PaPar: A Parallel Data Partitioning Framework for Big Data Applications

Hao Wang*, Jing Zhang*, Da Zhang, Sarunya Pumma, Wu-chun Feng
Department of Computer Science, Virginia Tech, Blacksburg, VA USA
Email: {hwang121, zjing14, daz3, sarunya, wfeng}@vt.edu

***Abstract*—Today, big data applications can generate large-scale data sets at an unprecedented rate; and scientists have turned to parallel and distributed systems for data analysis. Although many big data processing systems provide advanced mechanisms to partition data and tackle the computational skew, it is difficult to efficiently implement skew-resistant mechanisms, because the runtime of different partitions not only depends on input data size but also algorithms that will be applied on data. As a result, many research efforts have been undertaken to explore user-defined partitioning methods for different types of applications and algorithms. However, manually writing application-specific partitioning methods requires significant coding effort, and finding the optimal data partitioning strategy is particularly challenging even for developers that have mastered sufficient application knowledge.**

In this paper, we propose *PaPar*, a *Pa*rallel data *Par*titioning framework for big data applications, to simplify the implementations of data partitioning algorithms. PaPar provides a set of computational operators and distribution strategies for programmers to describe desired data partitioning methods. Taking an input data configuration file and a workflow configuration file as the input, PaPar can automatically generate the parallel partitioning codes by formalizing the user-defined workflow as a sequence of key-value operations and matrix-vector multiplications, and efficiently mapping to the parallel implementations with MPI and MapReduce. We apply our approach on two applications: muBLAST, a MPI implementation of BLAST algorithms for biological sequence search; and PowerLyra, a computation and partitioning method for skewed graphs. The experimental results show that compared to the partitioning methods of applications, the codes generated by PaPar can produce the same data partitions with comparable or less partitioning time.

***Keywords*-Partition; Skew; Big Data; MapReduce; MPI**

I. Introduction

In the past decade, big data processing systems have been gaining momentum; and scientists have turned to these systems to process large scale and unprecedented data. Most of these systems provide advanced mechanisms to tackle the load imbalance (a.k.a skew), which is a fundamental problem in parallel and distributed systems. For example, MapReduce [5] and its open source implementation Apache Hadoop provide the speculative scheduling to replicate last few tasks of a job on different compute nodes. Many mechanisms, including [34], [2], [18], [25], [9], [16], [3], [29], are also proposed to mitigate skew by optimizing task scheduling, data partitioning, job allocation, etc. Although these runtime methods are able to handle skew to a certain extent and do not require the code modification on applications, they can not get optimal application performance, because the runtime of application not only depends on input data size but also algorithms that will be applied on data. Therefore, many research efforts have been taken to explore the application-specific partitioning methods, including [1], [19], [17], [4], [37]. However, manually writing application-specific partitioning codes requires huge coding efforts. More challenging is the truth that finding the optimal data partitioning strategy is hard even for developers having adequate application knowledge, leading to the iterative and incremental development of design, evaluation, redesign, reevaluation, and so on.

In this paper, we target the complexity of developing application-specific data partitioning algorithms and propose PaPar, a parallel data partitioning framework for big data applications, to simplify their implementations. We identify a set of common functionalities used in data partitioning algorithms, e.g., sort, group, distribute, etc., and put them into PaPar as the building blocks of computational operators and distribution strategies. We provide a set of interfaces to construct the workflow of partitioning algorithms with these operators. PaPar can parse the configurations of input data types and workflow jobs, generate the parallel codes after formalizing the workflow as a sequence of key-value operations and matrix-vector multiplications. Finally, PaPar will map the workflow sequence to the parallel implementations with MPI and MapReduce.

In our evaluation, we use two applications as the case studies to show how to use PaPar to construct user-defined partitioning algorithms. The first driving application is muBLAST [35], a MPI implementation of BLAST algorithms for biological sequence search. The second is PowerLyra [4], a computation and partitioning method for skewed graphs. We conduct our experiments on a cluster with 16 compute nodes. The experimental results show that the code generated by PaPar can produce the same partitions as the applications but with less partitioning time. Compared to the multithreaded implementation of muBLASTP partitioning, PaPar can achieve up to 8.6-fold and 20.2-fold speedups for two widely used sequence databases. Compared to the parallel implementation of PowerLyra partitioning, PaPar can also deliver comparable performance for different input

*Hao Wang and Jing Zhang have contributed equally to this work.

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.119

graphs.

II. Background and Motivation

In this section, we first describe our driving applications, summarize the common functionalities needed in their data partitioning, and then discuss our motivation and design requirements.

A. Driving Applications

muBLASTP: BLAST is a fundamental Bioinformatics tool to find the similarity between sequences. muBLASTP is a MPI implementation of BLAST for protein sequences. By building the index for each database partition instead of input queries and optimizing the search algorithms with spatial blocking through memory hierarchy, muBLASTP can achieve better performance than the widely used BLAST implementations, e.g., mpiBLAST [20]. The performance of muBLASTP is sensitive to the partitioning methods: because of the nature of heuristics in the search algorithms, the runtime of sequence search depends on the distribution of sequence lengths more than the total size of each partition. The optimized partitioning method [36] tries to satisfy: (1) database partitions have similar numbers of sequences, (2) database sequences having the similar encoded length are distributed to different partitions, and (3) the sizes of encoded sequence data in partitions are similar. Figure 1 illustrates such an implementation. This method manipulates a four-tuple index that represents the encoded sequence pointer, the encoded sequence length, the description pointer, and the description length for each sequence. The partitioning method first sorts the index based on the encoded sequence length, and then distributes sequences to different partitions with a cyclic manner.

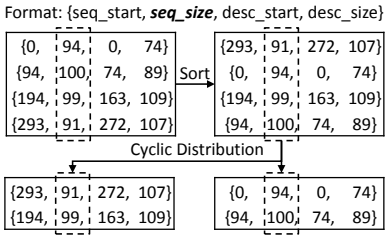


Figure 1: The partitioning method in muBLASTP: sort and distribute sequences based on the encoded sequence length.

PowerLyra: PowerLyra is a graph computation and partitioning engine on skew graphs. Other than the vertex-cut and edge-cut partitioning methods, PowerLyra provides a hybrid method to partition graph data. It first does the statistics to generate a user-defined factor, e.g., vertex indegree or outdegree, then splits vertices to a low-cut group and a high-cut group based on this factor, and applies different distribution policies on each group. Integrated with GraphLab [22], PowerLyra can bring significant performance benefits to many graph algorithms, e.g., PageRank, Connected Components, etc. Figure 2 from [4] shows this hybrid-cut method. In this case, PowerLyra uses the vertex indegree to divide the low-cut group and high-cut group with a predefined threshold. For the low-cut group, PowerLyra evenly assigns a vertex with all its edges (in-edges) to a partition; and for the high-cut group, PowerLyra distributes edges of each vertex to different partitions.

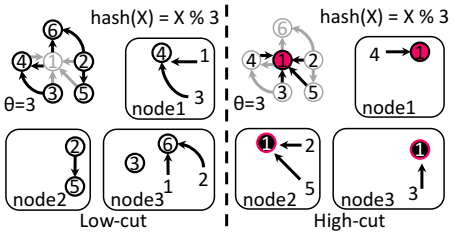


Figure 2: The hybrid-cut in PowerLyra: count vertex indegree, split vertices to the low-cut group and high-cut group; for the low-cut, distribute a vertex with all its in-edges to a partition, and for the high-cut, distribute edges of a vertex to different partitions.

B. Motivation

These driving applications illustrate the application-specific methods are necessary for better performance and scalability, even if the underlying systems provide the data partitioning methods. We also observe that there are several common functionalities used in these two applications to partition data, e.g., the sort operation: muBLASTP needs to sort sequences (as the value) by the encoded sequence length (as the key), and PowerLyra may group the edges belonging to a same in-vertex (as the value) and sort them by the vertex indegree (as the key). Therefore, our motivation is to design a framework to provide such common functionalities and simplify the implementations of application-specific partitioning algorithms. This task is not straightforward: even if the same functionality is needed, the requirements on these functionalities are quite different. For example, for the sort operation, the key and value in muBLASTP can be obtained from input data, i.e., the encoded sequence length and the sequence entry; while in PowerLyra, neither the key (the vertex indegree) nor the value (the grouped edges belonging to a same in-vertex) can be retrieved from input. As a result, the framework must have the capability to concatenate multiple operators, add/delete data attributes, and change data formats on demand. We summarize the design requirements:

Correctness: The framework needs to generate the user-defined partitioning codes. For the same input data, the partitions produced by the framework should be the same to those generated by the original partitioning algorithms.
Comprehensiveness: The framework needs to provide adequate building blocks to construct user-defined partitioning algorithms, and be able to extend more building blocks as well. Other than the key-value concept for unstructured data, the framework also needs to provide easy to

use interfaces to define multiple data types, considering many scientific applications manipulate structured and semi-structured data. Not only processing data from the input file, the framework also needs to support the in-memory data partitioning, because the intermediate data may need repartitioning and redistribution at runtime. **Efficiency:** The generated partitioning codes should be optimized to avoid data partitioning to be a performance bottleneck. Therefore, the framework needs to adopt the sophisticated techniques from recent research.

III. Methodology

Figure 3 shows the high-level architecture of our framework. The user interfaces are two configuration files. One is to describe the input data format, and the other is to describe the computational operations in the user-defined partitioning algorithm. By parsing the input data configuration and the workflow configuration, the framework can understand the data structure and set corresponding keys and values for each operation listed in the workflow. Users are allowed to register their own computational operator as a new building block by inheriting the *Operator* class and implementing the functionality, which will be discussed in Section III-B. The PaPar framework will generate the workflow which will be launched as a sequence of jobs at runtime.

A. Interface for Data Types

To read the structured data, MapReduce framework, e.g., Hadoop, provides a base class to unify the user interface: users need to implement their own parser for the input data structure by inheriting the Hadoop *InputFormat* class. In this class, users need to implement *getSplits* method to split the input file and generate a list of data blocks, each of which will be assigned to an individual mapper at runtime. Users also need to implement the *getRecordReader* method to extract individual input elements (records) from each split, and set the key and value for the mapper. Although many research projects [28], [14], [15], [31], [23], [32] have leveraged this mechanism to process structured data on MapReduce, and we also support this mechanism in PaPar, we prefer a programming-free method as the interface for user-defined data structures. We provide the *InputData* configuration file to allow users to describe their data structures.

Figure 4 shows the example how to describe the BLAST sequence index. The *input_format* and *start_position* sections indicate that BLAST sequence file is a binary file, and the index data starts at 32 bytes. The *element* section describes the index data structure consisting of four integers: *seq_start*, *seq_size*, *desc_start* and *desc_size*. According to the configuration file, the parser of PaPar will tell the *InputFormat* class to skip the first 32 byte of the file, and treat every 16 bytes (4 bytes/integer * 4 integers) as an entry. Figure 5 shows the example for the text format used in

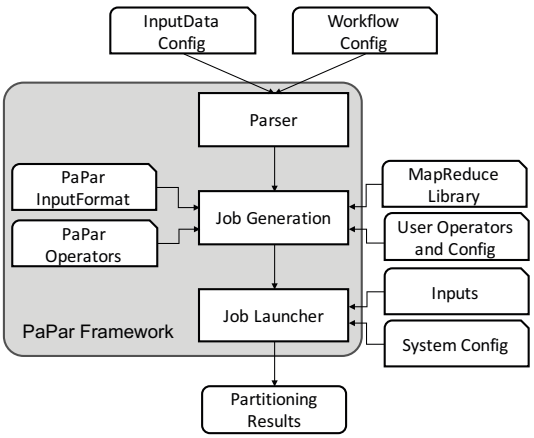


Figure 3: The high-level architecture of PaPar framework

PowerLyra. The *element* section indicates that each element represents an edge from *vertex_a* to *vertex_b*, separated by the *Tab* character "\t" and ended with the *Enter* character "\n". Similarly, the *InputFormat* class will treat each line in the text file as an entry, and fill two characters in each line to a two-tuple. Note that for derived data types, users may need to declare the nested elements in the configuration file. By providing such a configuration file as an interface, PaPar can support different input data types.

```
<input id="blast_db" name="BLAST Database file">
  <input_format>binary</input_format>
  <start_position>32</start_position>
  <element>
    <value name = "seq_start" type = "integer"/>
    <value name = "seq_size" type = "integer"/>
    <value name = "desc_start" type = "integer"/>
    <value name = "desc_size" type = "integer"/>
  </element>
</input>
```

Figure 4: Data type description for BLAST index

```
<input id="graph_edge" name="edge lists">
  <input_format>text</input_format>
  <element>
    <value name = "vertex_a" type = "String"/>
    <delimiter value="\t"/>
    <value name = "vertex_b" type = "String"/>
    <delimiter value="\n"/>
  </element>
</input>
```

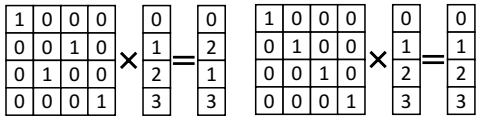
Figure 5: Data type description for graph data

B. Operators

We define a set of operators as the building blocks to implement the workflow of desired partitioning algorithms. Users can construct a workflow through the *Workflow* configuration file. For a data partitioning program, we observe that the input and output data formats are usually same, while the formats of intermediate data during partitioning may be different. For example, as discussed in Section II-B,

the PowerLyra hybrid-cut will count the vertex indegree, which is a new attribute. Based on the behaviors of operators on input data, we define three types of operators. First, the *Basic* operators, e.g., *sort*, *distribute*, *split*, *group*, etc., will reorder input data but not add or delete any attribute. For example, the sort operation will move entries from one compute node to another but keep data unchanged. Although multiple basic operators are usually concatenated to construct a workflow, a single basic operator can also be treated as a complete workflow. Second, the *Add-on* operators, e.g., *count*, *max*, *min*, *mean*, *sum*, etc., will add or delete data attributes. Different with the basic operators, the add-on operators themselves can not construct a workflow or a job in the workflow. They need to cooperate with the basic operators. Third, the *Format* operators, e.g., *orig*, *pack*, and *unpack*, can change the data format, but not reorder data or add/delete any attribute. Note that the input and output data discussed in this section refers to the input and output of an operator instead of the input and output files of a partitioning program.

Table I shows the details of the operators. Most of them will set one field of input data (or intermediate data) as the key and do the computation following the key-value concept. We will present more details with the driving applications in Section III-C. In this paragraph, we focus on the *policy* parameter used in *distribute*, which is an operator not following the key-value concept. In a partitioning algorithm, an entry from the input file is usually put into one partition of output. Although sometimes one entry may be put into multiple partitions for better performance or fault tolerance [8], we discuss the one-to-one mapping like the perfect hash in this paper. We design two basic types of policies, i.e., cyclic and block. The partitioning algorithms generated by PaPar will read the parameters *policy* and *numPartitions* from the configuration file at runtime, and formalize the policy to a matrix-vector multiplication operation. We borrow the idea of a domain-specific language (DSL) [7] to define a policy as a permutation matrix: L_m^{km}, $x_{ik+j} \mapsto x_{jm+i}, 0 \leqslant i < m, 0 \leqslant j < k$, which performs a stride-by-m permutation on a vector x having km items. In the distribution policy, x is the input data represented as a vector having km entries, and m is the stride to permute entries. Figure 6(a) illustrates the example to permute 4 entries with the stride 2 in the cyclic manner. The corresponding permutation matrix is L_2^4. Figure 6(b) illustrates the example for the block policy, which will not permute entries and the matrix is L_4^4. After the permutation, the contiguous data will be sent to 2 partitions for the distribution. The benefit of using the permutation matrix is to decouple the distribution policies from the workflow when PaPar generates the codes: at the time of code generation, it is not necessary to bind a distribution policy; and at runtime, the parameters *policy* and *numPartitions* will be processed and the permutation matrix will be generated, while the codes of the distribution operator are not changed. At runtime, the matrix-vector multiplication is enforced by multiple mappers in parallel and each mapper only processes its local data distribution based on the multiplication result.



(a) Cyclic matrix L_2^4 (b) Block matrix L_4^4

Figure 6: Formalize the distribution polices to matrix-vector multiplication

Though the operators listed in the table are sufficient for most cases, PaPar allows users to define their own operators. Users need to inherit one of these three operator classes, and provide a configuration file to describe the operator. Figure 7 shows an example of customized *sort*. The user needs to specify the class and argument types to tell the framework how to invoke it.

```
<prog id="Sort" type="operator"
      name="MapReduce sort operator">
  <import classpath="/user/mr/sort"
          package="com.mr.sort" class="Sort"/>
  <arguments>
    <param name="inputPath" type="String"/>
    <param name="outputPath" type="String"/>
    <param name="keyId" type="KeyId"/>
    <param name="ascending" type="boolean"
           default="true"/>
  </arguments>
</prog>
```

Figure 7: Configuration file for *sort* operator

C. Case Studies

To demonstrate the capability and usability of PaPar, we use muBLASTP and PowerLyra as the case studies.

muBLASTP: Figure 8 shows the workflow configuration file of muBLASTP partitioning. Three parameters listed in the *argument* section are the input file name, the output file name, and the number of partitions. Two operators *sort* and *distribute* are defined, each of which will be mapped to a job. We use the symbol *$* to represent the variable coming from intermediate data. For example, for the operator *distribute*, its input comes from the output of operator *sort*, which is labeled as "$sort.outputPath" in the configuration. The optional parameter *num_reducers* is used to launch reducers at runtime. Each operator can use a parameter defined in the workflow arguments or overwrite it in its own parameter section.

Figure 9 illustrates the workflow of muBLASTP partitioning, which sorts input by the encoded sequence length and then distributes elements evenly to multiple partitions with the cyclic policy. This figure follows the MapReduce style. In the figure, the left most part shows the index data of muBLASTP. Two jobs are launched in the workflow. The *sort* job sorts entries by using the sequence length *seq_size*.

Table I: Operators of PaPar workflow

Basic Operator
Sort(String inputPath, String outputPath, Class<?> inputFormat, Class<? extends Format> outputFormat, ValueId key, int flag, Class<? extends AddOn> addOn) Sort data with the given key. ***inputPath***: path of input. ***ouputPath***: path of output. ***inputFormat***: the format of input data. ***outputFormat***: the format of output data. ***key***: key to sort input data. ***flag***: pre-defined sorting type; -1: ascending, 1: descending. ***addOn***: add-ons.
Group(String inputPath, String outputPath, Class<?> inputFormat, Class<? extends Format> outputFormat, ValueId key, Class<? extends AddOn> addOn) Group data with the given key. ***inputPath***: path of input. ***ouputPath***: path of output. ***inputFormat***: the format of input data. ***outputFormat***: the format of output data. ***key***: key to group input data. ***addOn***: add-ons.
Split(String inputPath, List<String> outputPathList, Class<?> inputFormat, List<? extends Format> outputFormat, ValueId key, SplitPolicy policy, Class<? extends AddOn> addOn) Split data with the given split operation and key. ***inputPath***: path of the list of inputs. ***outputPathList***: file list for outputs. ***inputFormat***: the format of the input data. ***outputFormat***: the format of the output data. ***key***: key for splitting. ***policy***: the policy for splitting data. ***addOn***: add-ons.
Distribute(String inputPath, String outputPath, Class<?> inputFormat, Class<? extends Format> outputFormat, DistrPolicy policy, int numPartitions, Class<? extends AddOn> addOn) Distribute data with the given policy. ***inputPath***: path of input. ***ouputPath***: path of output. ***inputFormat***: the format of input data. ***outputFormat***: the format of output data. ***policy***: policy of distribution: cyclic and block. ***numPartitions***: number of partitions. ***addOn***: add-ons.
Add-on Operator
count(List<T> elements, ValueId key) Count the number of elements with the specific key.
max(List<T> elements, ValueId value) Get the maximum of the specific values of elements.
min(List<T> elements, ValueId value) Get the minimum of the specific values of elements.
mean(List<T> elements, ValueId value) Get the average of the specific values of elements.
sum(List<T> elements, ValueId value) Get the sum of the specific values of elements.
Format Operator
orig(List<T> keyVale) (default) Output data with the input format.
pack(List<T> keyVale) Output data with the packed format.
unpack(List<T> keyValue) Output data with the unpacked format.

The mappers will shuffle key-value pairs to different reducers according to the range of keys, which is sampled when reading the input. The data sampling will be discussed in Section III-D. In this case, as an example, the entries having the key *seq_size* ranging from 90 to 95 are assigned with the reduce-key "1", and then shuffled to the reducer "1". The reducers will sort entries by the key *seq_size* and write output data after removing the temporary reduce-key, because the basic operators will only reorder data but not change data as the definition.

The *distribute* job will distribute entries with the cyclic policy. The mappers will enforce the cyclic policy by applying the matrix-vector multiplication in parallel. In this case, each mapper knows there are 4 entries at local and 3 partitions for the output. Therefor, the permutation matrix L_3^4 is generated to permute the entires locally. After that, the mapper will distribute the entries to corresponding partitions. For example, the mapper "0" will send the entries "0" {566, 51, 490, 120}, "3" {1041, 79, 1107, 76} to the partition "0", the entry "1" {783, 64, 799, 91} to the partition "1", and so on. Because a reducer is launched to write data for a partition, the reducer id is used as the reduce-key. The reducers of the *distribute* job will write data to the output after removing the temporary reduce-key. Note that, some applications may need to adjust output data. For example, muBLASTP needs to recalculate the start pointers of sequence data and description data. This process has been implemented as a user-defined add-on operator. The algorithm recalculating the muBLASTP index has been discussed in [36], and we skip the details in this paper.

PowerLyra: As introduced in Section II-A, the hybrid-cut of PowerLyra will generate the new attributes and use them as the key and value of corresponding operators. Figure 10 shows the configuration file, which concatenates three basic operators *group*, *split*, and *distribute* in the workflow.

Figure 10 shows the details. The input data represents edges, i.e., *vertex_a* $\rightarrow$ *vertex_b*). The *group* job uses out-vertex *vertex_b* as the key to group edges in the map stage, and uses the add-on operator *count* to add a new attribute on each edge, i.e., vertex indegree, and uses the format operator *pack* to pack output data in the reduce stage. The *split* job then splits the packed entries based on the key *indegree*, which is the new attribute added by the add-on operator *count*. The *split* operator will send the entries which *indegree* are larger than or equal to *threshold*, i.e., 4 in this example, to the high-degree output, and others to the low-degree output. Note that, for the high-degree output, the format operator *unpack* is used to unpack data from the packed organization (as shown in the step 5 in the figure). The third job *distribute* will then operate on two different formats of intermediate

```
<workflow id="blast_partition"
          name="BLAST database partition">
  <arguments>
    <param name="input_path" type="hdfs"
           format="blast_db"/>
    <param name="output_path" type="hdfs"
           format="blast_db"/>
    <param name="num_partitions" type="integer"/>
    <param name="num_reducers" type="integer"
           value="3"/>
  </arguments>
  <operators>
    <operator id="sort" operator="Sort"
            num_reducers="$num_reducers">
      <param name="inputPath" type="String"
             value="$input_path"/>
      <param name="ouputPath" type="String"
             value="/user/sort_output"/>
      <param name="key" type="KeyId"
             value="seq_size"/>
    </operator>
    <operator id="distr" operator="Distribute">
      <param name="inputPath" type="String"
             value="$sort.ouputPath"/>
      <param name="outputPath" type="String"
             value="$output_path"/>
      <param name="distrPolicy" type="DistrPolicy"
             value="roundRobin"/>
      <param name="numPartitions" type="integer"
             value="$num_partitions"/>
    </operator>
  </operators>
</workflow>
```

Figure 8: Configuration file for muBLASTP

data and generate two permutation matrices, i.e., L_3^4 for the high-degree and L_3^3 for the low-degree. Note that L_3^3 in this case happens not to permute data, because there are 3 entries for 3 partitions. In a general case, L_N^M will enforce the cyclic distribution when M is larger than N. As the *distribute* is the last step in the workflow, all data will be unpacked to make sure the output has the same format of input.

D. Implementations

We map our framework on top of Apache Hadoop (2.7.0), MapReduce-MPI (abbr. MR-MPI) [24], and MPI. The interfaces of first two MapReduce systems are similar. On Hadoop, we implement the interfaces of processing structured data by inheriting *InputFormat* class. We implement those operators in Java, and generate Hadoop jobs for the workflow. On MR-MPI, an open-source C++ implementation of MapReduce on MPI, we use C++ to implement mappers and reducers by calling MR-MPI interfaces. The MR-MPI library can help us to hide the details of MPI based data shuffle and synchronization. On MPI, we currently use MPI non-blocking interfaces (Isend, Irecv, and Wait) to implement the data shuffle. During the execution of a PaPar-generated partitioner, the jobs are launched one by one following the order defined in the workflow configuration file. Several important techniques are also implemented as below:

Code Generation: We implement a parser to parse the configuration files and generate the Hadoop or MPI based partitioner by directly calling the backend implementations of operators. This method has been widely used in the code generation from a higher-level description to a lower-level implementation, e.g., from SQL to MapReduce jobs in Apache Hive [13], from SQL to GPU kernels [33], from DSL to SIMD implementations of sorting networks [12], etc. We plan to use an internal representation (IR) [6] to decouple the binding between the frontend and the backend in the future work.

Data Sampling: We implement the data sampling to balance the workload for the reduce stage. For example, for the *sort* operator, the temporary reduce-key corresponding to the range of input data is needed. In order to avoid the imbalance on reducers, we follow the mechanisms proposed in [9] to sample data on every node and approximate to the global data distribution. Based on the distribution of the user-set key and the number of reducers, we set the proper data range for each temporary reduce-key.

Data Compression: This optimization is used to compress the packed data. As shown in the hybrid-cut of PowerLyra, the *group* operator will call the *pack* operator to pack edges having the same in-vertex, resulting in the redundant data in this packed format. As shown in Figure 11, after the step 3, the reducer 0 has the packed data as {{2, 1, 4}, {3, 1, 4}, {4, 1, 4}, {5, 1, 4}}, and the redundant data is 1. This optimization uses the Compressed Sparse Row (CSR) and its transposition Compressed Sparse Column (CSC), which are widely used in sparse matrix computations [30], [21], [27], to compress data. In this case, the CSC format {0, {2, 3, 4, 5}, {4, 4, 4, 4}} is used: 0 is the start pointer of the in-vertex 1, the first vertex in the graph; {2, 3, 4, 5} is the out-vertex id array, and {4, 4, 4, 4} is the value array. Because the value array may include different values (depending on the algorithm to generate the attribute), we do not compress the value array to keep the generality. This optimization can improve the data communication performance, while it highly depends on the input data. We have observed up to 13% improvement for the graph datasets in our evaluation.

IV. Experiments

A. Experimental Setup

We conduct our evaluations on a homogeneous cluster consisting of 16 compute nodes. Each node has two 8-core Intel Xeon E5-2670 (Sandy Bridge) CPU running at 2.60 GHz, 64 GB memory, and 512 GB local disk. These nodes are linked by 10Gbps Ethernet and a Quad Data Rate (QDR) InfiniBand interconnect. Because both muBLASTP and PowerLyra are implemented in C++, we map PaPar on MR-MPI that leverages the MapReduce concept and the in-memory communication on MPI to provide comparable performance. All codes are compiled with MVAPICH2 library (version 2.2) and GCC 4.5.3. In all experiments, the execution time is the average time of five runs without I/O time.

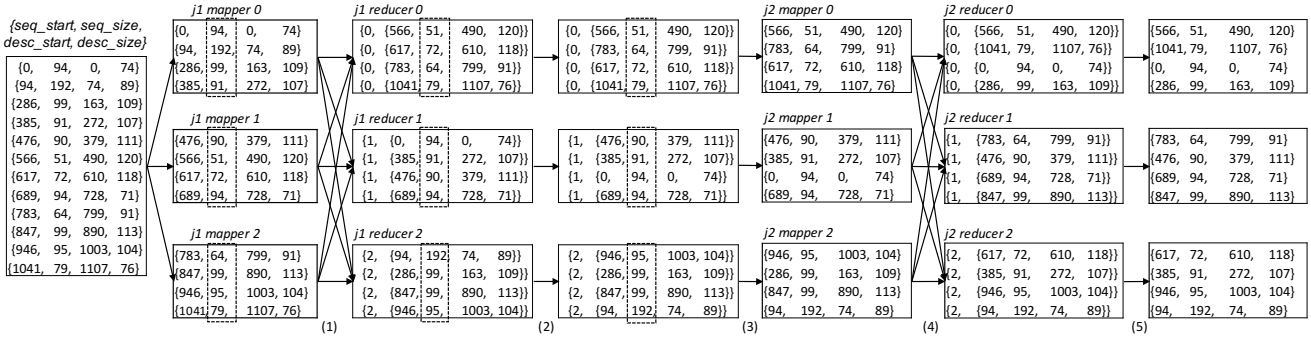


Figure 9: The workflow of muBLASTP data partitioning. The *Sort* job will sort the *index* elements by the user-defined key *seq_size* (in the dashed boxes), including: (1) mappers will shuffle data to reducers with the sampled reduce-key; (2) reducers will sort data by the key *seq_size*; (3) store data by removing the reduce-key. The *Distribute* job will distribute the sorted elements to partitions with the cyclic policy, including: (4) mappers will shuffle data to reducers with the generated reduce-key (reducer id); (5) remove the temporary reduce-key.

```
<workflow id="hybrid_cut" name="Hybrid-cut">
  <arguments>
    <param name="input_file" type="hdfs"
           format="graph_edge"/>
    <param name="output_path" type="hdfs"
           format="graph_edge"/>
    <param name="num_partitions" type="integer"/>
    <param name="threshold" type="integer"/>
  </arguments>
  <operators>
    <operator id="group" operator="group">
      <param name="inputPath" type="String"
             value="$input_file"/>
      <param name="outputPath" type="String"
             value="/tmp/group" format="pack"/>
      <param name="key" type="KeyId"
             value="vertex_b"/>
      <addon operator="count" key="vertex_b"
             attr="indegree"/>
    </operator>
    <operator id="split" operator="Split">
      <param name="inputPath" type="String"
             value="$sort.outputPath"/>
      <param name="outputPathList"
             type="StringList"
             value="/tmp/split/high_degree,
                    /tmp/split/low_degree"
             format="unpack,orig"/>
      <param name="key" type="KeyId"
             value="$group.$indegree"/>
      <param name="policy" type="SplitPolicy"
             value="{>=,$threshold},
                    {<,$threshold}"/>
    </operator>
    <operator id="distr" operator="Distribute">
      <param name="inputPath" type="String"
             value="/tmp/split/">
      <param name="outputPath" type="String"
             value="$output_path"/>
      <param name="policy" type="distrPolicy"
             value="graphVertexCut"/>
      <param name="numPartitions" type="integer"
             value="$num_partitions"/>
    </operator>
  </operators>
</workflow>
```

Figure 10: Configuration file for PowerLyra hybrid-cut

In the muBLASTP experiments, two partitioning methods are generated by PaPar. One is the default method to keep the number of sequences in partitions similar. We label it as "block". The other is the optimized method that will sort the index and distribute the sequences in a cyclic manner. We label it as "cyclic". We use two popular protein databases as the test datasets: *env_nr* database and *nr* database. The *env_nr* database consists of about 6,000,000 sequences with the total size at 1.7 GB, and the *nr* database has over 85,000,000 sequences with the size at 53 GB. Most of the sequences in two databases are less than 100 letters. We follow the experimental setups in [35] to randomly pick up sequences from corresponding databases to construct three batches, each of which includes 100 sequences. In the batch "100" and "500", all sequences are less than 100 and 500 letters, respectively; and for the "mixed" batch, we randomly select 100 sequences without the limitation of length.

In the PowerLyra experiments, we generate codes for three types of partitioning methods, "edge-cut", "vertex-cut", and "hybrid-cut" shown in Figure 2 . We choose PageRank as the test algorithm, which computes the rank of vertices in a graph. We use the snapshot version of PowerLyra with the tuned command line parameters downloaded from the PowerLyra website. The threshold parameter of hybrid-cut is set to 200 to divide the vertices into the low-cut or high-cut group. We choose three graph datasets: *Google*, *Pokec* and *LiveJournal*, from SNAP [26]. The datasets are stored in the *EdgeList* format as shown in Figure 5. Table II shows the statistics of these datasets.

Table II: Statistics of graph datasets

Graph	Vertices	Edges	Type	Triangles
Google	875713	5105039	Directed	13391903
Pokec	1632803	30622564	Directed	32557458
LiveJournal	4847571	68993773	Directed	177820130

In our evaluations, we first compare the partitions generated by PaPar and by the partitioning programs of driving applications. The results show that PaPar can produce the same partitions as the driving applications. After that, we present the performance numbers, including the execution time of applications with different partitioning algorithms,

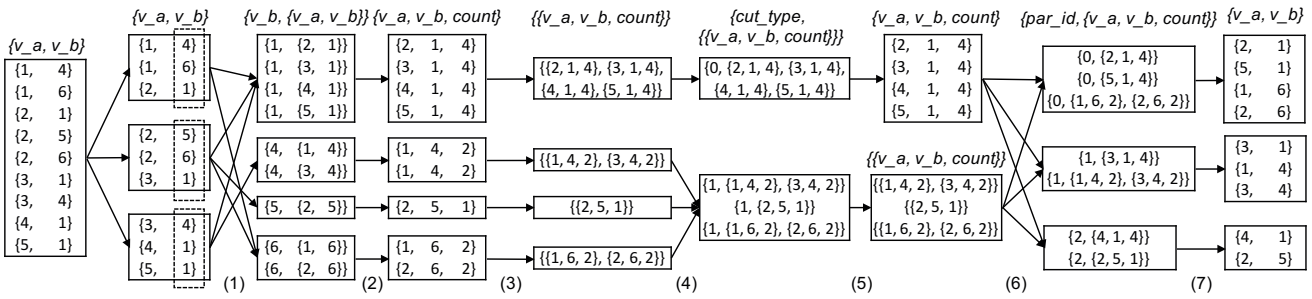


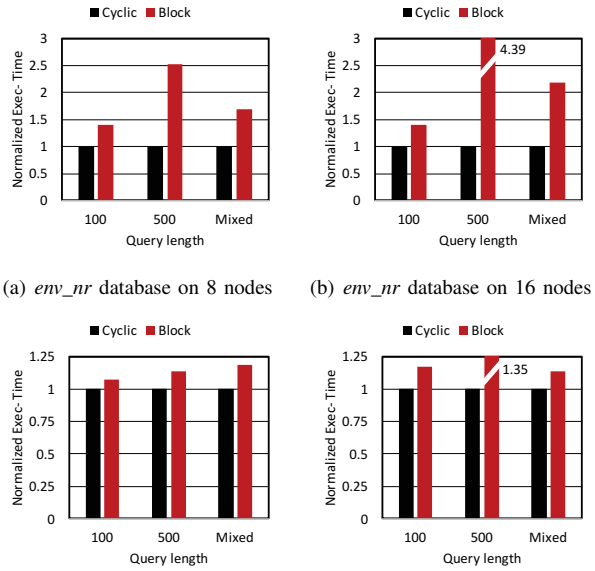
Figure 11: The workflow of PowerLyra hybrid-cut algorithm. The *Group* job will group the edges by in-vertex, including: (1) mappers will shuffle data to reducers by setting the in-vertex id as the reduce-key; (2) the add-on operator *count* will add a new attribute indegree for each edge; (3) the format operator *pack* will change the output format to the packed one. The *Split* job will split data into two groups, including: (4) based on the split condition in the configuration file (indegree is larger than or equal to 4 in this case), mappers will set the reducer id as the temporary reduce-key and shuffle data to reducers; (5) based on the different formats of output files, the *unpack* operator is applied on the high-degree part to unpack the data format. The *Distribute* job will distribute the entries in the cyclic manner, including: (6) mappers will shuffle data to reducers by setting the reducer id as the reduce-key; (7) reducers will remove the temporary reduce-key.

the partitioning time on the given input data sets, and the scalability on multiple compute nodes.

B. Evaluation of BLAST Database Partitioning

Figure 12 shows the normalized execution time of muBLASTP search for three batches on 8 and 16 compute nodes with the cyclic and block polices. muBLASTP follows the MPI + OpenMP programming model, and the best performance can be achieved when binding a MPI process to one CPU (socket) and launch multiple OpenMP threads (8 on our Intel Sandy Bridge CPU) in one MPI process. As a result, on 8 nodes, we produce 16 (8 * 2) partitions; and on 16 nodes, the partition number is 32 (16 * 2). In these figures, the cyclic policy is the clear winner that can bring obvious performance benefits to muBLASTP, no matter which combination of database and batch is used. We also observe that the cyclic policy can achieve more performance benefits for the larger batch, i.e. the batch "500". That means the skew is more significant for the longer queries because they have relatively longer search time.

Because the cyclic policy can deliver better performance to muBLASTP search, we compare the partitioning time of PaPar and default muBLASTP partitioning for this policy. Figure 13(a) shows the normalized partitioning time on 16 nodes for the *env_nr* and *nr* databases, respectively. Because the current implementation of muBLASTP partitioning only provides a multithreaded method for the input database [35], it can not scale out on 16 nodes. On the contrary, PaPar can map to MapReduce and MPI implementations, and scale on multiple compute nodes. As shown in the figure, PaPar can achieve 8.6x and 20.2x speedups over default muBLASTP partitioning on 16 nodes for two databases, respectively. Note that even on a single compute node, PaPar is faster, thanks to ASPaS [12], a highly optimized mergesort implementation on multicore processors. We used it in the sort operator implementation. Figure 13(b) shows the scalability up to 16 nodes. Compared to its own single node



(a) *env_nr* database on 8 nodes (b) *env_nr* database on 16 nodes

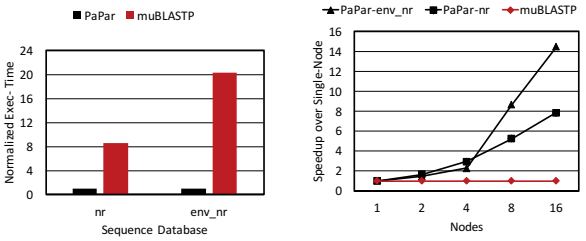
(c) *nr* database on 8 nodes (d) *nr* database on 16 node

Figure 12: Normalized execution time of muBLASTP with the cyclic partitioning and block partitioning (normalized to cyclic) on *env_nr* and *nr* databases.

implementation, PaPar can obtain 7.9x and 14.3x speedups for the *nr* and *evn_nr* databases, respectively.

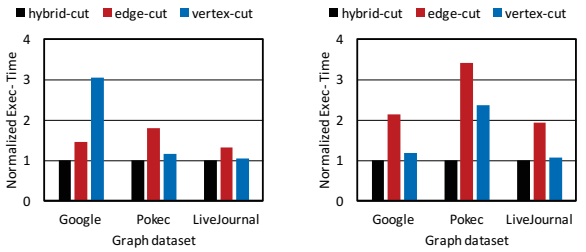
C. Evaluation of Hybrid-Cut Graph Partitioning

Figure 14 shows the normalized execution time of PageRank with "hybrid-cut", "edge-cut", and "vertex-cut" on 8 and 16 nodes. The hybrid-cut can deliver the best performance as we expected. The vertex-cut distributes a vertex with all its in-edges to a partition, which favors the vertices having low-degrees. Because the three datasets in our experiments follow the power law distribution that have much more low-degree vertices, the vertex-cut, instead of the edge-cut, has the closer performance to the hybrid-cut.



(a) Partitioning time on 16 nodes (b) Scalability (up to 16 nodes)

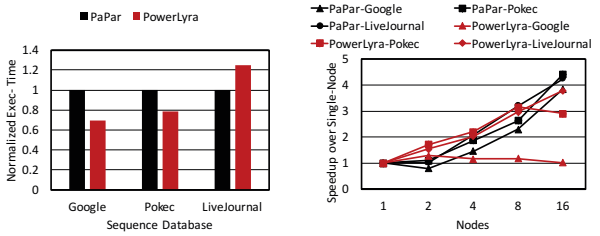
Figure 13: Partitioning time (cyclic) for *env_nr* and *nr* databases, and strong scalability of codes generated by PaPar, compared to muBLASTP partitioning program.



(a) PageRank running on 8 nodes (b) PageRank running on 16 nodes

Figure 14: Normalized execution time of PageRank (with PowerLyra) for hybrid-cut, edge-cut, and vertex-cut (normalized to hybrid-cut).

Figure 15(a) shows the normalized partitioning time of PaPar codes and PowerLyra on 16 nodes for the hybrid-cut. On the Google and Pokec datasets, PowerLyra has the better performance; while PaPar can deliver 1.2x speedup on the LiveJournal dataset. There are several reasons leading to the variable performance comparison. PaPar is mapped on MR-MPI to balance the programmability and performance but without those optimizations on multicore processors used by PowerLyra, e.g., the NUMA-aware data access. Therefore, PowerLyra is faster for the small and medium datasets, where the single node performance counts more. However, such a benefit is offset in the communication intensive case on multiple nodes. Although PowerLyra is integrated with GraphLab on top of MPI, its data shuffle is still based on the socket communication on Ethernet. On the contrary, PaPar maps to MR-MPI that uses MPI instead of socket communication. In our experiments, the MVAPICH2 library can use Remote Direct Memory Access (RDMA) communication on InfiniBand to improve the performance. Furthermore, PowerLyra uses the dynamic approach that calculates scores for low-degree vertices in each partition. This method introduces additional overhead, especially for graphs which vertices cluster together, e.g., the LiveJournal dataset. Figure 15(b) also demonstrates the variable performance. PowerLyra can scale up to 8 and 16 nodes for the Pokec and LiveJournal datasets, respectively, but cannot scale on multiple nodes for the Google dataset; while, PaPar can scale up to 16 nodes for all three datasets.



(a) Partitioning time on 16 nodes (b) Scalability (up to 16 nodes)

Figure 15: Partitioning time (hybrid-cut) and strong scalability of codes generated by PaPar framework, compared to PowerLyra.

V. Related Work

Over the past few years, many efforts have been taken to explore the skew problem. The speculative scheduling is the basic method of MapReduce that can speculatively relaunch last few tasks on other nodes. Late [34] speculatively launches tasks having the longest estimation remaining time. Mantri [2] restarts the task having inconsistent runtime. Flexslot [10], [11] dynamically changes the numbers of slots for stragglers. These methods require rerun the whole task or subtasks, assuming the skew comes from the heterogeneous hardware malfunction or resource contention.

The dynamic methods usually use data repartitioning and migration to resolve the skew problem at runtime. Skewtune [18] mitigates the skew for MapReduce applications by identifying the straggler, repartitioning its unprocessed input data, and rescheduling data to other nodes. Libra [3] has revealed the keys having more values may become a performance bottleneck in the reduce stage, and proposes a solution to repartition large keys with a new sampling method. OLH [25] proposes a key chopping method and a key packing method to split large keys and group medium keys, respectively. TopCluster [9] proposes a distributed monitoring framework to capture the local data distribution on each mapper, identify the most relevant subset data, and approximate the global data distribution. This method provides complete information for appropriate skew tacking methods. Although these mechanisms can mitigate the skew without the modification of applications, the effort to improve partitioning algorithms is still valuable, because application-specific partitioning methods can get better performance and scalability as illustrated in SkewReduce [17], PowerLyra [4], and Polymer [37]. It is possible to extend PaPar to support the dynamic workload redistribution. For example, when repartitioning intermediate data from Mappers to Reducers is necessary, we can use the PaPar distribution function with the cyclic policy to rebalance the key-value pairs between reducers.

For the static mechanisms, SkewReduce [17] proposes a cost function based framework for spatial feature extraction applications manipulating multidimensional data. PowerLyra [4] is a graph computation and partitioning engine for

skew graphs. The hybrid-cut method is proposed to partition input data. Polymer [37] is a graph processing engine for the NUMA compute node. A differentiated partitioning and allocation mechanism can put graph data into the local memory bank, and a NUMA-aware mechanism can convert random accesses on local memory to sequential accesses on remote memory. Our work proposes a framework to enforce user-defined partitioning methods, and can be integrated with the previous research to simplify their implementations.

VI. Conclusions

In this paper, we propose the PaPar framework to generate application-specific partitioning algorithms. Taking two configuration files as input, PaPar can formalize the partitioning workflow as a sequence of key-value operations and matrix-vector multiplications, and map to implementations on MPI and MapReduce. We use muBLASTP and PowerLyra as the case studies to show how to generate the user-defined partitioning algorithms with PaPar. Our evaluations illustrate PaPar can generate the same partitions with comparable or less partitioning time.

Acknowledgement

This research was supported in part by the NSF BIGDATA program via IIS-1247693 and the NSF XPS program via CCF-1337131. We also acknowledge Advanced Research Computing at Virginia Tech for access to high-performance computational resources.

References

[1] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[2] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.

[3] Q. Chen, J. Yao, and Z. Xiao. Libra: Lightweight data skew mitigation in mapreduce. *IEEE Transactions on Parallel and Distributed Systems*, 26(9):2520–2533, 2015.

[4] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*. USENIX Association, 2004.

[6] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for gpus:(via language support for architectures and compilers). In *ACM SIGPLAN Notices*, volume 47, pages 1–12. ACM, 2012.

[7] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel. Operator language: A program generation framework for fast kernels. In *Domain-Specific Languages*, pages 385–409. Springer, 2009.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[9] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *2012 IEEE 28th International Conference on Data Engineering*, pages 522–533. IEEE, 2012.

[10] Y. Guo, J. Rao, C. Jiang, and X. Zhou. Flexslot: Moving hadoop into the cloud with flexible slot management. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 959–969. IEEE, 2014.

[11] Y. Guo, J. Rao, C. Jiang, and X. Zhou. Moving mapreduce into the cloud with flexible slot management and speculative execution. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[12] K. Hou, H. Wang, and W.-c. Feng. Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 383–392. ACM, 2015.

[13] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang. Major technical advancements in apache hive. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1235–1246. ACM, 2014.

[14] W. Jiang, V. T. Ravi, and G. Agrawal. A map-reduce system with an alternate api for multi-core environments. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 84–93. IEEE Computer Society, 2010.

[15] T. Kaldewey, E. J. Shekita, and S. Tata. Clydesdale: structured data processing on mapreduce. In *Proceedings of the 15th international conference on extending database technology*, pages 15–25. ACM, 2012.

[16] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 75–86. ACM, 2010.

[18] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[19] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 225–236. IEEE, 2013.

[20] H. Lin, X. Ma, W. Feng, and N. F. Samatova. Coordinating Computation and I/O in Massively Parallel Sequence Search. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):529–543, 2011.

[21] W. Liu and B. Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350. ACM, 2015.

[22] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[23] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko. Hadoop-bam: directly manipulating next generation sequencing data in the cloud. *Bioinformatics*, 28(6):876–877, 2012.

[24] S. J. Plimpton and K. D. Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[25] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 16. ACM, 2012.

[26] S. N. A. Project. Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/.

[27] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15. ACM, 2015.

[28] S. Sehrish, G. Mackey, J. Wang, and J. Bent. MRAP: A Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 107–118. ACM, 2010.

[29] S. Tang, B.-S. Lee, and B. He. Dynamic job ordering and slot configurations for mapreduce workloads. *IEEE Transactions on Services Computing*, 9(1):4–17, 2016.

[30] H. Wang, W. Liu, K. Hou, and W.-c. Feng. Parallel transposition of sparse data structures. In *Proceedings of the 2016 International Conference on Supercomputing*, page 33. ACM, 2016.

[31] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A Novel MapReduce-like Framework for Multiple Scientific Data Formats. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 443–450. IEEE, 2012.

[32] Y. Wang, A. Nandi, and G. Agrawal. SAGA: Array Storage as a DB with Support for Structural Aggregations. In *Proceedings of the 26th international conference on scientific and statistical database management*, page 9. ACM, 2014.

[33] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013.

[34] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

[35] J. Zhang, S. Misra, H. Wang, and W.-c. Feng. mublastp: database-indexed protein sequence search on multicore cpus. *BMC bioinformatics*, 17(1):443, 2016.

[36] J. Zhang, S. Misra, H. Wang, and W.-c. Feng. Eliminating Irregularities of Protein Sequence Search on Multicore Architectures. In *Proceedings of the 31st IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2017.

[37] K. Zhang, R. Chen, and H. Chen. Numa-aware graph-structured analytics. In *ACM SIGPLAN Notices*, volume 50, pages 183–193. ACM, 2015.