Comparison of Distributed Data-Parallelization Patterns for Big Data Analysis: A Bioinformatics Case Study

Jianwu Wang, Daniel Crawl, Ilkay Altintas
San Diego Supercomputer Center
University of California, San Diego
La Jolla, U.S.A
{jianwu, crawl, altintas}@sdsc.edu

Kostas Tzoumas, Volker Markl
Technische Universität Berlin

Berlin, Germany
{kostas.tzoumas, volker.markl}@tu-berlin.de

ABSTRACT

As a distributed data-parallelization (DDP) pattern, MapReduce has been adopted by many new big data analysis tools to achieve good scalability and performance in Cluster or Cloud environments. This paper explores how two binary DDP patterns, i.e., CoGroup and Match, could also be used in these tools. We re-implemented an existing bioinformatics tool, called CloudBurst, with three different DDP pattern combinations. We identify two factors, namely, input data balancing and value sparseness, which could greatly affect the performances using different DDP patterns. Our experiments show: (*i*) a simple DDP pattern switch could speed up performance by almost two times; (*ii*) the identified factors can explain the differences well.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Distributed Applications*; D.2.11 [**Software Engineering**]: Software Architectures – *Domain-specific architectures.*

General Terms

Algorithms, Design, Experimentation, Performance, Verification

Keywords

Distributed Data-Parallelization; MapReduce; Performance Comparison

1. INTRODUCTION

Distributed Data-Parallelization (DDP) patterns [2], e.g., MapReduce [3], are reusable practices for efficient design and execution of big data analysis and analytics applications. Such DDP patterns combine data partition, parallel computing and distributed computing technologies. To date, many applications in scientific domains such as bioinformatics, ecology and geoinformatics utilized DDP patterns for large-scale datasets in distributed environments. For example, the CloudBurst tool [4] uses MapReduce to parallelize bioinformatics sequence mapping on multiple compute nodes for large datasets.

Over the last couple of years, DDP patterns other than MapReduce have also been proposed [6][7][10] to solve different types of big data analysis problems. Yet to the best of our knowledge, comparisons of different DDP patterns on performance when applied to the same tool and the main factors affecting such performance have not been studied. This paper explores different DDP options for such a scenario where different DDP patters can be used alternatively and switched for efficiency purposes. Our experiments show that performances of different DDP vary greatly for different inputs and run time parameter configurations. As a result of our study, we identify two main factors that affect the DDP pattern performance. We expect these results to help DDP researchers in finding the best DDP pattern and configurations for their own tools.

The contributions of this paper include: (*i*) Using an existing bioinformatics tool as an example, we demonstrate multiple feasible DDP options for many sequence mapping bioinformatics tools; (*ii*) We identify two key factors affecting the performances of different DDP options; (*iii*) We conduct experiments to demonstrate the feasibility of the identified factors and show that switching DDP option could speed up performance by over 1.8 times.

Outline. The rest of this paper is organized as follows. In Section 2, we discuss advantages of DDP patterns and explain four DDP patterns we use in this paper in detail. Section 3 describes how multiple DDP patterns could be used for many sequence mapping tools via a case study and identifies factors affecting the performances for different DDP pattern options. In Section 4, we describe the results of our experiments to show the performance differences for different DDP options and the feasibility of the identified factors in a Cluster environment. Section 5 discusses related work. In Section 6, we discuss our conclusions and plans for future work.

2. DDP PATTERNS

DDP patterns provide opportunities to easily build efficient big data applications, which can execute in parallel by splitting data in distributed computing environments. Such DDP patterns include *Map*, *Reduce*, *Match*, *CoGroup*, and *Cross* (a.k.a. *All-Pairs* [6][7]). Based on the functional programming paradigm, each DDP pattern represents a higher-level programming model to perform a certain function. Developers using these DDP patterns only need to focus on how their specific data process problems can be depicted in these programming models, and write corresponding user functions.

The definitions and key properties of the DDP patterns that are used in this paper are listed in Table 1. Map and Reduce process a single input, while CoGroup and Match process two inputs. For all patterns, each input data is a set of <*key*, *value*> pairs. Further formal definitions of these patterns can be found in [7].

There are an increasing number of execution engines that implement one or more DDP patterns described in Table 1. By taking care of underlying communications and data transfers, these DDP execution engines can execute user functions in parallel with good scalability and performance acceleration when running on distributed resources. In addition to the well-adopted MapReduce execution engine Hadoop[1], Cloud MapReduce[2] and MapReduce-MPI[3] are also available MapReduce execution engines. The Stratosphere system supports all the DDP patterns in Table 1 as part of their PACT programming model [7]. Meanwhile, CoGroup, Match and Cross could be transformed into Map and Reduce to run on Hadoop. For instance, the Cascading framework[4] supports dataflow paradigm using DDP patterns such as *Each* (similar to Map) and CoGroup and transforms these patterns into MapReduce to run on Hadoop. Our bioKepler project[5] [17] also supports CoGroup, Match and Cross on Hadoop by differentiating inputs [13].

Table 1. Definitions and key properties of the DDP patterns used in this paper

Map	• Independently processes each <*key, value*> pair. • User function can be executed in parallel based on input keys.
Reduce	• Partitions the <*key, value*> pairs based on their keys. • All pairs with the same key are processed together in one instance of the user function. • Executions for different keys can be done in parallel.
CoGroup	• Partitions the <*key, value*> pairs of the two input sets based on their keys. • For each input, all values with the same key form one subset. • Each user function instance gets three inputs comprised of a key and two value subsets for the key. • Executions for different keys can be done in parallel. • If a key is only at one input set, the value subset for this key from the other input will be empty.
Match	• Partitions the <*key, value*> pairs of the two input sets based on their keys. • Value pairs from the Cartesian product of the two value subsets for the same key are processed together. • Each value pair from the product and the key will be processed by an instance of the user function. • Executions for different inputs can be done in parallel. • If a key is only at one input set, no user function will be instantiated for this key.

3. A BIOINFORMATICS CASE STUDY: DDP OPTION COMPARISON FOR SEQUENCE MAPPING

In this section, we use a popular bioinformatics tool for sequence mapping, called CloudBurst [4], to demonstrate how different DDP pattern combinations could be used for the same tool and compare their performances. Original CloudBurst is implemented on top of Hadoop using Map and Reduce DDP patterns. Our re-implementations using Map, Reduce, CoGroup and Match are built on top of Stratosphere to utilize direct support for all of these four DDP patterns we used.

3.1 Sequence Mapping Bioinformatics Tools

Next-generation sequencing (NGS) technologies caused a huge increase in the amount of DNA sequence data being produced [1]. The sequences from NGS devices, called reads, are usually mapped to a reference sequence dataset in order to know whether there are similar fragments in reference data for each read and where the similar fragments are. The similarity threshold is defined by a parameter called mismatches (k), which is the maximum allowed length of differences. Many popular sequence mapping tools employ an algorithm called *seed-and-extend* [12]. This algorithm first finds sub-strings called seeds with exact matches in both the reads, or query sequences, and the reference sequences, and then extends the seeds into longer, inexact matches. The length of a seed is decided by parameter mismatches (k) and another parameter called reads length (m) with formula $m/(k+1)$. Although following the same algorithm, these tools use different methods for finding and extending seeds, resulting in different features and performances.

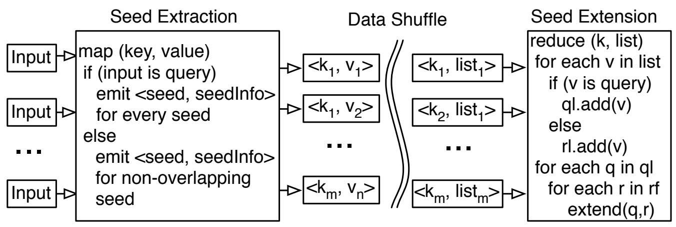

Figure 1. MapReduce Implementation of CloudBurst.

CloudBurst is a parallel seed-and-extend sequence mapping tool [4]. By adopting Map and Reduce patterns, CloudBurst has *Seed Extraction*, *Data Shuffle* and *Seed Extension* phases, as depicted in Fig. 1. The Seed Extraction phase uses the Map pattern to process both the query and reference sequences and emits seeds in parallel. Its output is a list of <*key*, *value*> pairs where *key* is the seed identifier and *value* contains detailed information of the seed, including its sequence string and offset in original sequence. The Data Shuffle phase groups seeds shared between the query and reference sequences. The Seed Extension phase extends the shared seeds by allowing mismatches. This phase uses the Reduce pattern so different seeds can be processed in parallel. In this phase, CloudBurst first splits the value list into two lists by checking whether it is query or reference. Next, it gets all pairs of elements from the Cartesian product of the two lists, and extends seeds for these pairs to generate mapping results. The scalability and performance speedup of CloudBurst in distributed environments have been described in [4]. Based on the results in [4], we picked CloudBurst as a good case study to try different DDP patterns and analyze their differences.

Please note that the query and reference datasets of CloudBurst have to be distinguished throughout the phases. Since Map and Reduce only support one input dataset, the query and reference datasets have to be differentiated in an indirect way when using Map and Reduce patterns. The details of this differentiation are outside the scope of this paper.

[1] Hadoop Project: http://hadoop.apache.org, 2013

[2] Cloud MapReduce: http://code.google.com/p/cloudmapreduce/, 2013

[3] MapReduceMPI: http://www.sandia.gov/~sjplimp/mapreduce.html, 2013

[4] Cascading Framework: http://www.cascading.org/, 2013

[5] bioKepler Project: http://www.biokepler.org/, 2013

3.2 Using CoGroup and Match Patterns for CloudBurst

We use the *MapCoGroup* and *MapMatch* patterns to re-implement CloudBurst, which are shown in Fig. 2 and Fig. 3 respectively. CoGroup and Match are natural DDPs to process two input datasets, and they both can have two Map functions for the Seed Extraction phase for query (*Qry* in the figures) and reference (*Ref* in the figures) data separately. For the Seed Extension phase, the differences of each pattern compared to the MapReduce implementation of CloudBurst are as follows:

- **CoGroup:** Each CoGroup user function instance gets a reference list and a query list for the same key (l^r and l^q in Fig. 2), so it needs one less loop compared to the MapReduce implementation of CloudBurst, i.e., it does not need the first loop in Fig. 1.
- **Match:** Because each Match user function instance directly gets one query value and one reference value for the same key (q and r in Fig. 3), it does not need any of the loops in the MapReduce implementation of CloudBurst.

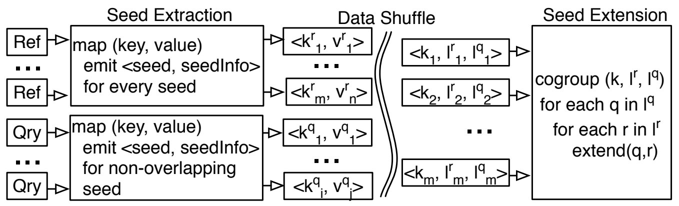

Figure 2. MapCoGroup Implementation of CloudBurst.

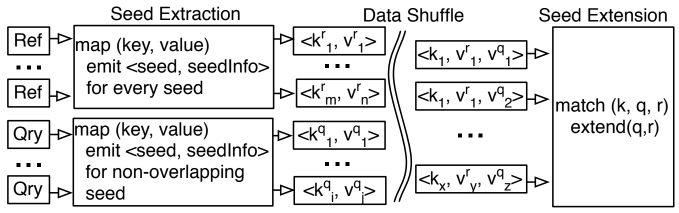

Figure 3. MapMatch Implementation of CloudBurst.

We also re-implemented CloudBurst in Stratosphere for MapReduce pattern by following the same logic in Fig. 1. From programming perspective, most of the original code of CloudBurst is reused in our new implementations for all three DDP pattern combinations. Some changes were made to comply with the Stratosphere APIs for the DDP patterns and input/output formats.

As mentioned in Section 3.1, many sequence mapping tools use a similar seed-and-extend logic with CloudBurst, so we believe the findings can be also applied for many other sequence mapping tools.

3.3 DDP Performance Comparison

For the abovementioned DDP options for CloudBurst, the Map parts are almost the same. The only difference is the Map function in MapReduce is split into two Map functions for MapCoGroup and MapMatch. Since the Map outputs of these DDP options are the same, the Data Shuffle phase also takes the same amount of time. So the main difference is in the way the map output data is read into and processed in Reduce/CoGroup/Match.

Total execution time of Reduce/CoGroup/Match includes two main parts: user function execution time and overhead to load the user function and input data. By looking into how Reduce pattern and Match pattern work, we identify two main factors that could affect their performances. The first factor is the difference between the numbers of keys in the two input data, denoted as p. The second factor is the average number of values per query/reference key, denoted as q.

The first factor, namely p, reflects the balance of the two input datasets. If one dataset is much larger than the other one, their key sets will have less common keys. Then Reduce pattern will have more user function executions with empty set for one input data, and these executions will not generate any results. But Match will not have user function executions for this situation. Imbalanced input datasets will cause a lot of unnecessary Reduce function execution. So Reduce is more suitable for balanced input datasets and Match is more suitable for imbalanced ones.

The second factor, namely q, reflects the sparseness of the values for each key. If each key has a lot of values, Match has to have a separate user function instance for each possible value pairs. While Reduce only needs one execution to process all values for the same key. At this situation, overhead of loading the user function and input data becomes dominant. So Reduce is more suitable for condensed values per key and Match is more suitable for sparse values per key.

For each user function execution, CoGroup takes less time than Reduce since it does not need the first loop in Seed Extension phase of Fig. 1, but takes more time than Match because it also have unnecessary executions with empty set from one input. Meanwhile, CoGroup's user function execution number is the same with Reduce's and less than Match's. So its total overhead is the same with that of Reduce and less than that of Match. Overall, CoGroup's total execution time should be in the middle of those of Reduce and Match.

4. EXPERIMENTS

We have done experiments by running CloudBurst using different patterns to answer two questions: (*i*) Would changing the DDP pattern to execute the same function have a big influence on the performance? (*ii*) Can the two factors identified in Section 3.3 adequately explain the performance differences?

We tested our implementations of CloudBurst in Stratosphere (version 0.2) with Reduce, CoGroup and Match in a compute Cluster environment. Five compute nodes are used in these experiments: one for master node and the other four for slave nodes. Each node has two four-core Intel 2.5GHz CPUs, 32GB memory, and runs Red Hat 5.9 Linux. Since our target is to compare performance differences with different DDP patterns, not to verify scalability, we only run the programs with a static physical environment.

In CloudBurst, both values of p and q will change accordingly when k value changes, and k value normally ranges from 0 to a small integer. So we tested different executions of the same program and parameters except the value of k. All experiments below are done with 12 parallel Map/Reduce/Match/CoGroup instances.

For all the experiments in this section, we repeat the execution for each configuration three times and find their execution times are similar. Thus, only their average values are listed for these execution times.

4.1 Execution Performance of CloudBurst using Different DDP Options

The first experiment is done with two real large datasets. The query dataset is publicly available sequencing reads from the 1000 Genomes Project[6] (access number: SRR001113), which has over nine million sequences and the sequence length is 47. The reference dataset is Monterey Bay metagenomic sequence data from CAMERA project[7], which has over 1.2 million sequences whose lengths range from 26 to 458. In this experiment, the Reduce/CoGroup/Match user function execution numbers range from three thousand to 11 billion. The executions of these different DDP patterns have the same results if their parameters are the same, but their performances are very different as seen in Table 2. The execution times for all three DDP combinations increase with different speeds when k value increases. The speedup of MapMatch over MapReduce reaches 1.890 (=2.786/1.474) for k=0. We will interpret the performance differences in the next sub-section.

Table 2. The execution times (unit: minute) of different DDP implementations of CloudBurst for large query and reference

Mismatch number (*k*)	**0**	**1**	**2**	**3**
MapReduce	2.786	3.405	3.537	8.622
MapCoGroup	1.564	1.916	2.477	24.640
MapMatch	1.474	1.883	2.689	47.393

The second experiment is done with only a large reference dataset. The reference dataset is the same as above and the query dataset only include the first 5000 sequences of the query dataset from the 1000 Genomes Project that is used in the last experiment. In the second experiment, the Reduce/CoGroup/Match user function execution numbers range from three thousand to seven million except the Match number for k=0 is zero since no seeds with the same key are found from both query and reference dataset. From Table 3, we can see the performances of MapMatch are always better than those of MapReduce.

Table 3. The execution times (unit: minute) of different DDP implementations of CloudBurst for only large reference

Mismatch number (*k*)	**0**	**1**	**2**	**3**
MapReduce	1.920	2.313	2.565	2.538
MapCoGroup	1.523	1.754	1.907	1.888
MapMatch	1.453	1.690	1.763	1.799

Table 2 and 3 also show MapCoGroup's execution times are always between those of MapReduce and MapMatch. It matches our analysis in Section 3.3. Based on the above experimental data, we can clearly see that different DDP patterns have great impact on the execution performance of CloudBurst. Implementations using Match and CoGroup have much better performances when parameter k's value is low. The experimental data also show no DDP pattern combination is always the best, even only for different configurations of the same tool. The execution performance varies based on input data and parameters. These initial experiments provide evidence that DDP pattern selection of the same tool could be very important for its performance.

[6] The 1000 Genomes Project: http://www.1000genomes.org/, 2013

[7] CAMERA Project: http://camera.calit2.net/, 2013

4.2 Effectiveness of Performance Factors

To explain the performance differences shown above and verify the effectiveness of the two factors identified in Section 3.3, we use log information get the values of p and q for each execution, and try to find the possible relationships between these factors and the performance differences. The data is shown in Table 4 and 5 along with speedup ratio of MapMatch to MapReduce. These tables show the values of the two factors greatly affect which DDP pattern has better performance. Both tables show most speedup ratios decrease along with the decrease of p values and the increase of q values. For k=3, it is interesting to see MapMatch is much slower than MapReduce for large query and reference, but a little faster for only large reference. We believe the reason is that q value decreases more dramatically than p value increases from the experiment with large query and reference to the experiment with only large reference (In Table 4 and 5, for k=3, p value difference between only large reference and large query and reference is 16 folds, while q value difference is 166 folds).

Table 4. The relationship between execution speedup and its factors for large query and reference

Mismatch number (*k*)	**0**	**1**	**2**	**3**
Key set size difference (p) (unit: million)	167	163	116	0.28
Average value number per key (q)	1.6E-5	1.6E-2	6.05E-1	2.73E3
Speedup ratio of Map-Match to MapReduce	1.890	1.704	1.201	0.181

Table 5. The relationship between execution speedup and its factors for only large reference

Mismatch number (*k*)	**0**	**1**	**2**	**3**
Key set size difference (p) (unit: million)	179	189	149	4.16
Average value number per key (q)	0	1.8E-5	4.6E-4	1.74
Speedup ratio of Map-Match to MapReduce	1.321	1.369	1.455	1.411

The relationships between the two factors with the performance differences shown from the above experimental data match our analysis in Section 3.3. So it verifies our analysis and feasibility of these factors. Please note that in these initial experiments the changes of k causes changes of both p and q. In the future, we will try to change one factor's value while keeping the other's value the same by deliberately manipulating input datasets.

5. RELATED WORK

There have been many sequence mapping tools using DDP patterns to process big sequence data in parallel, such as CloudBurst, CloudAligner [8] and CloudBLAST [9]. The differences of these tools are measured in terms of capabilities, accuracy, performance and so on. But they are all built on top of MapReduce pattern, and do not discuss whether other DDP patterns could also be used for their tools.

Other DDP patterns, such as All-Pairs [6], PACT [7] in Stratosphere, Sector/Sphere [10], are often compared with MapReduce when they are first proposed. Some studies, such as [6], show their applications in bioinformatics. But most of these comparisons are done using different DDP engines, so it is hard to know how different DDP patterns affect the same tool's performance on the same DDP engine. In [11], several applications were implemented and compared using different DDP patterns, but no experimental data on their performance differences were given.

There are also studies on performance modeling of DDP pattern. Paper [14] and [15] estimate MapReduce performance by mathematically modeling its phases using different sets of parameters such as Hadoop parameters, user input profiles and system-level parameters. Study in [16] identified five factors; such as I/O mode and indexing that affect Hadoop's performance. Our work is different from the above work from a few aspects: (*i*) These studies focus on system level factors, while we focus on application level factors; (*ii*) These studies only study MapReduce pattern, whereas we study performance comparison between DDP patterns; (*iii*) These studies are tested on Hadoop, and our experiments is based on Stratosphere DDP execution engine. Further, because of these differences, we think our work is complementary to these studies and can work together to analyze the performance of specific DDP applications.

6. CONCLUSIONS AND FUTURE WORK

In the "big data" era, it is very popular and effective to use DDP patterns in order to achieve scalability and parallelization. These DDP patterns also bring challenges on which pattern or pattern combination is the best for a certain tool. This paper demonstrates different DDP patterns could have a great impact on the performances of the same tool. We find that although MapReduce can be used for wider range of applications with either one or two input datasets, it is not always the best choice in terms of application complexity and performance. To understand the differences, we identified two affecting factors, namely input data balancing and value sparseness, on their performance differences. The feasibility of these two factors is verified through experiments. We believe many tools in bioinformatics and other domains have a similar logic with CloudBurst as they need to match two input datasets, and therefore could also benefit from our findings.

For future work, we plan to investigate more tools that are suitable for multiple DDP patterns and their performances on other DDP engines like Hadoop, which will generalize our findings. We will also study how to utilize the identified factors to automatically select the best DDP pattern combination from multiple available ones.

7. ACKNOWLEDGMENTS

This work was supported by NSF ABI Award DBI-1062565 for bioKepler. The authors would like to thank the rest of bioKepler and Stratosphere teams for their collaboration. We also thank the FutureGrid project for experiment environment support.

8. REFERENCES

[1] J. Shendure and H. Ji, "Next-generation DNA sequencing," Nature Biotechnology 26 (10) (2008) 1135–1145.

[2] J. Wang, D. Crawl, and I. Altintas. "A framework for distributed data-parallel execution in the Kepler scientific workflow system," Procedia Computer Science 9 (2012): 1620-1629.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Communications of the ACM 51 (1) (2008) 107–113.

[4] M. Schatz, "CloudBurst: Highly sensitive read mapping with MapReduce," Bioinformatics 25 (11) (2009) 1363–1369.

[5] A. D. Smith, Z. Xuan, and M. Q. Zhang, "Using quality scores and longer reads improves accuracy of Solexa read mapping," BMC Bioinformatics, 9, 128 (2008).

[6] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, D. Thain, "All-Pairs: An abstraction for data-intensive computing on campus Grids," IEEE Transactions on Parallel and Distributed Systems 21 (2010) 33–46.

[7] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, D. Warneke, "Nephele/PACTs: A programming model and execution framework for web-scale analytical processing," In: Proceedings of the 1st ACM symposium on Cloud computing, SoCC '10, ACM, New York, NY, USA, 2010, pp. 119–130.

[8] T. Nguyen, W. Shi, and D. Ruden, "CloudAligner: A fast and full-featured MapReduce based tool for sequence mapping," BMC research notes 4, no. 1 (2011): 171.

[9] A. Matsunaga, M. Tsugawa, and J. Fortes. "CloudBlast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications," In: Proceedings of IEEE 4th International Conference on eScience, eScience'08, 2008, pp. 222-229. IEEE, 2008.

[10] Y. Gu and R. Grossman, "Sector and Sphere: The design and implementation of a high performance data cloud," Philosophical Transactions of the Royal Society A 367 (1897) (2009) 2429–2445.

[11] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke, "MapReduce and PACT-comparing data parallel programming models," In : Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), GI, Bonn, Germany, pp. 25-44. 2011.

[12] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing." Briefings in bioinformatics 11.5 (2010): 473-483.

[13] M. Płóciennik, T. Żok, I. Altintas, J. Wang, D. Crawl, D. Abramson, F. Imbeaux, B. Guillerminet, M. Lopez-Caniego, I. C. Plasencia, W. Pych, P. Ciecieląg, B. Palak, M. Owsiak, Y. Frauel and ITM-TF contributors, "Approaches to Distributed Execution of Scientific Workflows in Kepler," accepted by Fundamenta Informaticae.

[14] H. Herodotou, "Hadoop Performance Models," Technical Report, CS-2011-05, Duke Computer Science, 2011, http://www.cs.duke.edu/starfish/files/hadoop-models.pdf.

[15] X. Lin, Z. Meng, C. Xu, and M. Wang. "A practical performance model for Hadoop MapReduce," In: Proceedings of Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on, pp. 231-239. IEEE, 2012.

[16] D. Jiang, B. C. Ooi, L. Shi and S. Wu, "The performance of mapreduce: An in-depth study," In: Proceedings of the VLDB Endowment 3, no. 1-2 (2010): 472-483.

[17] I. Altintas, J. Wang, D. Crawl, W. Li, "Challenges and approaches for distributed workflow-driven analysis of large-scale biological data", In: Proceedings of the Workshop on Data analytics in the Cloud at EDBT/ICDT 2012 Conference, DanaC2012, 2012.