Vol.2014-ARC-213 No.29
Vol.2014-HPC-147 No.29
2014/12/10

Performance Analysis of MapReduce Implementations for High Performance Homology Search (Unrefereed Workshop Manuscript)

Chaojie Zhang[1,a] Koichi Shirahata[1,2,b] Shuji Suzuki[1,c] Yutaka Akiyama[1,2,d]
Satoshi Matsuoka[1,2,e]

Abstract: Homology search to be used in emerging bioinformatics problems such as metagenomics is of increasing importance and challenge as its application area grows more broadly while the computational complexity is increasing, thus requiring massive parallel data processing. Earlier work by some of the authors have devised novel algorithms such as GHOSTX, but the master-worker parallelization to enumerate and schedule for data processing was done with a privately developed, MPI-based master-worker framework called GHOST-MP. An alternative is to utilize the now-popular big data software substrates, such as MapReduce with abundant associated software tool-chains, but it is not clear whether the massive resource required by metagenomic homology search would not overwhelm its known limitations. By converting the GHOST-MP master-worker data processing pipeline to accommodate MapReduce, and benchmarking them on a variety of high-performance MapReduce incarnations including Hadoop and Spark, we attempt to characterize the appropriateness of MapReduce as a generic framework for metagenomics that embody extremely resource consuming requirements for both compute and data.

1. Introduction

Homology search to be used in emerging bioinformatics problems such as metagenomics is of increasing importance and challenge as its application area grows more broadly while the computational complexity is increasing. One way to cope with the increasing complexity is to utilize massively parallel data processing. Required dataset for metagenomic search consists of queries and database, each of whose size will reach Gigabytes to Terabytes, and total data size to compute will grow to product of these two datasets (i.e. Exabytes to Zettabytes). BLAST [1], [2] is proposed as a basis of homology search algorithms and there have been a lot of efforts on improving the algorithm. Earlier work by some of the authors have devised novel algorithms such as GHOSTX [3] and extend the algorithm to distributed computing environments. Their work has demonstrated their implementation scales well on existing supercomputers including TSUBAME2.0 [4] and K computer [5], but the master-worker parallelization to enumerate and schedule for data processing was done with their privately developed MPI-based master-worker framework called GHOST-MP.

An alternative to using GHOSTX is to utilize the now-popular big data software substrates, such as MapReduce with abundant associated software tool-chains, but it is unclear how to apply MapReduce to extremely large-scale homology search in an efficient way. Firstly, It is not obvious how to design and implement homology search algorithms onto the MapReduce model. Specifically, how to handle two different dataset called queries and database which homology search algorithms receive using MapReduce is not straightforward. Secondly, performance characteristics of MapReduce-based implementations of homology search should be considered in order to achieve high performance homology search.

By converting the GHOSTX master-worker data processing pipeline to accommodate MapReduce, and benchmarking them on a variety of high performance MapReduce incarnations including Hadoop [6] and Spark [7], we attempt to characterize the appropriateness of MapReduce as a generic framework for metagenomics that embody extremely resource consuming requirements for both compute and data. We consider two different MapReduce-based designs of homology search considering data allocation of queries and database. Then we implement one of the designs onto Hadoop and Spark and conduct performance analysis on real world metagenomic dataset. We also compare our MapReduce-based implementations with GHOST-MP, an existing distributed implementation of GHOSTX on MPI-based master-worker framework.

Our preliminary experiments reveal that Hadoop and Spark perform comparable with original GHOSTX and GHOST-MP. The results show even faster using MapReduce in the case of

1 Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
2 JST CREST
a) zhang.c.am@m.titech.ac.jp
b) koichi-s@matsulab.is.titech.ac.jp
c) suzuki@bi.cs.titech.ac.jp
d) akiyama@cs.titech.ac.jp
e) matsu@is.titech.ac.jp

large query size. Spark with YARN performs 7.80x faster than the original GHOSTX and Hadoop is also 6.36x faster than the original GHOSTX. We consider the performance improvement of Hadoop and Spark comes from the difference of allocation of input query data. We can assign and compute multiple query files in parallel by splitting a query file into smaller splits for Hadoop and Spark, while we use a single query file for GHOSTX.

Here we describe a summary of contributions of our work:

- We describe MapReduce-based designs of homology search algorithms.
- We implement a homology search algorithm using multiple widely used MapReduce implementations.
- We show comparative performance analysis on multiple MapReduce implementations and a MPI-based homology search implementation.

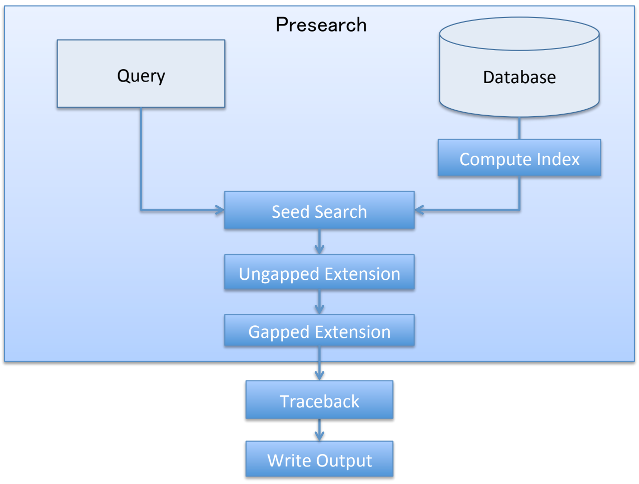

Fig. 1 Workflow of homology search

2. Background

We introduce overview of homology search and its existing algorithms. We explain required dataset, computational workflow, as well as fast algorithms of homology search. Then we also describe overview of MapReduce and its existing implementations.

2.1 Homology Search Algorithms

Homology search or alignment search is an approach to identify genes based upon homology with genes that are already publicly available in sequence databases by using a search algorithm. Homology search is used in the field of Metagenomics, the study of genetic material recovered directly from environmental samples for advancing knowledge in a wide variety of application domains, such as medicine, engineering, agriculture, ecology. Homology search algorithms are used as tools for life science researchers to gain a set of high-scoring pairs from an exhaustive list of protein coding sequences similar to a given query sequence, such as the amino-acid sequence of different proteins or the nucleotides of DNA sequences.

BLAST (Basic Local Alignment Search Tool) [1], [2] is proposed as a fast homology search algorithm and its implementation is widely used as a standard homology search tool. BLAST applies a heuristic algorithm much faster than previous approaches such as a full alignment procedure using the Smith-Waterman algorithm [8] or FASTA [9]. **Fig. 1** shows an overview of BLAST workflow. Firstly, BLAST finds seeds that are substring of database sequences similar to the substrings of a query sequence. Then, BLAST makes alignments by extending those seeds without gaps, and then similar, nearby seeds are brought together by a chain filter. Finally, BLAST makes alignments from seeds with gaps.

There have been a lot of efforts for improving BLAST [10], [11]. These efforts achieve speedup from the BLAST algorithm by improving search algorithms. Some of the authors also make efforts on accelerating BLAST. GHOSTX [3] adopts the seed-extend alignment algorithm used by BLAST. GHOSTX achieved approximately 131-165 times faster than BLAST. GHOSTX finds seed that are highly similar segments between database sequences and the query sequence. Next, GHOSTX obtain alignments by extending those seeds without gaps for larger similar regions. Finally, GHOSTX make alignments by extending the seeds with gaps. In order to accelerate the seed search process, GHOSTX constructs suffix array both for the query and the database before the search. In addition, instead of fixing the length of a seed like BLAST, GHOSTX extends it till the matching score exceeds a given threshold to reduce the computation time for untapped extension while not losing the sensitivity.

There exists also an extension of GHOSTX for distributed computing environments. GHOST-MP is built on GHOSTX with MPI library for homology search on supercomputers like K computer and TSUBAME, or general PC clusters. It achieves distributed paralleling search process through a master-worker style. In GHOST-MP's algorithm, it accomplishes I/O optimization for paralleled file system by utilizing locality of database chunks to achieve high speed processing.

2.2 MapReduce and Its Implementations

MapReduce is a programming model used for large data sets effectively through distributed algorithm across a cluster. MapReduce is composed of two major functions. The Map function takes in the input and emits key-value pairs that represent useful information from the input. These key-value pairs are later passes to reduce function to process the final results. The Reduce function produce zero or more outputs based on the values associated with each different key. An advantage of MapReduce is that it can handle large-scale data even when the data is larger than host memory capacity by handling memory overflow automatically. Another characteristic is that MapReduce can also handle compute node failures by applying techniques of fault tolerance. MapReduce is suitable for large-scale data processing and its implementations are widely used.

Hadoop [6] is a now-popular open-source software framework implemented in Java for storing and processing large data distributively on clusters. Hadoop is consisted of Hadoop Common, Hadoop Distributed File System (HDFS), Hadoop YARN, and Hadoop MapReduce. HDFS is a highly fault-tolerant distributed system, designed for applications with large data sets. Hadoop Yarn manages the compute resources in the file system and schedule jobs.

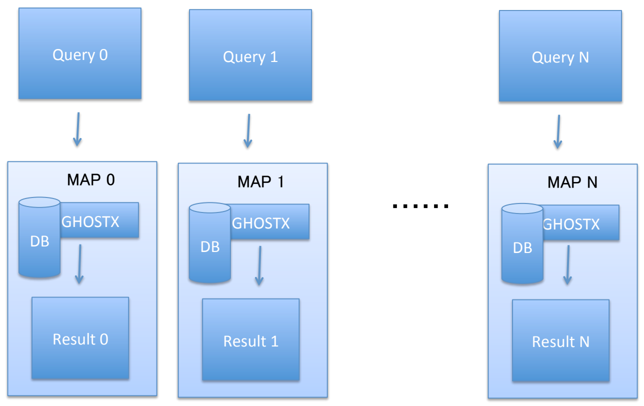

Fig. 2 Design of homology search with replicated database

Spark [7] is a fast open-resource cluster computing framework implemented in Scala, building on top of HDFS. Spark promises performance up to 100 times faster than Hadoop MapReduce in some certain applications. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements that can be persistent in memory and operated in parallel [12].

Some of the authors are also developing a MPI-based high performance MapReduce implementation running on either CPUs or GPUs [13], [14]. The implementation utilizes multiple GPUs on a large number of nodes and has demonstrated its scalability on the TSUBAME2.0 supercomputer. The implementation also handles memory overflow from GPUs by introducing chunk-based out-of-core GPU processing with overlapping of data transfers.

3. MapReduce-based Designs of Homology Search

We describe how to design homology search on MapReduce. Our main idea is to parallelize query data onto multiple Mappers. We consider two different designs based on how to assign query data and database onto worker nodes. On the two designs, query data is distributed onto the worker nodes on both designs while database allocation strategies are different. Note that we assume computing environments equip local disk on each compute node.

3.1 MapReduce-based Design with Database Replication

We describe a design of homology search on MapReduce using database replication. Query data is distributed on worker nodes while database is replicated among the worker nodes. **Fig. 2** describes how MapReduce works on the design. First, input query data files are copied to a distributed file system (e.g. HDFS) and the database file is replicated onto local disk on each compute node. After putting query and database, a client submits a job with a MapReduce application binary. A homology search application is called in map function of the MapReduce application. After submitting the application, each Mapper runs the homology search application with a split of query data and whole database for each map function the Mapper calls. A Mapper emits outputs of homology search for each query. Whole set of results from map functions is simply the final result.

This database replication design is useful when the size of database is small, since the result of each query is directly computed using whole database for each query. When the whole database can fit on local disk on each node, runtime can utilize locality of database. On the other hand, when the size of database is large, not only it may not fit on local disks but also parallelization efficiency may decrease because of the reduction in the locality of the database.

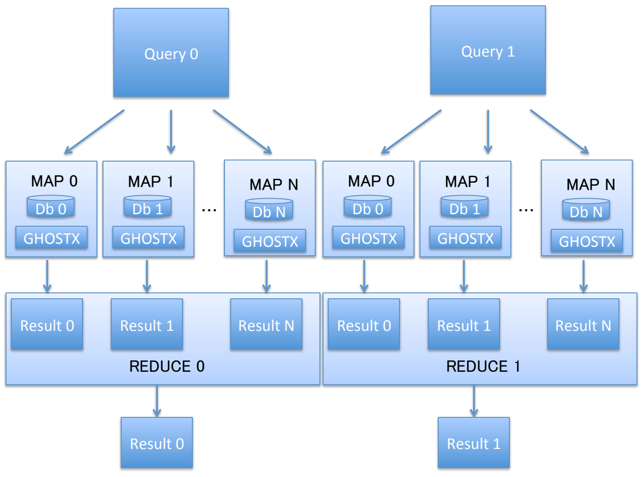

Fig. 3 Design of homology search with distributed database

3.2 MapReduce-based Design with Database Distribution

We consider another design that distributes database as well as query data. Query data is distributed on worker nodes and database is also distributed on the worker nodes. **Fig. 3** describes how the design works. First, input query data files are copied to distributed file system in the same way as the database replication design. Database is split to multiple chunks and each chunk is distributed on each node. These chunks can be also replicated to multiple nodes when the number of nodes is larger than the number of chunks. After putting query and database, a client submits a job with a MapReduce application binary. While a homology search application is called in each map function in similar way as the database replication design, result of each map function is different in that the result is a partial search result with a chunk of database. The results of Mappers are passed to Reducers and the Reducers merge the partial search results into a final search result for each query.

An advantage of this database distribution design is that the task granularity is smaller which can result in better parallelization efficiency. The number of tasks (i.e. the number of map function calls) with this database distribution design is larger than the database replication design since the database is divided to multiple chunks and each chunk is assigned to a Mapper. Having large number of tasks might not always be good; locality of database may become worse since each map function requires a specific chunk, which may result in multiple movements of chunks among worker nodes.

4. Implementations of Homology Search on MapReduce

We implement MapReduce-based homology search on existing multiple MapReduce implementations. We use GHOSTX as a sequential implementation and extend it onto the MapReduce

```
hadoop pipes\
  -D hadoop.pipes.java.recordreader=true\
  -D hadoop.pipes.java.recordwriter=true\
  -files [db_files]\
  -input [input_dir]\
  -output [output_dir]\
  -inputformat WholeFileInputFormat\
  -program ghostmr
```

Fig. 4 Calling GHOSTX from Hadoop Pipes. ghostmr is the compiled binary program incorporated original GHOSTX with a Hadoop Pipes application.

model. We implement the database replication design described in section 3.1 on Hadoop and Spark.

4.1 Implementation of Homology Search on Hadoop

In order to use GHOSTX on top of Hadoop, we need a way to call C++ from Java since GHOSTX is written in C++ while Hadoop is written in Java. There are several ways for calling GHOSTX from Hadoop, including Hadoop Pipes, Hadoop Streaming, and Java Native Interface. Hadoop Pipes is a library that allows C++ source code to be used for Mapper and Reducer code. Hadoop Pipes provides C++ API of map and reduce functions and users write the functions in C++ according to input and output formats provided by Hadoop. Hadoop Streaming is a more generic API that allows programs written in any language to be used as Mapper and Reducer implementations. While Hadoop Pipes and Hadoop Streaming are similar in that they split the application code into a separate process, they are different in that Hadoop Pipes uses serialization to covert the types into bytes that are sent to the process via socket, while Hadoop Streaming uses Unix standard streams as the interface. Java Native Interface (JNI) is a programming framework that enables Java code running in Java Virtual Machine (JVM) to call native applications and libraries written in other language such as C++. We select Hadoop Pipes since it provides closer interface with Java-based Mapper and Reducer. We modify the interface of original GHOSTX program so that Mapper can call GHOSTX program and setting query and database files through HDFS.

In order to distribute query and database files, we use different approaches for each dataset. As for query files, we use HDFS in a standard way for distributing multiple query files onto local disks on each node. We distribute the query files by the following command; `hdfs dfs -put [query_files] [input_dir]`. On the other hand, we do not distribute but copy the same database files onto each node since the database files are identical among all the nodes. To do this, we use `-files` option provided by Hadoop Pipes which copies specified files to cluster. As for query files, we need to avoid splitting them since the design of replicated database assigns one whole query file per Mapper, and Hadoop splits input data into lines and assign each line per map function by default. In order to disable splitting a query file into multiple splits, we implement WholeFileInputFormat for Hadoop Pipes based on [6]. We pass the customized input format to Hadoop Pipes by using `-inputformat` option. We run our GHOSTX on Hadoop by the following command described in **Fig. 4**.

```
spark-submit\
  --class "GhostMR"\
  --master yarn-client\
  --num-executors [num_nodes]\
  --executor-cores [num_threads]\
  --files [db_files]\
  --jars lib/hadoop-mapreduce-client-core-[ver].jar\
  ghostmr.jar
```

Fig. 5 Calling GHOSTX from Spark. ghostmr.jar is the compiled bytecode incorporated original GHOSTX with a Spark application.

4.2 Implementation of Homology Search on Spark

As with the case of Hadoop, we need a way for calling C++ from Scala since GHOSTX is written in C++ while Spark is written in Scala. Spark provides resilient distributed dataset (RDD) `pipe()` operation, which pipes each partition of RDD through a shell command in the same way as Unix pipe operation. RDD `pipe()` operation receives RDD input and sends output through Unix standard input and output. We apply GHOSTX to the `pipe()` operation, by simply executing GHOSTX binary program in `pipe()`.

In order to pass input files to Spark, we assign query files through HDFS and assign database files by copying to local disks on each node. In order to assign query files through HDFS to Spark, we put the query files to HDFS before running the application. We need to avoid splitting them since the Map-only design assigns one whole query file per Mapper as with the case of Hadoop described in section 4.1. In order to disable splitting a query file into multiple splits, we apply WholeFileInputFormat for Spark. We pass the customized input format to Spark by using `-jars` option with the jar file including WholeFileInputFormat. During running the application, it reads the query files from HDFS using `SparkContext.textFile()` method onto a RDD, then the RDD passes the query files to `pipe()`. As for database files, we copy them using `--files` option provided by Spark similar to Hadoop. **Fig. 5** describes the actual command for submitting GHOSTX on Spark.

5. Preliminary Experiments

In order to understand performance characteristics of MapReduce implementations, we conduct comparative performance experiments. We compare the elapsed time of homology search using existing MapReduce implementations as well as a MPI-based master worker implementation in order to investigate effectiveness of MapReduce-based implementation. We conduct data size scaling using different datasets as well as scaling of using multiple compute nodes. We use 1.1GB of query data named SRS014107 obtained from Data Analysis and Coordination Center for Human Microbiome Project website (`http://www.hmpdacc.org/`) [15]. We use 49MB of FASTA database named pdbaa obtained on November 4th, 2014 from The National Center for Biotechnology Information website (`http://www.ncbi.nlm.nih.gov/`) [16]. Note that we split input query files into 10MB of smaller files before putting them to HDFS for Hadoop and Spark, since we use WholeFileInputFormat as we described in section 4. Note that we do not include the elapsed time of database construction nor the time of data placement to

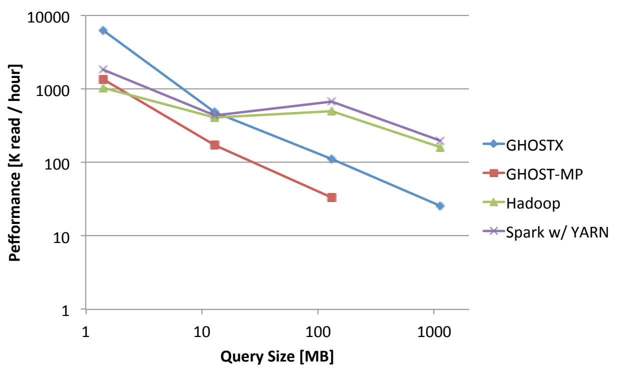

Fig. 6 Performance of data size scaling on single node

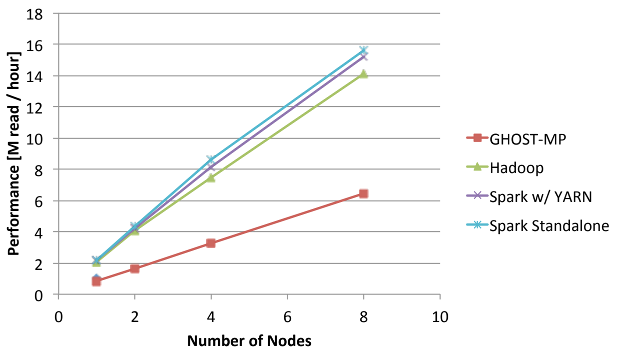

Fig. 8 Peformance of weak scaling with 13MB of query per node

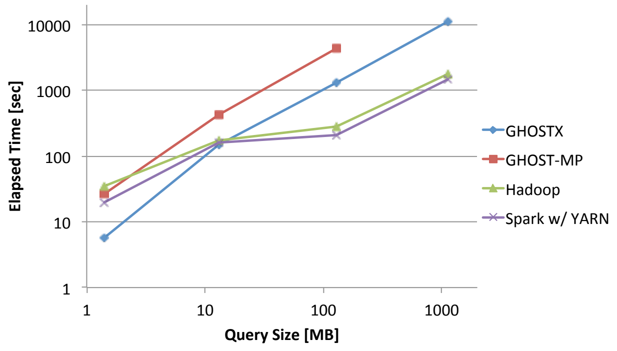

Fig. 7 Elapsed time of data size scaling on single node

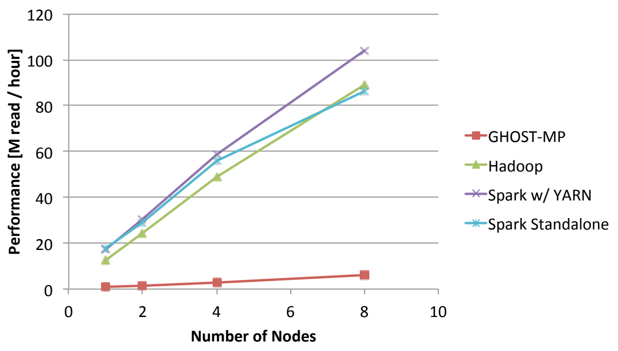

Fig. 9 Peformance of weak scaling with 130MB of query per node

local disk or HDFS.

We use our multiple compute nodes cluster, in which a machine equips one Intel(R) Core(TM) i7-3930K 3.20GHz (6 cores) CPU running in hyper-threading mode, 48GB of main memory, 102GB of a local SSD, and running Scientific Linux release 6.1. We use GCC 4.4.5 for the implementations. We use Hadoop version 2.4.1, Spark version 1.1.0, GHOSTX version 1.3.4, and GHOST-MP version 1.2.1. We use YARN scheduler on Hadoop, and use YARN and standalone schedulers on Spark. We use OpenMP for GHOSTX and GHOST-MP using 12 threads per node and use SSDs for placing query data and database as well as for writing output results. We build GHOST-MP with original configuration, without defining CHUNK and IOMASTER parameters. We use one worker process per node for GHOSTMP and set optional parameters to be equal to that of GHOSTX. We do not apply OpenMP parallelization for Hadoop and Spark.

5.1 Data Size Scaling

First we conduct data size scaling using single node with different datasets. We fix the size of database to 49MB, and use different sizes of query data. **Fig. 6** shows the performance results of data size scaling. X-axis indicates query data size and y-axis indicates thousands of query reads per hour. **Fig. 7** also shows the elapsed time of data size scaling. X-axis indicates query data size and y-axis indicates elapsed time of homology search. Note that we do not show the results of Spark with standalone scheduler since it performs similar to with YARN.

In the case of small query size, the results exhibit that the original GHOSTX performs the fastest. GHOSTX performs 6.11x faster than Hadoop and 3.45x faster than Spark with YARN using 1.4MB of query. We consider this performance difference derives from the overhead of MapReduce operations such as task invocation, task allocation, and redundant I/O operations. On the other hand, in the case of 1.1GB of query, Spark with YARN performs the fastest, 7.80x faster than the original GHOSTX. Hadoop is also 6.36x faster than the original GHOSTX. We consider a possibility of the faster performance on Hadoop and Spark comes from the difference of allocation of input query data. We use a single query file for GHOSTX, while we split the query file into smaller splits and place them on HDFS beforehand. By splitting the query file, we can assign and compute multiple query files in parallel. Although GHOSTX also computes in parallel by using OpenMP, we consider the reason of performance degradation derives from read and write operations are not parallelized. We will further investigate the cause of the performance difference as future work.

5.2 Weak Scaling

We also conduct weak scaling experiments using up to 8 nodes. We fix the size of database to 49MB. **Fig. 8** and **Fig. 9** shows the results of weak scaling using 13MB and 130MB of query size per node each other. X-axis indicates the number of nodes and y-axis indicates millions of query reads per hour.

The results indicate that all the implementations exhibit good scalability. We consider the results comes from the facts that homology search mainly consists of computational and I/O operations as well as the application includes little communication since computation of each query is independent of other queries. Another possible reason is that the implementations have little possibility to suffer load imbalance since workload we use is well balanced. The results also show that MapReduce-based implementations perform faster than GHOST-MP. Hadoop performs 2.18x and 15.0x faster and Spark performs 2.36x and 17.5x faster compared with GHOST-MP on 8 nodes using 13MB and 130MB of query each other. We consider the speedup derives from the performance degradation using large query size on GHOST-MP as shown in Fig. 6.

6. Related Work

MapReduce-based bioinformatics implementations have been studied [17], [18], [19], [20], [21], [22], [23], [24]. Their work indicate a wide range of applications using MapReduce related to bioinformatics as well as show high scalability on clusters and clouds using existing MapReduce implementations such as Hadoop. Their work focus on introducing algorithms or demonstrating scalability on relatively small number of nodes. However, our work focus on high performance and scalable homology search using MapReduce on large-scale computing environment such as supercomputers and analyze high performance MapReduce implementations.

K MapReduce (KMR) [25] is a MPI-based MapReduce implementation for large-scale supercomputers such as K computer. KMR optimizes shuffle operation by collective communication utilizing interconnect on K computer. Their work also conducted experiments using GHOST-MP by replacing master-worker tasking library in GHOST-MP with KMR. Although their work achieved high communication and I/O performance on K computer, they did not compare with other existing MapReduce implementation. We compare multiple MapReduce implementations and investigate high performance MapReduce-based homology search.

There have been efforts on MPI-based parallelization of bioinformatics applications. mpiBLAST [26] is a MPI-based parallelization of BLAST that achieves high scalability by optimizing allocation of database. mpiBLAST applys database segmentation which distributes a chunk of database to each node and let each node searches a unique portion of database. While mpiBLAST is high optimized for BLAST, our work focus on MapReduce-based high performance homology search since MapReduce is more widely used framework and can handle memory overflow and compute node failures.

7. Conclusion

In order to understand performance characteristics of MapReduce implementations, we present MapReduce-based designs and implementations of a homology search algorithm. We conduct comparative performance analysis of existing widely used MapReduce implementations as well as comparison with an existing MPI-based master-worker implementation of a homology search algorithm. Preliminary experiments reveal that Hadoop and Spark perform comparable with original GHOSTX and GHOST-MP, and even perform faster with large dataset.

Future work includes implementing other MapReduce-based homology search designs such as database distribution design as well as homology search on other MapReduce implementations such as our MPI-based MapReduce implementation. We also consider conducting further detailed performance analysis including using larger dataset on large-scale computing environments such as TSUBAME2.5.

Acknowledgments This research was supported by JST, CREST (Research Area: Advanced Core Technologies for Big Data Integration).

References

[1] Altschul, S. F., Gish, W., Miller, W., Myers, E. W. and Lipman, D. J.: Basic local alignment search tool, *Journal of molecular biology*, Vol. 215, No. 3, pp. 403–410 (1990).

[2] Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W. and Lipman, D. J.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *Nucleic acids research*, Vol. 25, No. 17, pp. 3389–3402 (1997).

[3] Suzuki, S., Kakuta, M., Ishida, T. and Akiyama, Y.: GHOSTX: An improved sequence homology search algorithm using a query suffix array and a database suffix array, *PloS one*, Vol. 9, No. 8, p. e103833 (2014).

[4] Matsuoka, S., Endo, T., Maruyama, N., Sato, H. and Takizawa, S.: The Total Picture of TSUBAME2.0, *Tsubame e-Science Journal*, Vol. 1, pp. 2 – 4 (2010).

[5] Yamamoto, K., Uno, A., Murai, H., Tsukamoto, T., Shoji, F., Matsui, S., Sekizawa, R., Sueyasu, F., Uchiyama, H., Okamoto, M. et al.: The K computer Operations: Experiences and Statistics, *Procedia Computer Science*, Vol. 29, pp. 576–585 (2014).

[6] White, T.: *Hadoop: the definitive guide: the definitive guide*, " O'Reilly Media, Inc." (2009).

[7] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I.: Spark: cluster computing with working sets, *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 10–10 (2010).

[8] Smith, T. and Waterman, M.: Identification of common molecular subsequences, *Journal of Molecular Biology*, Vol. 147, No. 1, pp. 195 – 197 (online), DOI: http://dx.doi.org/10.1016/0022-2836(81)90087-5 (1981).

[9] Lipman, D. J. and Pearson, W. R.: Rapid and sensitive protein similarity searches, *Science*, Vol. 227, No. 4693, pp. 1435–1441 (1985).

[10] Kent, W. J.: BLAT—the BLAST-like alignment tool, *Genome research*, Vol. 12, No. 4, pp. 656–664 (2002).

[11] Ma, B., Tromp, J. and Li, M.: PatternHunter: faster and more sensitive homology search, *Bioinformatics*, Vol. 18, No. 3, pp. 440–445 (2002).

[12] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. and Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, pp. 2–2 (2012).

[13] Shirahata, K., Sato, H., Suzumura, T. and Matsuoka, S.: A Scalable Implementation of a MapReduce-based Graph Processing Algorithm for Large-scale Heterogeneous Supercomputers, *Proceedings of the 2013 IEEE/ACM 13th International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '13, IEEE, pp. 277–284 (2013).

[14] Shirahata, K., Sato, H. and Matsuoka, S.: Out-of-core GPU Memory Management for MapReduce-based Large-scale Graph Processing, *Proceedings of the IEEE Cluster 2014*, IEEE, pp. 277–284 (2013).

[15] Turnbaugh, P. J., Ley, R. E., Hamady, M., Fraser-Liggett, C., Knight, R. and Gordon, J. I.: The human microbiome project: exploring the microbial part of ourselves in a changing world, *Nature*, Vol. 449, No. 7164, p. 804 (2007).

[16] Wheeler, D. L., Barrett, T., Benson, D. A., Bryant, S. H., Canese, K., Chetvernin, V., Church, D. M., DiCuccio, M., Edgar, R., Federhen, S. et al.: Database resources of the national center for biotechnology information, *Nucleic acids research*, Vol. 35, No. suppl 1, pp. D5–D12 (2007).

[17] Taylor, R. C.: An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics, *BMC bioinformatics*, Vol. 11, No. Suppl 12, p. S1 (2010).

[18] Gaggero, M., Leo, S., Manca, S., Santoni, F., Schiaratura, O., Zanetti, G., CRS, E. and Ricerche, S.: Parallelizing bioinformatics applications with MapReduce, *Cloud Computing and Its Applications*, pp. 22–23 (2008).

[19] Matsunaga, A., Tsugawa, M. and Fortes, J.: Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications, *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, IEEE, pp. 222–229 (2008).

[20] Meng, Z., Li, J., Zhou, Y., Liu, Q., Liu, Y. and Cao, W.: bCloudBLAST: An efficient mapreduce program for bioinformatics applica-

tions, *Biomedical Engineering and Informatics (BMEI), 2011 4th International Conference on*, Vol. 4, IEEE, pp. 2072–2076 (2011).

[21] Yang, X.-l., Liu, Y.-l., Yuan, C.-f. and Huang, Y.-h.: Parallelization of BLAST with MapReduce for long sequence alignment, *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, IEEE, pp. 241–246 (2011).

[22] Sunarso, F., Venugopal, S. and Lauro, F.: Scalable Protein Sequence Similarity Search using Locality-Sensitive Hashing and MapReduce, *arXiv preprint arXiv:1310.0883* (2013).

[23] Leo, S., Santoni, F. and Zanetti, G.: Biodoop: bioinformatics on hadoop, *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, IEEE, pp. 415–422 (2009).

[24] Sun, M., Zhou, X., Yang, F., Lu, K. and Dai, D.: Bwasw-Cloud: Efficient sequence alignment algorithm for two big data with MapReduce, *Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the*, IEEE, pp. 213–218 (2014).

[25] Matsuda, M., Maruyama, N. and Takizawa, S.: K MapReduce: A scalable tool for data-processing and search/ensemble applications on large-scale supercomputers, *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, IEEE, pp. 1–8 (2013).

[26] Darling, A., Carey, L. and Feng, W.-c.: The design, implementation, and evaluation of mpiBLAST, *Proceedings of ClusterWorld*, Vol. 2003 (2003).