage Systems. We selected 40 random samples from the pilot data set and used their alignment files sequenced by MOSAIK on different applications. In order to test number of threads for shared memory parallelization of GATK for UG, we run it with different numbers of parameters for a training dataset and selected the parameter set that gave the best result. We performed the same method to set the number of regions for dynamic scheduling scheme (while using by-locus partitioning with PAGE).

#### B. Experimental Results

We now evaluate the performance of PAGE, with two sets of experiments focusing on scalability while increasing computing power, and performance as the data volume is increased.

**Parallel Scalability Evaluation:** We evaluated the parallel scalability of applications with PAGE by varying number of nodes (and cores) we allocated, while also varying the scheduling schemes and data partitioning methods. We used by-locus partitioning with all 4 applications while by-chromosome partitioning is applied only for RTC and IR. This is because the execution model of the other two applications, VarScan and UG, is inter-dependent processing in which the maximum number of data chunks is limited by the number of chromosomes in the genome (24 in our case). We used 10

---

files for RTC and IR, 30 files for UG and 40 files for VarScan. The results are shown in Figure 4.

In scalability experiments with RTC, we are not able to use GATK-Queue, and thus no comparison of distributed memory scalability is feasible. In order to see its shared-memory parallelization capability, we run RTC for 10 files with 8 threads and the execution took 78012 seconds. Subsequently, we performed scalability experiments with RTC with different parameters of PAGE. As seen from the Figure 4(a), PAGE’s all executions with 8 cores (within a single node) outperform GATK. In observing the performance, as the number of nodes increases, static scheduling has higher performance with lower number of nodes, which is because with dynamic scheduling, one of the cores is allocated as the master process. Static scheduling with by-locus partitioning has better performance than dynamic scheduling with by-locus partitioning. However, when we apply by-chromosome partitioning and use more than 16 cores, dynamic scheduling outperforms static scheduling. By increasing number of cores from 8 to 128 and using by-locus partitioning, the speed-ups obtained for static and dynamic scheduling schemes are 14.1x and 17x (number of workers increased from 7 to 127), respectively. When we apply by-chromosome partitioning, the speed-ups for static and dynamic scheduling schemes decreased to 5.5x and 12.3x, respectively, as expected. Overall, these results show that dynamic scheduling is better than static scheduling in terms of maintaining workload balance among nodes when the chunks are not of equal size.

RTC can be considered a region-based algorithm, since its output contains a list of regions. However, a by-locus partitioning does result in correct parallelization, and particularly, we can see that PAGE produced the same output with GATK for both partitioning types.

In scalability evaluation of IR, we compared PAGE with the GATK-Queue framework. As noted in the title of the Figure 4(b), GATK is unable to use multiple cores within each node. Thus, it is not surprising that the absolute performance for PAGE is much better. But, it is interesting to note that going from 1 node to 16 nodes, GATK’s speed increased by 3.25x while PAGE’s speed increased by 9x with dynamic scheduling and by-locus partitioning. We can again see that using our dynamic scheduling has better performance than static scheduling when we apply by-chromosome partitioning.

In scalability evaluation of UG, we compared PAGE with parallelization of GATK and GATK-Queue. We increased number of nodes from 1 to 16. In each node, we employed all cores for PAGE’s execution while we used 2 data threads and 7 CPU threads for GATK’s execution. As seen from the Figure 4(c), PAGE’s both scheduling schemes outperform GATK.

In terms of actual execution time, static scheduling gives better results for lower scalability while dynamic scheduling performs better for higher number of nodes. When the number of cores is increased by 16x, execution time of PAGE-dynamic, PAGE-static and GATK increased by 12.8x, 11x and 10.9x, respectively.

VarScan is not supported on GATK, and our comparison focused on using Hadoop Streaming. In previous experiments, we increased number of nodes from 1 to 16. However, using single node was not enough to keep all converted data (roughly around 510 GB) in HDFS. Therefore, we increased number of nodes from 8 to 128 while keeping the computing power at the same level by allocating single core on each node. In order to achieve this in Hadoop’s execution, we set the maximum number of map tasks running simultaneously by a task tracker to 1. As seen from Figure 4(d), PAGE has better performance in terms of actual execution time for all cases. PAGE achieves 12.7x speedup with static scheduling and 11.8x speedup with dynamic scheduling while the execution with Hadoop achieves 6.9x speedup.

Scalability With Respect to Dataset Sizes: In this set of experiments, we varied the number of files to be processed in order to see the performance of PAGE, Hadoop and GATK with different dataset sizes. We used 128 processors for PAGE’s executions and used the equivalent parameter settings for GATK and Hadoop. We run our 4 applications with these systems, and increased the load from 1 to 16 alignment files (all of similar sizes). The results are shown in Figure 5.

For RTC experiments, we are unable to make any comparison against another platform. However, we can see that the increase in execution time for the best version is at most linear. The by-chromosome partitioning has lower performance than by-locus partitioning, as expected. For the low number of files, its performance is worse because of not being able to use all the nodes due to insufficient number of chromosomes. With by-chromosome partitioning, static scheduling gives better results than dynamic scheduling in this experiment, in contrast with the previous experiment. This is because the sizes of files used in previous experiment had a higher variance than the files used in this experiment. Overall, we can conclude that static scheduling is preferable when the data can be divided into equal chunks; on the other hand, dynamic scheduling does better managing unequal sized chunks.

In experiments with IR, recall that GATK-Queue can only use 1 of the 8 cores. Still, we can see that performance of our system is better by more than a factor of 8. Dynamic scheduling also outperforms static scheduling for all cases. When the data set size increased by 16x, dynamic scheduling’s execution times increase only by 4.7x and 2.7x for by-locus

<table>
<thead>
<tr>
<th>Data Conversion</th>
<th>Bash Scripting</th>
<th>OR Hadoop Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Manually divide genome into equal regions</td>
<td>• Define BAM and Fasta file formats to Hadoop</td>
</tr>
<tr>
<td></td>
<td>• Run SamTools for each region in parallel</td>
<td>• Run SamTools as a map task</td>
</tr>
<tr>
<td></td>
<td>• Upload the data to HDFS</td>
<td>• Merge results in mpileup format as a reduce task</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Processing</th>
<th>Run VarScan as a map task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td>cat bash shell command</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hadoop Streaming</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bash Scripting</td>
<td>Run SamTools &amp; VarScan together as a map task</td>
</tr>
</tbody>
</table>

<p>| PROGRAMMABILITY COMPARISON OF HADOOP STREAMING AND PAGE |
|--------------------------------------------------------|---------------------------------------------------|</p>
<table>
<thead>
<tr>
<th>Hadoop Streaming</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bash Scripting</td>
<td>Run SamTools &amp; VarScan together as a map task</td>
</tr>
</tbody>
</table>

<p>| TABLE III |
|-----------|----------------|</p>
<table>
<thead>
<tr>
<th>Data Conversion</th>
<th>Bash Scripting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Manually divide genome into equal regions</td>
</tr>
<tr>
<td></td>
<td>• Run SamTools for each region in parallel</td>
</tr>
<tr>
<td></td>
<td>• Upload the data to HDFS</td>
</tr>
<tr>
<td>OR Hadoop Implementation</td>
<td>Run SamTools &amp; VarScan together as a map task</td>
</tr>
<tr>
<td>• Define BAM and Fasta file formats to Hadoop</td>
<td>• Run SamTools as a map task</td>
</tr>
<tr>
<td>• Merge results in mpileup format as a reduce task</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Processing</th>
<th>Run VarScan as a map task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td>cat bash shell command</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCALABILITY EVALUATION OF RTC</th>
</tr>
</thead>
</table>
| In scalability experiments with RTC, we are not able to use GATK-Queue, and thus no comparison of distributed memory scalability is feasible. In order to see its shared-memory parallelization capability, we run RTC for 10 files with 8 threads and the execution took 78012 seconds. Subsequently, we performed scalability experiments with RTC with different parameters of PAGE. As seen from the Figure 4(a), PAGE’s all executions with 8 cores (within a single node) outperform GATK. In observing the performance, as the number of nodes increases, static scheduling has higher performance with lower number of nodes, which is because with dynamic scheduling, one of the cores is allocated as the master process. Static scheduling with by-locus partitioning has better performance than dynamic scheduling with by-locus partitioning. However, when we apply by-chromosome partitioning and use more than 16 cores, dynamic scheduling outperforms static scheduling. By increasing number of cores from 8 to 128 and using by-locus partitioning, the speed-ups obtained for static and dynamic scheduling schemes are 14.1x and 17x (number of workers increased from 7 to 127), respectively. When we apply by-chromosome partitioning, the speed-ups for static and dynamic scheduling schemes decreased to 5.5x and 12.3x, respectively, as expected. Overall, these results show that dynamic scheduling is better than static scheduling in terms of maintaining workload balance among nodes when the chunks are not of equal size.

In scalability evaluation of IR, we compared PAGE with the GATK-Queue framework. As noted in the title of the Figure 4(b), GATK is unable to use multiple cores within each node. Thus, it is not surprising that the absolute performance for PAGE is much better. But, it is interesting to note that going from 1 node to 16 nodes, GATK’s speed increased by 3.25x while PAGE’s speed increased by 9x with dynamic scheduling and by-locus partitioning. We can again see that using our dynamic scheduling has better performance than static scheduling when we apply by-chromosome partitioning.

In scalability evaluation of UG, we compared PAGE with parallelization of GATK and GATK-Queue. We increased number of nodes from 1 to 16. In each node, we employed all cores for PAGE’s execution while we used 2 data threads and 7 CPU threads for GATK’s execution. As seen from the Figure 4(c), PAGE’s both scheduling schemes outperform GATK.

In terms of actual execution time, static scheduling gives better results for lower scalability while dynamic scheduling performs better for higher number of nodes. When the number of cores is increased by 16x, execution time of PAGE-dynamic, PAGE-static and GATK increased by 12.8x, 11x and 10.9x, respectively.

VarScan is not supported on GATK, and our comparison focused on using Hadoop Streaming. In previous experiments, we increased number of nodes from 1 to 16. However, using single node was not enough to keep all converted data (roughly around 510 GB) in HDFS. Therefore, we increased number of nodes from 8 to 128 while keeping the computing power at the same level by allocating single core on each node. In order to achieve this in Hadoop’s execution, we set the maximum number of map tasks running simultaneously by a task tracker to 1. As seen from Figure 4(d), PAGE has better performance in terms of actual execution time for all cases. PAGE achieves 12.7x speedup with static scheduling and 11.8x speedup with dynamic scheduling while the execution with Hadoop achieves 6.9x speedup.

Scalability With Respect to Dataset Sizes: In this set of experiments, we varied the number of files to be processed in order to see the performance of PAGE, Hadoop and GATK with different dataset sizes. We used 128 processors for PAGE’s executions and used the equivalent parameter settings for GATK and Hadoop. We run our 4 applications with these systems, and increased the load from 1 to 16 alignment files (all of similar sizes). The results are shown in Figure 5.

For RTC experiments, we are unable to make any comparison against another platform. However, we can see that the increase in execution time for the best version is at most linear. The by-chromosome partitioning has lower performance than by-locus partitioning, as expected. For the low number of files, its performance is worse because of not being able to use all the nodes due to insufficient number of chromosomes. With by-chromosome partitioning, static scheduling gives better results than dynamic scheduling in this experiment, in contrast with the previous experiment. This is because the sizes of files used in previous experiment had a higher variance than the files used in this experiment. Overall, we can conclude that static scheduling is preferable when the data can be divided into equal chunks; on the other hand, dynamic scheduling does better managing unequal sized chunks.

In experiments with IR, recall that GATK-Queue can only use 1 of the 8 cores. Still, we can see that performance of our system is better by more than a factor of 8. Dynamic scheduling also outperforms static scheduling for all cases. When the data set size increased by 16x, dynamic scheduling’s execution times increase only by 4.7x and 2.7x for by-locus
and by-chromosome partitioning methods, respectively, while static scheduling’s execution times increase by 11.7x and 5.4x, respectively. Because the amount of computation is increasing at least linear with dataset sizes, these results show that our system is not introducing any overheads of dealing with large-scale data.

From all our experiments, we can see that offering multiple scheduling schemes and partitioning methods turns out to be important for achieving speedups for different applications and dataset sizes. While we do not yet have a mechanism to automatically choose the best scheme for a given application and dataset, we believe that a user is likely to be able to make this choice, based on a broad understanding of their application and the dataset.

VII. RELATED WORK

There has been considerable work for developing parallel solutions for specific genomic algorithms, such as genomic alignment [32], [37], sequence comparison [17], searching local alignments [14], and error correction on short reads [33]. The advantage of our framework is in simplifying parallelization of a number of algorithms. BioConductor [18], which is primarily based on the R statistical programming language, includes certain parallel tools [5], but cannot be used to parallelize new applications.

Following the general trend in various commercial and scientific areas, the MapReduce [16] framework has been used for parallel genomic applications as well. Examples of applications parallelized include aligning and SNP calling [20], read mapping and duplicate removal [29], QTL association, and statistical analysis [22]. Use of the original map and reduce functions makes it harder to implement complex analyses, leading to some correctness issues (see, for example [29]). Among the challenges introduced by different data formats, data division is not easy for binary file formats like BAM or BCF. Even though libraries like Hadoop-BAM [28] can be used to process BAM files, equal-sized data blocks that ignore the location information of genetic data introduces complexity in implementation. Hadoop Streaming [3] can be considered the closest to the PAGE system, as it allows executables for map and reduce tasks. We have compared the performance with a popular SNP calling algorithm, and shown that our system results in better performance. Finally, we note that the Genomic Analysis Tool Kit (GATK) [27] was also developed to address the same limitations. We have extensively compared the performance of our system against GATK.
Other tools have been developed in context of genomic data. SamTools [24] allows user to do operations such as merging, indexing, sorting and so on, and also provides a library in C that can be used for developing other tools. BEDTools [30] is another program which provides various genomic data operations, for example, intersecting 2 BED files, computing coverage in BAM files, merging overlapping features and so on. However, BedTools does not provide a library to be used for developing tools. GenomicTools [34] is a flexible computational platform that allows users to perform operations on GFF, SAM/BAM and BED formats. Besides its command-line tools, it also provides API in C++. However, none of them support parallel application development.

Our work is also related to the problem of task scheduling in parallel systems. Because of a large volume of existing work in this area, we restrict ourself to the most similar system, which is MapReduce. In the conventional MapReduce approach, each task is assigned dynamically and reduce tasks cannot be started before all map tasks finish. A combiner function can be used to reduce the amount of intermediate results, and the task scheduling works in a first-in, first-out fashion, assuming that each task will require similar execution time. Because this simple approach may not work well for many cases, a number of studies have developed other schemes. This includes work on execution in a heterogeneous environment [31], clusters with multiple users [39] and improving data locality [40]. PAGE’s streaming dynamic scheduling differs from the conventional MapReduce approach, because reduce tasks do not have to wait for all map tasks to finish. Our partial reduction mechanism is also different from the use of a combiner function, because unlike combiner function, partial reduction can be applied to intermediate results produced by different nodes.

VIII. CONCLUSION

In this paper, we have a novel middleware, PAGE, which allows users to parallelize genomic applications in a convenient fashion. To maintain performance on different applications and datasets, PAGE supports multiple scheduling schemes, partitioning methods, and execution models. We have evaluated PAGE with four different programs, which are VarScan, Indel Realigner, Realigner Target Creator, and Unified Genotyper, and compared PAGE against GATK and Hadoop Streaming.

The main observations from our extensive evaluation are as follows. First, PAGE is able to parallelize different applications, maintaining high parallel efficiency. Second, each scheduling scheme can be useful for a certain type of environment and application - for example, dynamic scheduling is better when chunks are of unequal sizes, while static scheduling can be better when the number of nodes is smaller.
Finally, in comparison of PAGE with Hadoop Streaming and GATK, we show that it has better performance and scalability.

REFERENCES