Open access • Journal Article • DOI:10.1007/S10766-017-0524-Z

Combining Hadoop with MPI to Solve Metagenomics Problems that are both Data- and Compute-intensive — Source link

Han Lin, Zhichao Su, Xiandong Meng, Xu Jin ...+5 more authors

Institutions: University of Science and Technology of China, Lawrence Berkeley National Laboratory

Published on: 01 Aug 2018 - International Journal of Parallel Programming (Springer US)

Topics: Speedup

Combining Hadoop with MPI to Solve Metagenomics Problems that are both Data- and Compute-intensive

Han Lin[1], Zhichao Su[1], Xiandong Meng[2], Zhong Wang[2], Hong An[1]

[1] University of Science and Technology of China
[2] DOE Joint Genome Institute, Genomics Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
{linhan09, suzc}@mail.ustc.edu.cn
{xiandongmeng, zhongwang}@lbl.gov
han@ustc.edu.cn

Abstract. Metagenomics, the study of all microbial species cohabitants in an environment, often produces large amount of sequence data varying from several GBs to a few TBs. Analysing metagenomics data involving several steps, some steps are data intensive, and some are compute intensive. Typical bioinformatics pipelines attempt to analyse the entire data set on computer servers with several terabytes of RAM, which is very inefficient. To overcome this limit, here we propose a MapReduce based solution to partition the data based on their species of origin. We implemented the solution using BioPig, an analytic toolkit for large-scale genomic sequence data based on Apache Hadoop and Pig. We simplified data types and logic design, compressed k-mer storage and combined Hadoop with MPI to improve the computational performance. After these optimizations, we achieved up to 193x speedup for the rate-limiting step and 8x speedup for the entire pipeline, respectively. The optimized software is also capable to process datasets that are 16 times larger on the same hardware platform. Results from this case study suggest the combined Hadoop with MPI approach has great potential in large genomics applications that are both data-intensive and compute-intensive.

Keywords: Metagenomics, Hadoop, MPI, Optimization, Implementation, Pig Latin, BioPig, Big Data, Data-intensive, Compute-intensive

1 Introduction

Metagenomics represents the investigation of the microbes inside an environment(e.g. ocean, soil, and the human body) with sequencing technologies. The recently developed next-generation sequencing technologies[1] have dramatically increased the speed of DNA sequencing and enlarged the size of sequencing data. In the occasion of metagenomic sequencing, a large amount of sequence samples of various species are captured. To isolate every microbe from environmental samples by experimental methods is nearly impossible. So computational filtering of species is the key to solving the metagenomic problem.

The metagenomic sequence reads partition is a big-data problem, since the input data sets could be as large as tens or hundreds of GBs or even many TBs. Hadoop[2] is an open-source framework for big-data processing, which contains four core components: Hadoop Common, Hadoop Distributed File System(HDFS) , Hadoop YARN[3] and MapReduce[4] programming model. Hadoop Common provides the common utilities that support other modules; YARN is responsible for job scheduling and cluster resource management; HDFS ensures the reliable and distributed application data storage as well as high-throughput access to that; MapReduce permits numbers of tasks running in a massively parallel manner on a large-scale cluster. These features make Hadoop a very popular framework for big-data processing.

BioPig[5] is an analytic toolkit for large-scale sequence data based on Pig[6], which provides a SQL-style programming language called Pig Latin upon Hadoop's MapReduce programming model. With Pig Latin, programmer could write MapReduce applications on a higher level. Pig Latin can be extended by using User Defined Functions(UDFs), through which a user can control the data processing in a fine-grained style. BioPig contains some examples and necessary UDFs for genomic sequence data processing, such as sequence data load and store, k-mer generator, N50 calculator, assembler caller.

Reads clustering is a common method in metagenomics. Dime[7] is a framework for metagenomic sequence assembly, it provides a complete solution that consists of reads partition, assembly and merging. MrMC-MinH[8] is an algorithm for clustering metagenome reads based on Map-Reduce. However, none of them has been tested on large amount of data sets. SeqPig[9] is another Pig-based toolkit for process of large sequencing data sets. It also provides scalability over many computing nodes.

In the input data set, every input biological sequence is called a *read*, as usually mentioned by biologists. Every letter indicating a nucleotide(A, C, G or T) is called a *base*. Sometimes a fifth letter "N" will be used for uncertain. There are several common formats for storing reads. In our occasion, we use fastq format, which is a text-based format for biological sequence and its corresponding quality scores storing. A read sample represented in fastq format is showed in Fig.1. The first line starts with an '@' and followed by a READ_ID, which is a unique string representing the whole read. The second line is the sequence itself followed by a separator '+' in line 3. The last line is the sequence's quality score. Different character in this line indicates different quality of the corresponding base. The length of line 2 and line 4 must be equal.

```
@READ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

Fig. 1: A read in fastq format

K-mer is a significant concept in computational genomics. It refers to all the possible subsequences(of length k) inside a read. Figure 5 shows a simple example of a k-mer generator—generates k-mers from a read. K-mer step defines how many bases to step forward when calculating next k-mer.

In this paper, we present the implementation and optimization of an algorithm which partitions input reads based on the k-mer sharing information between reads pairs. The implementation is tested over the input of millions to hundreds of millions of reads. The methods to optimize our implementation, including simplifying data types, compressing k-mer storage, simplifying logic design, coupling MPI with Hadoop and making them work together, specialized Hadoop configurations, etc. are very instructive to similar work. And the idea that combine MPI with Hadoop reveals a new way to deal with big-data problems that are not both data-intensive and compute-intensive.

The rest of the paper is organized as follows. Section 2 presents our algorithm for reads partition. Section 3 explains our implementation based on BioPig. Section 4 presents the optimization strategies in detail. Section 5 shows the final results we get and section 6 discusses the whole work.

2 Reads Partition Algorithm

The k-mer sharing reads partition algorithm consists of three parts: **reads graph generation**, **graph partition** and **reads retrieving**. Reads graph generation generates a reads graph whose vertices represent input reads and edges represent k-mer sharing information among every reads pair. Graph partition is like a kind of cluster operation. It finds out all the connected components of the reads graph. The components that are in graph partition phase are disjoint subsets which contain partitioned reads. Reads retrieving maps every input read into different clusters according to the subset it belongs to. Reads graph generation and graph partition algorithms are described by Algorithm 1 and 2 respectively.

2.1 Reads Graph Generation

The algorithm is showed in Algorithm 1. The initial input of reads graph generation phase R is a set of samples that each of which consists of a set of reads. After processing, it outputs a reads graph RG, whose vertices represent reads and edges indicate relations of every reads pair.

Every input sample R_i will be checked one by one. After calculating every read's k-mers, we count the number of occurrences of every k-mer in the sample and filter the ones whose occurrence times is less than a min or greater than a max threshold. The filtered results are stored in UF. After the filter operation, the sample information is dropped.

The reserved information in UF is then used to check whether to keep an edge. Edge weight is the number of shared k-mers in the corresponding reads pair. Only the edges whose weight lies in the specified range are added into RG.

Algorithm 1 Reads Graph Generation

```
Input:
    Reads Set, $R = \{R_1, R_2, ..., R_m\}$;
    K-mer length, $k$;
    K-mer appearance threshold, $min\_k\_appear$ and $max\_k\_appear$;
    Edge weight threshold, $min\_edge\_weight$ and $max\_edge\_weight$
Output: Reads Graph, $RG$
 1: $UF \leftarrow \emptyset$
 2: for $R_i \in R$ do
 3:     $S \leftarrow \emptyset$
 4:     for $r \in R_i$ do
 5:         $k\text{-}mer\_set \leftarrow$ k-mer_generator$(r, k)$ as $\{(k\text{-}mer, readid)\}$
 6:         $S \leftarrow S \cup k\text{-}mer\_set$
 7:     end for
 8:     $B \leftarrow$ group $S$ by $k\text{-}mer$
 9:     $CK \leftarrow$ count($k\text{-}mer$ in B)
10:     $F \leftarrow$ filter $B$ by $CK \geq min\_k\_appear$ and $CK \leq max\_k\_appear$
11:     $UF \leftarrow UF \cup F$
12: end for
13: $E \leftarrow$ generate reads pair $(r_i, r_j)$ from $UF$
14: $RG \leftarrow \emptyset$
15: for $(r_i, r_j) \in E$ do
16:     $CE \leftarrow$ count($(r_i, r_j)$ in $UF$)
17:     if $CE \geq min\_edge\_weight$ and $CE \leq max\_edge\_weight$ then
18:         $RG \leftarrow RG \cup \{(r_i, r_j)\}$
19:     end if
20: end for
21: return $RG$
```

2.2 Graph Partition

The algorithm is showed in Algorithm 2. Graph partition splits the generated graph RG into disjoint subsets. Every subset is a connected component of the input reads graph.

The input graph E is described in the format of (r_i, r_j) so that we can ignore the isolated vertices. The found subsets are stored in a set V. For every new edge (r_i, r_j), we try to find how many subsets in V containing the linked vertices r_i and r_j. There are three possibilities:

1. No subset contains r_i or r_j. Create a new subset and add r_i, r_j into it.
2. One subset V_k containing r_i and/or r_j is found. Add these two vertices into V_k.
3. Two subsets V_k and V_m are found. One contains r_i while the other contains r_j. Combine the two subsets together.

Eventually, every subset's elements will be assigned a unique value, which is the tag of the subset.

Algorithm 2 Graph Partition

```
Input: Graph $E = \{(r_i, r_j)\}$, which $r_i$ and $r_j$ are both read ids.
Output: Disjoint subsets of vertices that indicate partitioned reads, $V$.
 1: $V \leftarrow \emptyset$
 2: for $(r_i, r_j) \in E$ do
 3:   $num\_found \leftarrow 0$
 4:   for $V_k \in V$ do
 5:     if $r_i \in V_k$ or $r_j \in V_k$ then
 6:       $found[num\_found] \leftarrow k$
 7:       $num\_found \leftarrow num\_found + 1$
 8:       if $num\_found > 2$ then
 9:         $break$
10:       end if
11:     end if
12:   end for
13:   if $num\_found = 0$ then
14:     $V_k \leftarrow \{r_i, r_j\}$
15:     $V \leftarrow V \cup V_k$
16:   else if $num\_found = 1$ then
17:     $k \leftarrow found[0]$
18:     $V_k \leftarrow V_k \cup \{r_i, r_j\}$
19:   else if $num\_found = 2$ then
20:     $k \leftarrow found[0], m \leftarrow found[1]$
21:     $V_n \leftarrow V_k \cup V_m$
22:     $V \leftarrow V - \{V_k, V_m\}$
23:     $V \leftarrow V \cup V_n$
24:   end if
25: end for
```

2.3 Reads Retrieving

In order to reduce the size of intermediate results, we only keep least information(read id) in the reads graph. So after graph partition phase, we should retrieve all the information of the original reads. That's the goal of reads retrieving. We chose to use a "join" operation, which is like the JOIN operation in SQL to link the subset tags and the original reads. Finally, we could output the partitioned reads into different files according to the subset tags.

3 Implementation

We implement the above reads partition algorithm with BioPig [5] , which is a Hadoop-based analytic toolkit for large-scale sequence data. BioPig provides programmability, scalability and some user defined functions(UDFs) for genomics data processing. We extend some of its functions and implement new UDFs we need.

Figure 7(a) shows the main software stack we use. Our implementation works with BioPig, which also works on Apache Pig[10]. Apache Pig is a platform for data flow processing that provides a high-level language called Pig Latin[6]. It works on Apache Hadoop through transforming the Pig Latin operations into a series of map-reduce tasks. The generated map-reduce tasks will be submitted to Hadoop cluster, which is responsible for scheduling these tasks and finishing the calculations. Apache Hadoop, which is upon the layer of Java runtime environment, contains three main modules: the distributed file system HDFS, the scheduling framework YARN and Map-reduce task framework. The Hadoop version we use is 2.7.3.

Pig Latin provides fundamental units (basic operations) to process interactively or construct streaming data processing scripts for structural and unstructured data. It has basic operations LOAD and STORE for data input and output respectively; it has relational operations like RANK, FILTER and ORDER as well as SQL-like operations like GROUP and JOIN. The data can be represented as relations. The term relations in Pig Latin is like variables in other common programming language and it represents data on which operators can take place during the processing period. Like a record in a database, every entry of the data could consist of multiple fields. For example, a relation user may consist of two fields: *userName* and *userId*.

BioPig provides sequence data loading and k-mer generating functions. The former supports reading reads data in fastq format while the latter supports generating k-mers from a read. For some other operations, like group, filter and join, there're corresponding native operations in Pig Latin.

In the reads graph generation phase, all the reads pairs need to be found for edge candidates. As k-mer shared information is regarded as edges' weight and only the edges with large weight value will be kept, we only consider the reads pairs with k-mer sharing. A UDF GenReadPair is designed. It consumes a tuple of read ids and outputs a data bag consisting of tuples, each of which is an ordered reads pair.

For graph partition, we use Google's connected components Map-Reduce algorithm [11] as the baseline. It is a very fast Map-Reduce algorithm that can easily scale to graphs with hundreds of billions of edges.

4 Optimization

4.1 Performance Analysis

We use two factors to evaluate performance: execution time and disk space usage. For disk space usage, we focus on */tmp* directory in HDFS. As Hadoop writes intermediate results into disks, this also reflects memory consuming of the application.

We use a 9-nodes cluster to deploy Hadoop with one master node and 8 slave nodes. The hardware and software settings of every node are showed in Table 1. There're two CPU sockets and 96GB memory on our nodes, every node is connected by 1Gb/s Ethernet connection. The operating system and other software are also listed in Table 1.

Table 1: Hardware and software setting

Item	Description
CPU	Intel Xeon E5-2660 @ 2.2GHz x2
Memory	DDR3 1333MHz 96GB
Hard Disk	SAS HDD 300GB
Network Connection	Intel Ethernet Adapter I350 with 1Gb/s, Mellanox QDR Infiniband 40Gb/s
Operating System	CentOS 7.3.1611, Linux 3.10.0
Java Runtime	JDK 1.8.0_20
Apache Hadoop	Apache Hadoop 2.7.3
Apache Pig	Apache Pig 0.15.0
MPI	Intel MPI 2017.0.098
Apache Tez	Apache Tez 0.7.0

The data sets we used vary in size from 1GB to 8GB in fastq format. They are Cow Rumen metagenome data sets provided by Joint Genome Institute[12] . The parameters were as follows: $k = 31$, $step = 2$, $min\_k\_appear = 2$, $max\_k\_appear = 2$, $min\_edge\_weight = 2$ and $max\_edge\_weight = 512$. We monitored the execution time of every phase and temporary directory size of graph generation phase. The results are displayed in Figure 2.

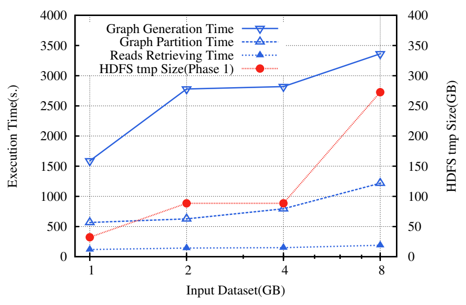


Fig. 2: Execution time of every phase and temporary folder size of phase 1

The results indicate that Graph Generation is the main hotspot of the algorithm. This is not surprising, since its processing is most complicated among all the three phases. This is why we only show its HDFS temporary folder size here. It is evident that the algorithm produces a large number of intermediate results—30-50x comparing to the input data size during Graph Generation phase. This makes it critical to control hard disk usage, for too much disk demand may stop us from processing larger data sets. The Graph Partition phase, as the figure shows, is also a hotspot. The good news is that even though it needs many iterations, it doesn't produce a lot of intermediate results. The Reads Retrieving seems not to be a bottleneck. This is because its logic is very simple compared to the other two.

4.2 Program Optimization

As we see from the results in section 5.1, our initial implementation consumes a large quantity of hard disk resources. It is essential to reduce memory footprint and hard disk usage for processing large amount of biological data. We implement two methods to achieve these goals: data type conversion and logic simplification.

Data Type Conversion The most common data type in sequence data processing is chararray in Pig Latin, which is actually an alias of string. That is really bloated compared to integer or long. K-mer is a sub-string of the initial input read, in which every base is a char. However, a char for a base is redundant, since every base could be one of only 5 different values (A, C, G, T or N for uncertain), so 3 bits instead of 8 bits are enough for a single base, or a byte(8 bits) can store at most 3 bases($5^3 = 125$ different cases). Therefore, compressing k-mer representation is a good way to reduce the size of intermediate results. This is already implemented in the k-mer generator in BioPig, but we use a more complete way.

In BioPig, every byte represents at most 3 bases. That is, $5^3 + 5^2 + 5 = 155$ different cases are stored in a byte. So only 155/256=60.55% of the data is true information. After optimization, we store every 3 bases in 7 bits, in that case, about $5^3/2^7 = 97.66\%$ is true information. We tested these two methods with data sets from 1GB to 8GB and compared their execution time of k-mer generator and the size of generated k-mer file. The results are showed in Figure 3. On average, 8.28% storage space is saved with only 6.89% run time increase. This is really cost-effective, since the extra time is negligible compared to the entire execution time while space saving is beneficial to memory, calculation, I/O and network communication. In the experiments, we use k=31. More space would be saved with a bigger k value, and the ideal case will be 12.5% with an infinite k.

In Figure 3, the execution time doesn't increase linearly in respect to input data size. This is because the Pig itself can arrange suitable number of map/reduce tasks according to data size and the k-mer generating is totally parallel without any communication. Bigger data sets result in more processing threads. That's why we achieve super-linear speedup here.

As mentioned above, chararray is the most common data type in reads processing. However, it requires not only more time to process but also more space to store. To overcome this obstacle, we use long integer to replace chararray types as much as possible. In Pig Latin, the operator RANK could assign a long value to every unique element in the specified data field with any type. Through this way, the field with any type will be converted to long, which is easier to compare, store or transmit. RANK is easy to implement, but also complicated

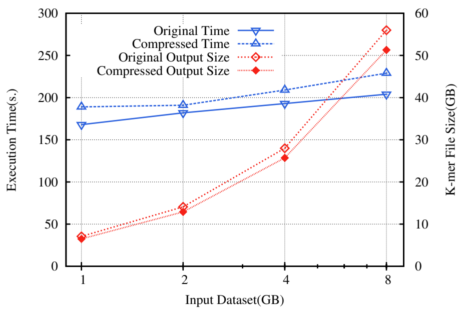


Fig. 3: Performance of k-mer generator

and slow. Since we only need a unique long value for every unique chararray, a good hash function is more suitable and surely faster.

We realized this data type conversion on almost all the chararray fields, such as *sampleid*, *readid* and *k-mer*. The extra hash operation requires more execution time during the data type conversion period, but this can accelerate other operations and reduce the size of intermediate results dramatically, which have much more positive effects on performance. Figure 4 shows the performance improvements after the data types are optimized. Comparing to the implementation without data type conversion($Default$ in the figure), data type conversion($Hashed$ in the figure) has advantages on both execution time and disk usage especially for large input data sets. For small input files, like 1GB to 8GB, data type conversion results in more execution time, because it caused more operations. But when the input data size grows further(16 GB in the figure), data conversion brings benefits to both execution time and hard disk usage. All the tests of data conversion show better hard disk usage. The larger the data set, the more significant the space saved. This makes it possible to process much larger data sets.

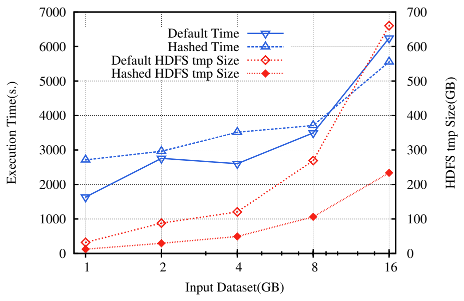


Fig. 4: Performance improvements of data type conversion

Logic simplification It is a common way to simplify the algorithm to obtain an approximate solution for acceleration when the algorithm is too complex or the input data set is too large to finish accurate calculation within tolerable execution time upon a limited platform. We also use this method to speed up our implementation.

According to running log, DISTINCT is one of the most time consuming operators, which removes duplicate tuples in a relation. It appears twice in reads graph generation phase, that is, one is to remove duplicate reads in the input data set and the other is to remove duplicate k-mers generated from the same read. We deleted both of them since we made sure every read id in input data set is unique and modified the k-mer generator to avoid generating duplicate k-mers from the same read.

Figure 5 shows an example of our k-mer generator modification. Let $k = 4$ and we focus on the k-mer $AACT$. It appears twice in $read0$ while once in $read1$. A typical k-mer generator generates "$AACT$ $read0$" twice so we have to remove the duplicate entries with DISTINCT operation. After modification, the new k-mer generator only produce one copy of "$AACT$ $read0$" and the time-consuming DISTINCT operation can be omitted.

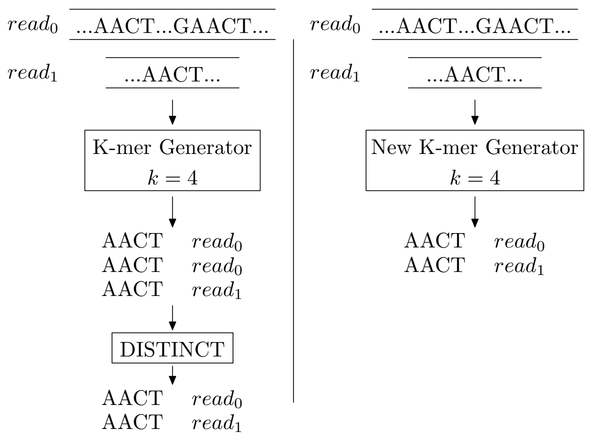


Fig. 5: An example of k-mer generator

In addition, there're two filters in the reads graph generation algorithm: remove all the k-mers that rarely or frequently appear inside every and among all samples. However, these filters are transformed into GROUP and COUNT operations in Pig Latin. As data size grows, the GROUP operation becomes very slow, since it requires sortation, comparison and communication among a lot of mappers and reducers. In order to accelerate these steps, we combined these two filters into one and introduced more stringent thresholds. The modification may affect the accuracy of the implementation, but it can provide sufficient results in a much faster way.

Figure 6 shows the performance improvements of logic simplification on different size of input data sets. The implementation with logic simplification is labeled with “Simplified Logic” while the other labeled with “Default Logic”. The previous optimizations have made it possible for larger data sets(32 GB in the figure) to be processed. In all these tests, up to 33.84% execution time is reduced through logic simplification.

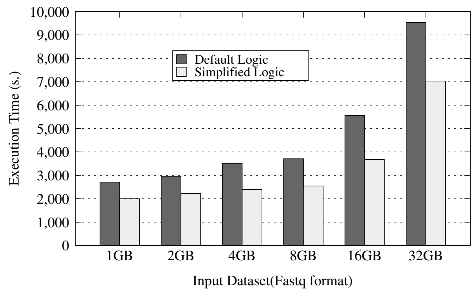


Fig. 6: Performance of logic simplification

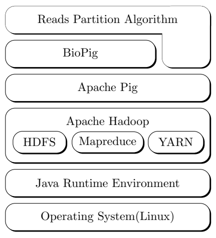
4.3 Graph Partition Redesign

Through performance analysis, we find that graph partition is one of the main hotspots. Since graph processing is an HPC problem which map-reduce is not really good at, finding another way to do this part of calculation is critical. HDFS provides basic API for C, we extended it and implemented the graph partition task in MPI. The implementation is based on the set union algorithm discussed in [13]. The optimized software stack is as Figure 7(b) shows. Now we have java runtime environment as well as MPI/C framework running on operating system. The latter uses HDFS as storage infrastructure and couple itself with Hadoop. Our new implementation of reads partition works on not only BioPig and Apache Pig, but also MPI/C framework.

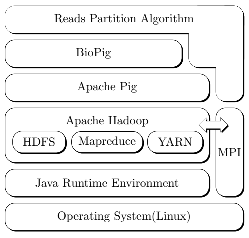
Figure 8(a) displays the MPI performance of graph partition and Figure 8(b) shows the runtime with respect to number of processes. The values in Figure 8(a) are presented in the logarithm of 2. In all the tests results presented, MPI version is 77-193x faster than Map-Reduce version. As the input data size grows, the gap between the two implementations is shrinking, but in the scale of our interest, MPI implementation is still much faster. The results in Figure 8(b) imply good scalability and larger graphs can take full advantage of more processes.

4.4 Software Stack and Hardware Optimization

As mentioned before, we use a complex software stack and program on a very high abstraction level. As a consequence, there’re much more factors having

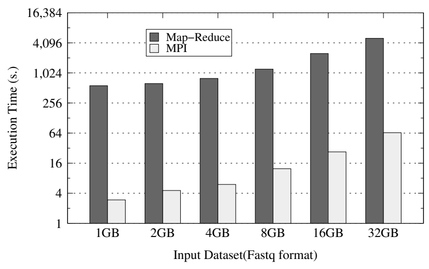


(a) The original software stack

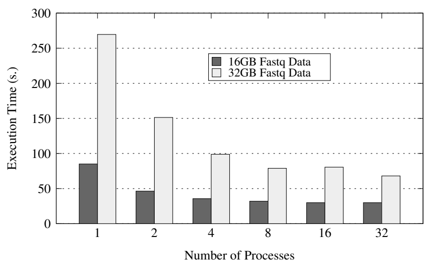


(b) The optimized software stack

Fig. 7: The software stack



(a) MPI and Map-Reduce Graph Partition performance



(b) Execution time of MPI Graph Partition in respect to number of processes

Fig. 8: MPI performance

effects on performance than that in traditional HPC applications. In other words, we have to consider every level of our software stack to gain a good performance. In this section we discuss some optimization attempts on Pig, Hadoop and even hardware.

Parameter Tuning The PARALLEL parameter in Pig script can control the number of reduce tasks for the MapReduce jobs generated by Pig at the operator level. In addition, through setting default_parallel in Pig script, the default number of reduce tasks of all generated MapReduce jobs at the script level can be specified. Pig sets the number of reduce tasks using a heuristic based on the input data size. Sometimes manually setting the parallelism according to the available resources could get better performance.

There're a series of configuration parameters that may affect a specific application upon hadoop. We mainly discuss options about block storage and data compression in this section. The block size parameter determines the size of every block stored in HDFS. It can affect Pig's performance since it directly determines the map parallelism – one map task for each HDFS block. Intermediate data compression is critical too since compressing the intermediate data during map and reduce phases can usually reduce the size of intermediate data significantly. Through this, not only disk space but also execution time will be saved.

Integrate Pig with Tez Apache Tez[14], which is an application framework built atop Apache Hadoop YARN, has two design themes that can release the performance of Map-Reduce applications: in-memory data processing and directed acyclic graph(DAG) tasks organization. In-memory data processing means that Tez can maximally remove unnecessary disk I/O and keep intermediate results inside memory as much as possible. DAG tasks organization means that Tez can reorganize all the Map-Reduce tasks and combine multiple tasks into one to reduce overheads.

Enhanced Network and Hard Disk Better hardware is also important to get good performance, network and hard disk are two candidates. Replacing Ethernet by faster Infiniband could reduce the data transferring time (usually in shuffle period) and using Solid State Disk(SSD) instead of Hard Disk Drive(HDD) could accelerate disk I/O. Even through the proportion of network data transferring and disk I/O will be weakened after some optimizations, they are still worthy of consideration.

Figure 9 shows the performance improvements achieved by software stack and hardware optimization. We see 6.3x and 1.54x performance benefits for 1GB and 32GB input data sets respectively and 4x speedup on average.

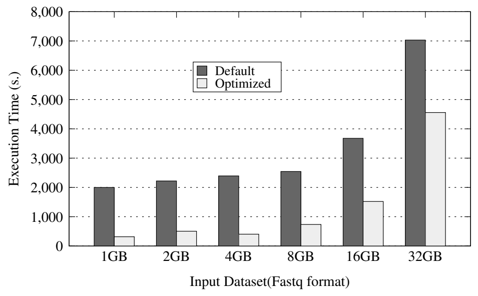
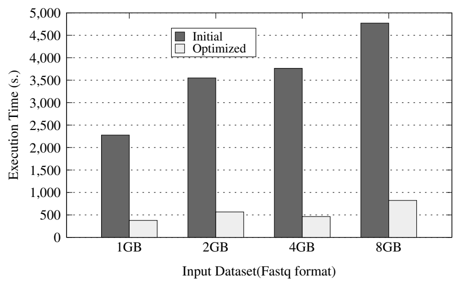


Fig. 9: The performance of software stack and hardware optimization

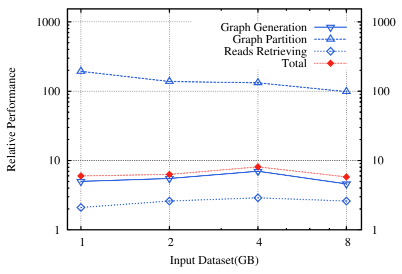
5 Results

Figure 10(a) compares the performance of optimized and original implementation. We witness 8.1x speedup at most and an average speedup of 6.6x. Figure

10(b) shows the performance improvement of every phase in respect to input data size. The y axis value is the logarithm of the base 10. It's clear that Graph Partition phase gets the highest speedup, which shows the affinity of MPI to the connected components calculation comparing to Map-Reduce. From Figure 10(b) we also see that the trend of speedups of the whole calculation is almost identical to that of Graph Generation phase. This is not surprising since after optimization, it turns out to be the most time-consuming phase and it dominates the whole execution time.



(a) The performance of optimized implementation



(b) Performance improvement of every calculation phase

Fig. 10: Performance improvements of optimizations

The optimization measures we take make it possible to process much bigger data sets on the same hardware platform. Figure 11 shows some big data sets we try on the specified platform. For 128GB data set, we use step=4 for k-mer generator to avoid intermediate k-mer data size exceeding the total disk space.

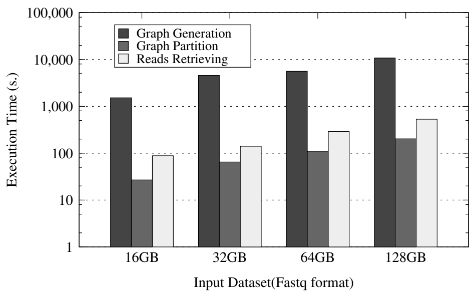


Fig. 11: Execution time for big data sets

The y axis value is the logarithm of the base 10. The Graph Generation still dominates the total execution time. Through all the optimization measures, the maximum data sets that could be processed has been enlarged 16 times from 8GB to 128GB upon the same hardware platform.

6 Discussion and Conclusion

The conventional big-data platforms, such as Hadoop[2], Spark[15], Apache Pig[10], can easily manage the large data sets with data-intensive operations, since they can provide fault-tolerant, distributed file system with resilience and high-throughput access. However, it's really hard for these platforms to deal with compute-intensive tasks, because their programming models are basically oriented to streaming or batch processing. On the contrary, the HPC programming models, like MPI[16], OpenMP[17] or CUDA [18] are primarily designed for compute-intensive processing. They have lower level programming APIs with much higher efficiency, but they don't possess fault-tolerant storage for big data sets. These features result in more programming work. Therefore, it is not appropriate to use HPC systems directly for big-data applications.

In this paper, we implement and optimize an algorithm for metegenomic reads partition. The problem is data-intensive(the reads graph generation phase) as well as compute-intensive(the graph partition phase), which has become a common feature of modern genomic problems[19]. We combine MPI with Hadoop to deal with the compute-intensive phase in the algorithm. This can fuse the advantages of the two platforms. Our fusion implementation is a preliminary attempt, but it has showed the great potential of the integration method to solve similar problems.

The optimization footprint uncovers a methodology for optimizing applications with high abstraction layers: first, tune the application; then optimize the system/software stack; at last, adjust the hardware configurations. We adopt some different optimization methods that are all very effective for our implementation. These measures, like data type conversion, data compression, specialized software stack configurations, and fusion of MPI with Hadoop, are also very effective in solving similar big-data problems.

There're some work about Pig or Hadoop tuning, such as[20] [21] [22] [23]. [20] presented a detailed step-by-step tuning process of K-mer counting on Hadoop. [21] and [22] tuned Hadoop performance at different software stack levels as well as hardware. [23] proposed an online performance tuning system which monitors a job's execution and tunes associated performance parameters. These works are to some degree similar to what we present in section 5.4.

The characteristics of HPC and big data frameworks and the potential benefits of integrating them were disscussed in prevoius works. [24] proposed a preliminary implementation of the high-performance big data system (HPBDS) called High-Performance Computing-Big Data Stack (HPC-ABDS). [25] analysed more applications and highlighted areas where HPC and Apache Big Data Stack have good opportunities for integration on the base of [24]. DataMPI [26] tried to extend MPI to support Hadoop-like Big Data Computing jobs. It showed performance and flexibility benefits while maintaining high productivity, scalability, and fault tolerance of Hadoop. [27] explored and compared two distributed computing frameworks on Google Cloud Platform: MPI/OpenMP and Apache Spark. The results showed that MPI/OpenMP outperforms Spark by more than one order of magnitude in terms of processing speed while Spark has advantages in some other aspects, such as data management infrastructure and fault tolerance. [28] implemented 3 matrix kernels on Spark and the comparisons with C+MPI implementations showed a performance gap of 10x - 40x without I/O. [29] proposed a system for integrating MPI with Spark and achieved 3.1-17.7x speedups on four graph and machine learning applications.

Some of the software stack tuning strategies should be tried at the beginning. By doing this, the execution time of single experiment could be reduced, meanwhile the optimization progress is accelerated.

References

1. Metzker, M.L.: Sequencing technologies—the next generation. Nature reviews genetics **11**(1) (2010) 31–46
2. Website: Apache hadoop. https://hadoop.apache.org
3. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al.: Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing, ACM (2013) 5
4. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1) (2008) 107–113
5. Nordberg, H., Bhatia, K., Wang, K., Wang, Z.: Biopig: a hadoop-based analytic toolkit for large-scale sequence data. Bioinformatics (2013) btt528
6. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM (2008) 1099–1110
7. Guo, X., Yu, N., Ding, X., Wang, J., Pan, Y.: Dime: A novel framework for de novo metagenomic sequence assembly. Journal of Computational Biology **22**(2) (2015) 159–177
8. Rasheed, Z., Rangwala, H.: A map-reduce framework for clustering metagenomes. In: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, IEEE (2013) 549–558
9. Schumacher, A., Pireddu, L., Niemenmaa, M., Kallio, A., Korpelainen, E., Zanetti, G., Heljanko, K.: Seqpig: simple and scalable scripting for large sequencing data sets in hadoop. Bioinformatics **30**(1) (2014) 119–120
10. Website: Apache pig. http://pig.apache.org
11. Kiveris, R., Lattanzi, S., Mirrokni, V., Rastogi, V., Vassilvitskii, S.: Connected components in mapreduce and beyond. In: Proceedings of the ACM Symposium on Cloud Computing, ACM (2014) 1–13
12. Hess, M., Sczyrba, A., Egan, R., Kim, T.W., Chokhawala, H., Schroth, G., Luo, S., Clark, D.S., Chen, F., Zhang, T., et al.: Metagenomic discovery of biomass-degrading genes and genomes from cow rumen. Science **331**(6016) (2011) 463–467

13. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM (JACM) **22**(2) (1975) 215–225
14. Website: Apache tez. https://tez.aprche.org
15. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. HotCloud **10**(10-10) (2010) 95
16. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the mpi message passing interface standard. Parallel computing **22**(6) (1996) 789–828
17. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering **5**(1) (1998) 46–55
18. Nvidia, C.: Compute unified device architecture programming guide. (2007)
19. Schmidt, B., Hildebrandt, A.: Next-generation sequencing: big data meets high performance computing. Drug Discovery Today (2017)
20. Shi, L., Wang, Z., Yu, W., Meng, X.: Performance evaluation and tuning of biopig for genomic analysis. In: Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems, ACM (2015) 9
21. Heger, D.: Hadoop performance tuning-a pragmatic & iterative approach. CMG Journal **4** (2013) 97–113
22. Joshi, S.B.: Apache hadoop performance-tuning methodologies and best practices. In: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, ACM (2012) 241–242
23. Li, M., Zeng, L., Meng, S., Tan, J., Zhang, L., Butt, A.R., Fuller, N.: Mronline: Mapreduce online performance tuning. In: Proceedings of the 23rd international symposium on High-performance parallel and distributed computing, ACM (2014) 165–176
24. Qiu, J., Jha, S., Luckow, A., Fox, G.C.: Towards hpc-abds: an initial high-performance big data stack. Building Robust Big Data Ecosystem ISO/IEC JTC 1 Study Group on Big Data (2014) 18–21
25. Fox, G.C., Qiu, J., Kamburugamuve, S., Jha, S., Luckow, A.: Hpc-abds high performance computing enhanced apache big data stack. In: Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, IEEE (2015) 1057–1066
26. Lu, X., Liang, F., Wang, B., Zha, L., Xu, Z.: Datampi: extending mpi to hadoop-like big data computing. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE (2014) 829–838
27. Reyes-Ortiz, J.L., Oneto, L., Anguita, D.: Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. Procedia Computer Science **53** (2015) 121–130
28. Gittens, A., Devarakonda, A., Racah, E., Ringenburg, M., Gerhardt, L., Kottalam, J., Liu, J., Maschhoff, K., Canon, S., Chhugani, J., et al.: Matrix factorization at scale: a comparison of scientific data analytics in spark and c+ mpi using three case studies. arXiv preprint arXiv:1607.01335 (2016)
29. Anderson, M., Smith, S., Sundaram, N., Capotă, M., Zhao, Z., Dulloor, S., Satish, N., Willke, T.L.: Bridging the gap between hpc and big data frameworks. Proceedings of the VLDB Endowment **10**(8) (2017) 901–912