UNIVERSITY OF CALIFORNIA
RIVERSIDE

Out-Of-Core MapReduce System for Large Datasets

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Gurneet Kaur

December 2021

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Zizhong Chen
Dr. Nael Abu-Ghazaleh
Dr. Zhijia Zhao

The Dissertation of Gurneet Kaur is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This work would not have been possible without the motivation and support of various people in my life.

I am forever grateful to my advisor, Dr Rajiv Gupta without his help, I would not have been here. I express my deepest gratitude to him for his unwavering, constant confidence and belief in my potential and capabilities. For his inspiration and guidance in my research, writing and presentations. For his understanding and patience through my highs and my lows. This dissertation is a culmination of his constant support, motivation and training throughout my PhD. Thank you, Dr. Gupta, for the most rewarding, memorable and treasured doctoral study under your guidance.

I would like to thank my dissertation committee members for their valuable feedback and support: Prof. Nael Abu-Ghazaleh and Prof. Zizhong Chen, Prof. Zhijia Zhao. I would like to thank Bourns College of Engineering for providing me the opportunity to carry out my doctoral studies. I would like to acknowledge the support of National Science Foundation via grants CCF-1524852, CCF-1813173, CCF-2002554, and CCF-2028714 to UC Riverside.

My daughter *Samreet* has been my main source of motivation to continue through my PhD journey. I started my PhD program around the time she was born. It has not been an easy journey with managing time between research and raising a kid and I hope I will be able to make it up to her in the upcoming years.

I remain ever grateful to my better-half *Gagandeep Singh* who is the key instigator in fueling my desire to pursue graduate studies. None of this would have been possible with-

out his unconditional love, understanding, co-operation and constant support throughout my PhD. Putting up with me through my grad-student life, taking care of our daughter, and supporting through thick and thins.

I am forever indebted to my parents for raising me as a confident child, fostering right values and always trusting me with all the decisions I have made. Thank you for educating me and helping me finish off my high school and successfully completing my undergrad in computer science even at the time of crisis in your business. I am thankful to my in-laws for their constant love, support and blessings. I am thankful to my brothers for being a source of inspiration through their own lives and motivating me to achieve better.

Finally, I would like to thank my best friends and all the family members for their endless love, support and blessings throughout my life.

To my better-half and parents for all the support. My daughter for being my inspiration to do better.

ABSTRACT OF THE DISSERTATION

Out-Of-Core MapReduce System for Large Datasets

by

Gurneet Kaur

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2021
Dr. Rajiv Gupta, Chairperson

While single machine MapReduce systems can squeeze out maximum performance from available multi-cores, they are often limited by the size of main memory and can thus only process small datasets. Even though today's computers are equipped with efficient secondary storage devices, the frameworks do not utilize these devices mainly because disk access latencies are much higher than those for main memory. Therefore, a single machine set up of Hadoop system performs much slower when it is presented with the datasets larger than the main memory. Moreover, such frameworks also require tuning a lot of parameters which puts an added burden on the programmer. While distributed computational resources are now easily available, efficiently performing large scale computations still remain a challenge due to out-of-memory errors and complexity involved in handling distributed systems. Therefore, we develop techniques to perform large-scale processing on a single machine by reducing the amount of IO and exploiting sequential locality when using disks.

First, this dissertation presents OMR, a single machine out-of-core MapReduce system that can efficiently handle datasets that are far larger than the size of main memory

and guarantees linear scaling with the growing data sizes. OMR actively minimizes the amount of data to be read/written to/from disk via on-the-fly aggregation and it uses block sequential disk read/write operations whenever disk accesses become necessary to avoid running out of memory. We theoretically prove OMRs linear scalability and empirically demonstrate it by processing datasets that are up to 5× larger than main memory. Our experiments show that in comparison to the standalone single-machine setup of the Hadoop system, OMR delivers far higher performance. Also OMR avoids out-of-memory crashes for large datasets and delivers high performance for datasets that fit in main memory.

Second, this dissertation presents a single-level out-of-core partitioner for large irregular graphs GO, which can successfully partition large graphs by performing just two passes over the entire input graph, *partition creation pass* that creates *balanced* partitions and *partition refinement pass* that reduces *edgecuts* in a memory constrained manner via disk-based processing. For graphs that can be successfully partitioned by the widely used Mt-Metis system on a single machine, GO produces balanced 8-way partitions with 11.8× to 76.2× fewer edgecuts using 1.9× to 8.3× less memory in comparable runtime.

Finally, we extend the API of the OMR system to enable graph partitioning and partition-based graph processing for large graphs that do not fit in memory. Our experiments show that the extended OMRGx system can be easily used to partition large graphs and perform the partition-based graph processing. The API provided allows the programmer to focus on the programming logic while remaining completely oblivious of the out-of-core processing required to handle large graphs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

MapReduce is a functional programming model to process large datasets in parallel. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all the intermediate values associated with the same intermediate key. Such a framework enables users to easily express the processing logic using the simple APIs while runtime takes care of all the other details like automatically scheduling computations across machines and managing the available resources.

The simplicity of MapReduce framework along with its applicability to practical problems has made it a popular choice amongst a range of computing platforms (e.g. Hadoop [1], Mars [9]). MapReduce model provides an efficient and scalable implementation on distributed runtime that scales the processing across available computing resources.

While distributed computational resources are now easily available, efficient large scale computations still remain a challenge due to out-of-memory errors and complexity

involved in handling distributed systems. Even distributed Hadoop commonly suffers from out-of-memory errors. iTask [6] reports 73 out-of-memory errors from stack overflow when using Hadoop in a distributed setting due to great memory pressure that occurs when processing large datasets. Although not all the errors have recommended fixes, fixing some of the errors requires tuning a number of parameters. More generally, using distributed systems requires users to be highly skillful since tuning a cluster using a number of parameters and debugging distributed algorithms are non-trivial tasks.

To avoid the complexity involved in handling distributed systems, performing analytics on single machine systems have gained popularity in recent years. The wide availability of multiple cores on todays desktops has led to the development of such data processing systems that can operate on a single machine. Although single machine systems are gaining popularity they are often limited by the available main memory. MapReduce Frameworks like Metis [14], Phoenix++ [21] are highly tailored to efficiently utilize the available cores and extract maximum efficiency to process large enough datasets that can fit in main memory. Such systems are naturally suitable for use cases requiring simple aggregations like counting, joining etc over data used/generated in other larger analyses and experiments. However, when such systems are presented with datasets larger than main memory, they simply fail.

Recent single machine graph analytics frameworks like GraphChi [13], GridGraph [47] etc have demonstrated that processing can scale beyond main memory by carefully utilizing disks. Such frameworks transparently support out-of-core processing by incorporating disk friendly data structures that can orchestrate disk accesses such that sequential disk

accesses get maximized. Furthermore, availability of frameworks like Infinimem [11] enable size oblivious programming by exposing simple read/write functions that can be directly used to scale runtime beyond main memory capacities.

Naively incorporating out-of-core support for MapReduce algorithms can significantly slow down the overall processing speed. Hadoop, for example, can be configured in a standalone single machine setting. However, it greatly suffers from performance bottlenecks due to massive exchange of intermediate results during its shuffle and sort phases that aggressively aims to sort the intermediate data to route key/value pairs to reducers, thereby causing larger (and more random) disk accesses.

Therefore, the goal of this dissertation is to develop an out-of-core MapReduce system that can transparently support out-of-core processing to efficiently handle large datasets in memory by lowering the overall I/O and by minimizing random disk accesses. We further extend this system and generalize it by supporting graph partitioning and partition-based graph processing.

1.1 Dissertation Overview

This dissertation presents OMRGx, a generic MapReduce framework that can efficiently handle large datasets as well as enables graph partitioning and partition-based graph processing. Figure 1.1 shows the overview of this dissertation. First, we propose OMR, an out-of-core MapReduce system which is highly optimized for single machine and can handle datasets far larger than the size of main memory. Second, we propose GO, a single-level out-of-core graph partitioner which can successfully partition large graphs in a

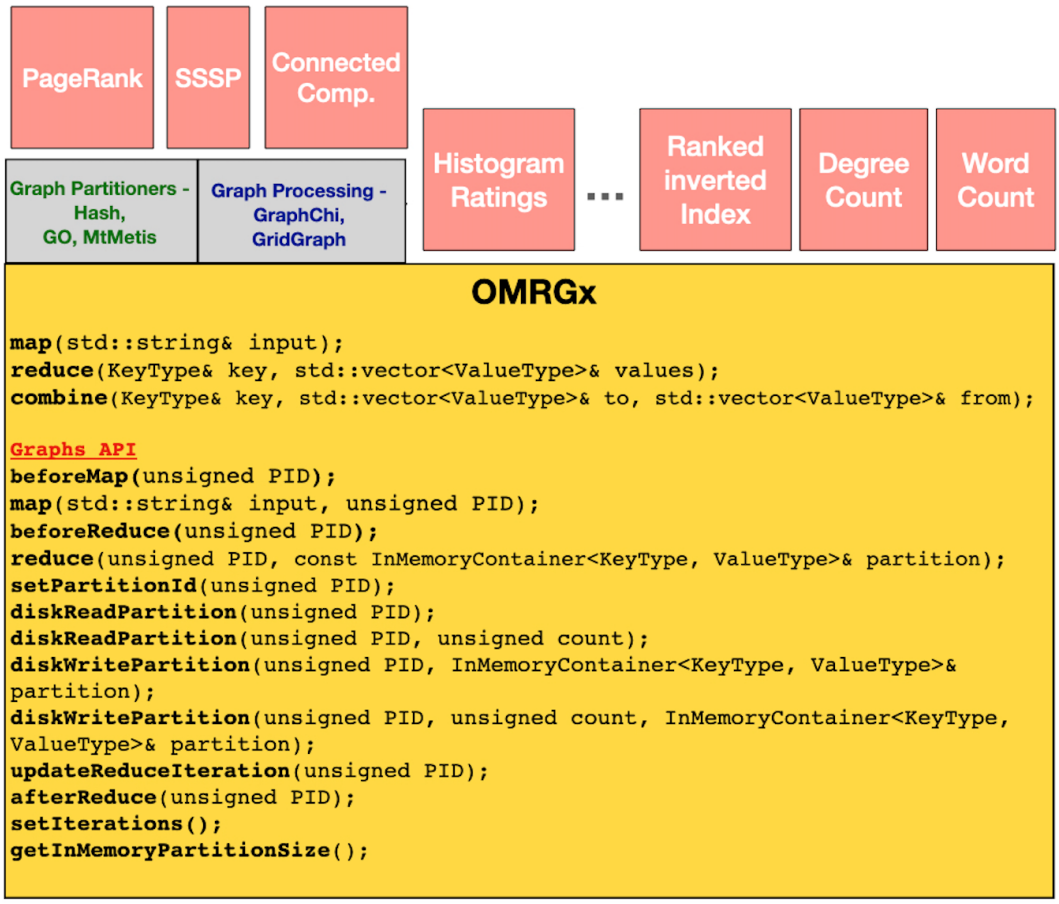


Figure 1.1: Dissertation Overview

memory constrained manner. Finally, we propose extending OMR to support end-to-end graph processing by providing the APIs for graph partitioning and partition-based graph processing. Next, we summarize the approaches proposed above.

1.1.1 Developing an Out-of-core MapReduce System

Naively incorporating secondary storage devices can lead to much higher disk access latencies. In Chapter 2, we describe our out-of-core MapReduce system to enable processing of very large datasets that cannot fit in main memory. Since secondary storage

incurs high processing costs, the main challenge is to scale linearly with size of these large datasets so that the processing times remain reasonably bounded. To achieve linear scalability, our system ensures that each `<key, value>` pair generated during the map phase is read/written from/to disk at most once throughout processing. The system also actively minimizes the intermediate data to be maintained while simultaneously maximizing sequential disk accesses to speed up the overall processing. It achieves this via the following techniques:

(a) Memory Constrained Processing. OMR partitions the available memory into disjoint bounded buffer spaces. This allows map and reduce threads to fully own the allotted buffer spaces for maintaining intermediate results. These buffer spaces are backed with record batches that reside on disk to avoid out-of-memory crashes.

(b) Sequential Block Disk Accesses via Ordered Records. OMR maintains a consistent ordering of intermediate results during the map phase using ordered in-memory buffers. As these buffers become full, they are written to disk in form of ordered batches of records using sequential block writes. During the reduce phase, the ordering across record batches allows threads to fetch required portions of batch records from disk using sequential block reads. Thus, maintaining ordering within batches of intermediate results eliminates the need to fully sort intermediate results.

(c) I/O Reduction via On-the-fly Aggregation. To reduce the amount of disk I/O, OMR actively performs combining operation during the map phase to merge intermediate results based on their keys. This is efficiently performed using the ordering information across in-memory buffers.

Using the above techniques, the I/O performed by OMR remains linear in size of intermediate data generated during the map phase, which makes it optimal (as proved in Section 2.4). Moreover, OMR optimizes data management via fixed sized key-value pairs by eliminating the need to maintain indexing information on disk, which further reduces random disk accesses. Finally, to maintain efficient in-memory execution, OMR employs lockless processing that eliminates thread synchronization within map and reduce phases.

1.1.2 Partitioning Large Graphs on a Single Machine

Graph analytics is employed in many domains to gain insights by analyzing large graphs representing entities and interactions among them. Real world graphs often contain millions of vertices and billions of edges, and iterative graph analytics queries require multiple passes over the graph until convergence. Therefore, there has been a great deal of interest in developing scalable graph analytics systems that exploit parallelism on shared memory systems.

Before parallel analytics on large graphs can be performed, they typically must first be partitioned. In context of a single shared-memory machine, the graph is partitioned to enable *out-of-core* or disk-based processing of a large graph on a single machine. When the graph is too large to fit in the memory available on the machine, it is divided into smaller partitions and stored on disk. This enables partitions to be loaded one at a time into memory and processed. For superior performance, partitioning algorithms endeavor to create partitions that are well balanced and minimize edgecuts (i.e., number of edges that cross partition boundaries).

Although the problem of graph partitioning is known to be NP-hard, highly effective *multilevel graph partitioning* algorithms have been developed and are widely used. Multilevel graph partitioning algorithms generally have higher memory requirements due to holding the original graph and all its coarsened graphs in memory. Therefore, the multilevel graph partitioning algorithms used by the most sophisticated partitioners cannot be run on the same machine as their memory requirements far exceed the size of the graph. To overcome this problem, we describe our *single-level* GO out-of-core graph partitioner that can successfully partition large graphs on a single machine. GO performs just two passes over the entire input graph, *partition creation pass* that creates balanced partitions and *partition refinement pass* that reduces edgecuts. Both passes function in a memory constrained manner via disk-based processing.

1.1.3 Extended Support for Graph Partitioning and Partition-based Graph Processing in OMR

The ease of expressing wide variety of algorithms using our highly optimized MapReduce system made us think in another direction - why not use MapReduce system for partitioning and end-to-end processing of large graphs? Such a system is already equipped with efficient engine to leverage parallelism, handle data efficiently in memory and perform optimized IO operations. Therefore, a *highly tailored* single machine MapReduce system becomes a good choice to partition and process large graphs that often need to be written off to disk due to their iterative nature and BSP model semantics. Hence, we can use the simple APIs provided by the MapReduce system to express the vertex-centric computations. Such a framework is also ideal to partition the graphs by mapping the data

to different partitions and assigning each reducer the task of refining each partition. This led us to take a step forward and extend our existing MapReduce system - *OMR* to support Graph Partitioning and Processing.

1.2 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 presents an out-of core MapReduce systems that works in a *memory constrained* manner and avoids out-of-memory crashes seen by the other systems. Chapter 3 presents a *single-level* out-of-core graph partitioner which performs just two passes to partition the input graph - *partition creation pass* and *partition refinement pass*. In Chapter 4, we describe a *generic* MapReduce framework which can partition and process large graphs. Chapter 5 discusses various research work in literature and Chapter 6 concludes the thesis as well as discusses directions for the future work.

Chapter 2

Out-of-core MapReduce for Large Datasets

The prevalence of large datasets has led to development of various efficient big data processing tools like Spark [26], MapReduce [5], PowerGraph [30], and many others. Such tools typically provide a simplified programming model along with an efficient runtime system to scale processing across available computing resources. The programming model enables users to easily express the processing logic using simple APIs (for example, `map()`, `reduce()`, etc.) whereas the runtime takes care of automatically scheduling computations across machines and managing the available resources to provide best performance.

While the above systems were initially designed for distributed processing environments, the wide availability of multiple cores on today's desktops has led to the development of such data processing systems that can operate on a single machine. Single machine MapReduce frameworks like Metis [14], Pheonix++ [21] and others [4, 10, 25] are

highly tailored to efficiently utilize the available cores and extract maximum efficiency to process large enough datasets that can fit in main memory. Such systems are naturally suitable for use cases requiring simple aggregations (like counting, joining, etc.) over data used/generated in other larger analyses and experiments. However, these systems are fundamentally designed for in-memory data processing. This means, their processing capabilities are severely limited by the amount of main memory since datasets are, more often than not, much larger than main memory sizes.

Recent works like [13, 45] have demonstrated that processing can efficiently scale beyond main memory by carefully employing disks (i.e., out-of-core processing) and developing a disk-friendly runtime to consciously maximize disk access bandwidth. Furthermore, availability of frameworks like InfiniMem [11] enable size oblivious programming by exposing simple read/write functions that can be directly used to scale runtime systems beyond main memory's capacity. However, naively enabling out-of-core support for MapReduce algorithms can significantly slowdown the overall processing since the shuffling and sorting phase typically performs massive exchange of intermediate results to route key-value pairs to reducers; such massive exchange in an out-of-core setting can lead to a high volume of random disk accesses, causing a strong performance bottleneck. While external sorting algorithms can help alleviate this issue, they still require multiple passes over data residing on disk which can be very expensive. This leaves us with an important challenge: *how to design an out-of-core MapReduce execution model that efficiently maps in-memory intermediate results to disk records such that: a) the overall disk I/O remains low; and, b) random disk accesses get minimized.*

In this chapter, we present an out-of-core MapReduce system - *OMR* which can efficiently process datasets *far larger* than the size of main memory and guarantees *linear scaling* with the growing data sizes.

2.1 Overview

OMR[1], a single machine Out-of-core MapReduce system that can successfully process datasets that are much larger than main memory sizes. To limit slowdowns from I/O overheads, OMR guarantees linear scaling with growing data sizes when using disks for processing by actively reducing the amount of data written/read to/from disk, and by ensuring that block sequential disk accesses are maximized.

OMR achieves its goals via three key techniques. First, OMR partitions the available memory into disjoint bounded buffer spaces. This allows map and reduce threads to fully own the allotted buffer spaces for maintaining intermediate results. These buffer spaces are backed with record batches that reside on disk to avoid out-of-memory crashes. Second, OMR maintains a consistent ordering of intermediate results during the map phase using ordered in-memory buffers. As these buffers become full, they are written to disk in form of ordered batches of records using sequential block writes. During the reduce phase, the ordering across record batches allows threads to fetch required portions of batch records from disk using sequential block reads. Thus, maintaining consistent ordering within batches of intermediate results eliminates the need to fully sort intermediate results. Third, to reduce the amount of disk I/O, OMR actively performs combining operation during the map phase

[1]OMR is pronounced *Homer*.

to merge intermediate results based on their keys. This is efficiently performed using the ordering information across in-memory buffers.

Using the above techniques, the I/O performed by OMR remains linear in size of intermediate data generated during the map phase, which makes it optimal (as proved in Section 2.4). Moreover, OMR optimizes data management via fixed sized key-value pairs by eliminating the need to maintain indexing information on disk, which further reduces random disk accesses. Finally, to maintain efficient in-memory execution, OMR employs lockless processing that eliminates thread synchronization within map and reduce phases.

The key contributions of this paper are as follows.

- We present a single machine out-of-core MapReduce system that processes datasets whose sizes are larger than main memory sizes.
- We design a lockless memory constrained processing model that actively reduces the amount of disk I/O via on-the-fly aggregation, and ensures that block sequential disk accesses get maximized whenever disk operations are required.
- We develop a key optimization that enables the use of fixed-sized records to eliminate maintenance of indexing information on disk and further reduce random disk accesses.
- We thoroughly evaluate OMR using eight MapReduce algorithms and compare its performance with that of Metis [14], a state of the art single-machine in-memory MapReduce system. The experimental results in Figure 2.2, Figure 2.3, and Figure 2.4 show that OMR efficiently processes datasets that are up to 5× larger than main memory, proving the linear scalability of OMR as analyzed in Section 2.4. In contrast,

Metis fails to process large data sets. Furthermore, results in Table 2.5 show that OMR outperforms Metis for smaller datasets that can fit in main memory and thus can be successfully processed by Metis.

- We also compare the performance of OMR with a standalone (single machine setup) Hadoop [1] system in Figure 2.2. While after tuning many parameters available on Hadoop, Hadoop is able to process large data sets, OMR outperforms Hadoop by 1.5-41.2× for data sets ranging from 8GB to 80GB in size.

2.2 MapReduce on a Single Machine

MapReduce [5] is a popular execution model to process large datasets in parallel. At the heart of MapReduce is a simple programming model consisting of two functions, `map()` and `reduce()`, and a scalable runtime system that efficiently manages data across *map* and *reduce* processing phases.

As an example, Algorithm 1 shows how the logic to count word frequencies can be expressed in MapReduce. We use the notation `<x, y>` to represent a pair of `x` and `y`, and `[z]` to represent a list containing multiple values; hence, `<x, [y]>` is a pair of `x` and the list `y`. In Algorithm 1, for each word in the input, the `map()` function emits a `<key, value>` pair (line 3) with key being the word itself and value being 1. The system aggregates all the values for a given key and passes the resulting `<key, [value]>` pair to `reduce()` function where the individual counts are summed together to produce the frequency of each word.

The programming model provides an optional `combine()` function to merge intermediate values for same key in the map phase to reduce processing needs for shuffling

`<key, [value]>` pairs and for the subsequent reduce phase. While expressing only the core processing logic using functions `map()`, `reduce()` and optionally `combine()` makes it easy for programmers to work with large datasets, the runtime remains burdened with processing details like parallelism, data partitioning, synchronization and resource management for efficient and scalable processing system.

Algorithm 1 Word count example in MapReduce

```
1: function MAP(line)
2:     for word ∈ line do
3:         emit(<word, 1>)
4:     end for
5: end function

6: function REDUCE(<key,[value]>)
7:     frequency ← SUM([value])
8:     OUTPUT(key, frequency)
9: end function
```

One option for employing MapReduce on a single machine is to use Hadoop as it can be setup in a standalone single machine configuration. Hadoop is able to handle large inputs using disks so that it doesn't run out of memory. However, since it is not fundamentally designed for a single machine environment, its shuffle and sort (merge) phase aggressively aims to sort the intermediate data which leads to larger (and more random) disk I/O. Even though Hadoop's performance can be improved by tuning a number of parameters, its workflow cannot be drastically changed to suit the needs of out-of-core processing where minimizing disk access while economically utilizing main memory becomes the primary concern. Moreover, parameter tuning imposes great deal of burden on the users and requires them to gain an architectural understanding of Hadoop. This is contrary to the simplicity of the MapReduce [5] model which aims at providing the user with a

simple programming interface without requiring detailed understanding of the underlying architecture. Our experimental results in Section 5 show that after tuning the available parameters, even though we are able to process large inputs using Hadoop, the observed execution times are still very high.

Another option available is to make use of a single machine MapReduce framework [4, 10, 14, 17, 21, 25] that aim to efficiently utilize the available cores to process large enough datasets that can fit in main memory. For example, Metis [14], designed optimally for single machine, minimizes the bottlenecks involved in aggregating `<key, [value]>` pairs by reorganizing intermediate data in a hash table of B+ tree entries. It also reduces thread synchronization across map and reduce phases by first allowing map threads to operate on separate hash tables and then repartitioning hash table entries for reducers to individually work on them.

While such frameworks provide high performance, the limited amount of main memory severely curtails their processing capabilities. For example, on a standard multicore machine with 16GB main memory, Metis can successfully process 8-16GB datasets for only two out of eight MapReduce algorithms as shown in Table 2.1. Even though today's computers are equipped with efficient secondary storage devices, the frameworks do not utilize them since disk access latencies are much higher than those for main memory.

Recent works in out-of-core big data processing like [13, 45] have shown that large amounts of data can be processed on a single machine by carefully orchestrating disk accesses such that sequential disk accesses get maximized. Furthermore, availability of frameworks

Table 2.1: Performance of Metis [14] on various MapReduce algorithms – ✗ represents out-of-memory crashes.

Algorithms	**Key Size**	**Value Size**	**Dataset**	**Metis**
WordCount	1-32 bytes	8 bytes	8GB	✓
			16GB	✓
			32GB	✗
InvertedIndex	1-32 bytes	32 bytes	8GB	✓
			16GB	✗
SequenceCount	7-132 bytes	4 bytes	8GB	✗
RankedInvert- edIndex	5-98 bytes	130 bytes	8GB	✗
MovieRatings	8 bytes	4 bytes	8GB	✗
DegreeCount	8 bytes	4 bytes	8GB	✗
			16GB	
AdjacencyList	8 bytes	Unbounded	8GB	✗
			16GB	
SelfJoin	1-32 bytes	Unbounded	8GB	✗

like InfiniMem [11] enable size oblivious programming by exposing simple read/write functions that can be directly used to scale runtime systems beyond main memory capacities. Therefore, in this work we develop an out-of-core MapReduce system to enable processing of very large datasets that cannot fit in main memory. Since secondary storage incurs high processing costs, the main challenge is to scale linearly with size of these large datasets so that the processing times remain reasonably bounded. To achieve linear scalability, our system ensures that each `<key, value>` pair generated during the map phase is read/written from/to disk at most once throughout processing. The system also actively minimizes

the intermediate data to be maintained while simultaneously maximizing sequential disk accesses to speed up the overall processing.

2.3 OMR: Out-of-core MapReduce system

In this section, we will first present our memory constrained processing model, and then discuss the I/O aspects across different phases of processing.

2.3.1 Minimizing I/O Overheads

A straightforward way to support processing of large datasets on a single machine is to incorporate disk-friendly out-of-core data structures in place of regular in-memory data structures. While such a solution will ensure that the system is no longer bounded by the main-memory size, it will effectively transform the accesses to irregular data-structures into random disk accesses, hence slowing down the entire processing. Furthermore, the traditional MapReduce processing model includes a shuffling phase where keys are sorted to be sent out to respective reduce tasks and performing this phase over data residing in out-of-core data structures can further increase random disk accesses and greatly impacting the processing speed.

To ensure reasonable input scalability, it becomes necessary to actively manage disk accesses such that expensive random accesses get minimized. Furthermore, sequential locality can be exploited by performing disk read and write operations at a coarser granularity, i.e., in blocks. In other words, maximizing sequential disk accesses can increase the disk utilization bandwidth, hence reducing the overheads of utilizing disks while processing.

Hence, we design the OMR processing model to ensure that:

1. Each `<key, value>` pair is written to disk at most once during map phase.

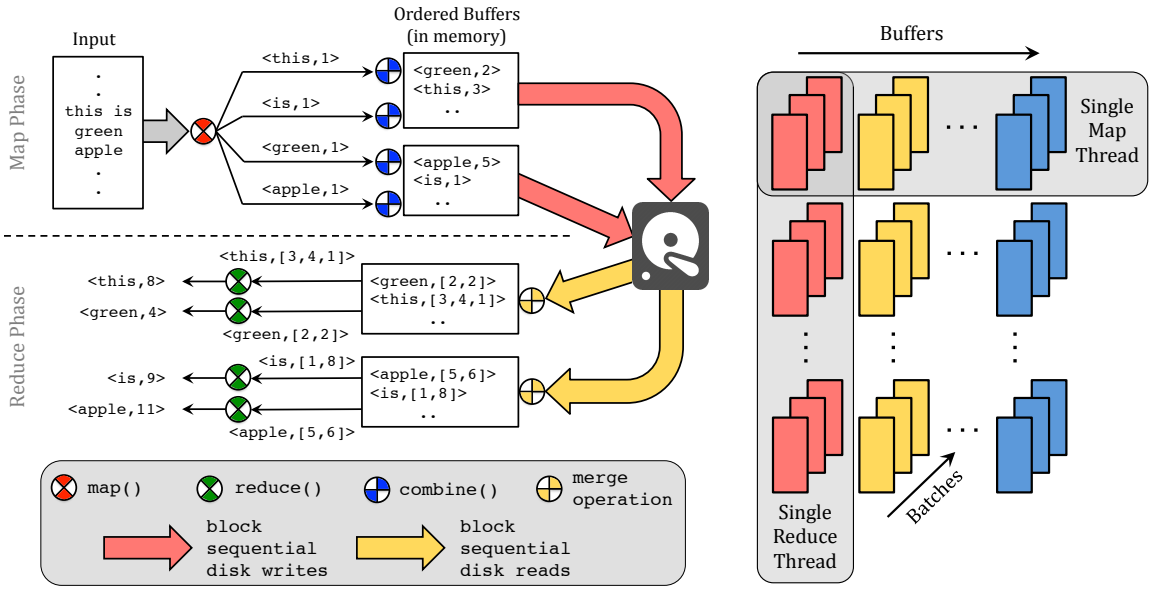
2. Each `<key, value>` pair is read from disk at most once during reduce phase.

3. Sequential disk accesses get maximized during both map and reduce phases.

2.3.2 Lockless Memory Constrained Processing Model

OMR maintains the above properties by incorporating a memory constrained processing model. In this model, we partition the available memory into disjoint buffer spaces such that their size remains bounded throughout the execution. As shown in Figure 2.1a, the `<key, value>` pairs emitted by the user-defined `map()` function are maintained in size-constrained *ordered in-memory buffers* that are ordered based on keys. As the buffers grow large, they are spilled off to disk to make room for further `<key, value>` pairs. These ordered buffers are written in entirety as *ordered batches* on the disk. Hence, at the end of the map phase, the disk contains multiple ordered batches, which become input to the subsequent reduce phase. Since these batches are ordered based on keys, the reduction process streams multiple batches from beginning to end and collects values for same keys (similar to merge process in mergesort) to pass `<key, [value]>` pairs to user-defined `reduce()` function.

The concurrent execution in the presence of multiple threads can be visualized by laying out the batches for each buffer in a three dimensional grid as shown in Figure 2.1b. During map phase, each thread owns a row of in-memory buffers and the `<key,`



(a) Overview of out-of-core MapReduce.

(b) Buffers and batches.

Figure 2.1: OMR Overview.

`value>` pairs processed by a thread are hashed to one of the buffer in its respective row using a user-defined consistent hashing function. As the size of buffers reach a certain memory threshold, they are written to disk as batches so that they can be emptied for further processing. Presence of multiple batches for each buffer is represented via the z dimension in Figure 2.1b. At the end of the map phase, any given key would be present in some or all of the batches with the same color. Hence, during the reduce phase, each thread independently operates on batches with the same color. This means, threads during the reduce phase own batches with the same color. Such disjoint ownership of buffers and batches during map and reduce phases enables threads to independently process data in the system, causing the entire data processing pipeline to be lockless. Next, we discuss different I/O aspects of the map and reduce phases in detail.

Algorithm 2 Map Phase

```
1: tid: Thread identifier
2: buffers[*][*]: Buffers in memory (2-dim)
3: batches[*][*][*]: Batches on disk (3-dim)
4: input: Input from file to map function
5: threshold: Parameter controlling block writes

6: for kvpair emitted by map(input) do
7:     key ← kvpair.getKey();
8:     value ← kvpair.getValue();
9:     bufferId ← hash(key);
       // Alias buffer
10:    buffer ← buffers[tid][bufferId];
11:    if key ∈ buffer then
12:        bufferValue ← buffer.getValue(key);
13:        combinedvalues ←
               combine(bufferValue, value);
14:        buffer.setValue(key, combinedvalues);
15:    else
16:        buffer.insertNew(kvpair);
17:    end if
18:    if |buffer| ≥ threshold then
19:        batch ← serialize(buffer);
20:        blockdiskwrite(batch);
           // Alias tbatches
21:        tbatches ← batches[tid][bufferId];
22:        tbatches.insertNewBatch(batch);
23:        buffer.clear();
24:    end if
25: end for
```

2.3.3 Map Phase: Sequential Writes of Ordered Batches

The `<key, value>` pairs emitted during map phase are maintained in *ordered in-memory buffers*. We employ red-black trees as our in-memory buffers with ordering based on keys. While the payload on these tree nodes can directly be the emitted value, we allow the nodes to hold list of values to support on-the-fly aggregation of values for same keys. Hence, multiple `<key, value>` pairs with same key are aggregated within these buffers as `<key, [values]>` pairs using the user-defined `combine()` function. Such on-the-fly aggregation

reduces the buffer size by: A) not repeating the key for each `<key, value>` pair; and, B) reducing values to be managed when aggregation results in scalar values (e.g., `sum`, `min`, etc.). The amount of size reduction achieved due to aggregation varies based on the dynamic interplay between available memory size and the nature of input.

Furthermore, if enough memory is available, the buffers are never spilled to disk and all the `<key, [value]>` pairs represent all values for their corresponding keys and hence, they can be directly sent to the reduce phase.

Algorithm 2 shows how `<key, value>` pairs are processed during map phase. In order to write a buffer off to disk (lines 18-24), its contents are serialized into a batch of contiguous records such that each `<key, [value]>` pair becomes a record. The records within a batch are ordered under the same ordering as that of their corresponding pairs in the buffer from which they are serialized. Since records within the batch are contiguous, as shown in Figure 2.1a, the entire batch can be written out to disk via a single sequential write (`blockdiskwrite()` on line 20). We use InfiniMem's block-based I/O [11] to seamlessly manage disk writes. In particular, InfiniMem provides efficient I/O support for variable size records which we leverage since size of our records vary based on number of values aggregated and variable sized key/value types (e.g., `string`). To support variable size records, InfiniMem also writes index information for batch records on disk; as we will see in Section 2.3.5, the runtime completely eliminates I/O for this additional index information for cases where record sizes remain fixed.

It is important to note that each `<key, value>` pair emitted by the `map()` function is written to disk at most once throughout the map phase. In absence of on-the-fly

aggregation, each `<key, value>` pair would be written exactly once to disk and the aggregation reduces this by combining multiple `<key, value>` pairs into `<key, [value]>` pairs. In worst case, the aggregation simply results in list of original values in `[value]` which still reduces the amount of data written to disk due to elimination of multiple same keys. On other hand, the ideal case is when aggregation results in a single scalar value which reduces disk write cost of multiple `<key, value>` pairs to a single `<key, value>` pair.

2.3.4 Reduce Phase: Sequential Reads from Ordered Batches

Since the map phase organizes data as ordered batches on the disk, the reduce phase carefully orchestrates reads from disk to maximize sequential disk reads. This is done using a process that streams records from batches in a manner similar to the merge phase in merge-sort algorithm. Furthermore, the map phase ensures that any given key can only be present in batches of the same color in Figure 2.1b, hence making subsets of batches independent of each other for the reduce phase. The runtime leverages this invariant during reduce phase by assigning one reduce thread for each disjoint set of buffers that hold similar set of keys.

Algorithm 3 shows processing performed during the reduce phase. The GETNEXTMINKV() function streams records from multiple batches belonging to a given reduce thread (line 18) using blocked sequential reads from batches on disk; `blockdiskread()` (line 19) reads k contiguous records from a given batch starting from next unread record. Since batches contain variable size records, we use InfiniMem's block-based I/O [11] to seamlessly manage reading of multiple variable size records from disk. The records are then deserial-

ized (line 20) to become `<key, [value]>` pairs and multiple pairs from different batches with same key are aggregated together by concatenating their list of values (line 21) to form a single `<key, [value]>` pair, as shown in Figure 2.1a. Since blocks of k records are read from each batch, it is guaranteed that all the values associated to the least k keys are available in `mergedpairs` at line 7. Hence, the pairs with k least keys are sent to `reduce()` for processing (lines 8-11).

Algorithm 3 Reduce Phase

```
1:  tid: Thread identifier
2:  batches[*][*][*]: Batches on disk (3-dim)
3:  k: Parameter controlling block reads

4:  mergedpairs ← ∅
5:  while unprocessed kvpairs ∈ batches[*][tid][*]
    do
6:      pairs ← getNextMinKV(tid, k);
7:      mergedpairs ← mergedpairs ∪ pairs;
8:      for pair ∈ k least pairs of mergedpairs do
9:          reduce(pair);
10:         mergedpairs ← mergedpairs \ {pair}
11:     end for
12: end while

13: for pair ∈ mergedpairs do
14:     reduce(pair); // Remaining pairs
15: end for

16: function GETNEXTMINKV(tid, numpairs)
17:     allpairs ← ∅
18:     for batch ∈ batches[*][tid][*] do
19:         records ←
              blockdiskread(batch, numpairs);
20:         pairs ← deserialize(records);
21:         allpairs ← allpairs ∪ pairs;
22:     end for
23:     return allpairs;
24: end function
```

It is important to note that since `blockdiskread()` always reads contiguous records starting from the earliest unread records, the runtime streams through all the batches exactly once during the reduce phase. This means, each `<key, value>` pair generated from the original input is read from disk at most once throughout the reduce phase because it is written to disk at most once throughout the map phase (see Section 2.3.3).

2.3.5 Optimizing I/O for Fixed Size Types

While InfiniMem provides efficient support for variable size `<keys, [values]>` pairs, it also indexes these records in order to correctly retrieve records from disk. This means, along with writing batches of varying record sizes on disk during the map phase,

Table 2.2: MapReduce algorithms and sizes associated with their keys and values.

Algorithm	**Key/Size**	**Value/Size**	**Combine/Size**	**Fixed Size**
MovieRatings	Integer/Fixed	Integer/Fixed	Integer/Fixed	
HistogramMovies	Integer/Fixed	Integer/Fixed	Integer/Fixed	Yes
DegreeCount	Integer/Fixed	Integer Pair/Fixed	Integer Pair/Fixed	
WordCount	String/Variable	Integer/Fixed	Integer/Fixed	Conditional:
InvertedIndex	String/Variable	Bit Vector/Fixed	Bit Vector/Fixed	bounded
SequenceCount	String/Variable	Integer/Fixed	Integer/Fixed	maximum
RankedInvertedIndex	String/Variable	Bit Vector/Fixed	Bit Vector/Fixed	key size
Grep	Integer/Fixed	String/Variable	String Vector/Variable	
SelfJoin	String/Variable	String/Variable	String Vector/Variable	No
AdjacencyList	Integer/Fixed	Pair of Integers/Variable	Vector of Pair of Integers/Variable	

InfiniMem also writes batches of index information corresponding to those records on disk. Similarly, in the reduce phase, InfiniMem first reads batches of index information from disk to then correctly read batches of variable size records from disk. While maintaining such additional information is unavoidable for variable size records, OMR automatically eliminates it for fixed size records.

To use fixed size records, we first understand the source of variability involved in record sizes. The size of records becomes variable due to three reasons: variable key size, variable value size, and variable number of values in `<key, [value]>` pairs. Since values are typically numerical types like integers, doubles, etc., they can easily be of same fixed size (e.g., 8 bytes for `uint64_t` in C++). Furthermore, for such numerical value types, partial aggregation capabilities of `combine()` function can retain the same fixed size by maintaining number of values in `<key, [value]>` to be exactly one; this is common across various reduction operations like `sum`, `min`, etc. which can be naturally decomposed. Table 2.2 shows various MapReduce algorithms and sizes associated with their keys and values. While algorithms use variable size strings for keys for efficient text processing, there are various algorithms for which keys are numerical types, making them fixed size. Also, algorithms relying on string types for keys can still use fixed size records by leveraging the domain knowledge of maximum size of keys across different input datasets.

Based on type information of keys and values and the return type of the `combine()` function, the runtime can automatically switch to utilizing fixed size records instead of variable size records. By using fixed size records, we completely eliminate maintenance of the index information for records and the related disk I/O operations induced by InfiniMem.

InfiniMem's uniform API for I/O with fixed and variable size records allows minimal changes in the runtime to incorporate this optimization. Furthermore, this optimization does not impact the overall MapReduce interface for users, hence allowing them to seamlessly benefit from this optimization.

2.4 I/O Analysis

We analyze the I/O efficiency of OMR in terms of number of block transfers between disk and main memory. We will show that the I/O cost of our strategy is linear in terms of the total size of `<key, value>` pairs generated by `map()` function, hence making it optimal. We first analyze the I/O cost when using fixed size records and then, extend it to the general case of using variable size records.

A) Fixed Size Records: Let B be the size of disk block transfer in terms of number of `<key, value>` pairs. With the total number of `<key, value>` pairs generated by `map()` function as n, we compute an upper bound in terms of block transfers as the ratio of n and B with an added cost of number of non-sequential seeks. The amount of I/O performed during runtime depends on the input's characteristic to leverage on-the-fly aggregation during the map phase. Let p be the average number of `<key, value>` pairs that get aggregated during map phase before the corresponding buffer is written off to disk ($p \geq 1$). This means, the amount of I/O decreases with increase in p. Since each batch of records gets written once during the map phase and then read once during the reduce phase, the total number of block transfers for each phase is $(n/p)/B$. To capture non-sequential disk seeks, we define b_m (b_r) as the number of contiguous pairs written (read) during the map (reduce) phase. Hence,

the number of non-sequential disk seeks can be computed by dividing the total number of pairs by b_m and b_r. For easier illustration, we assume p, B, b_m and b_r to be appropriate factors of n. Hence, the total I/O cost C_B can be computed as:

$$C_B = \underbrace{2 \cdot \frac{n/p}{B}}_{\substack{\text{sequential} \\ \text{block transfers}}} + \underbrace{\frac{n/p}{b_m} + \frac{n/p}{b_r}}_{\substack{\text{non-sequential} \\ \text{disk seeks}}} \tag{2.1}$$

Since $b_m > B$ and $b_r > B$, C_B is linear in $(n/p)/B$.

B) Variable Size Records: Here, B is the average size of disk block transfer in terms of number of records containing `<key, [value]>` pairs, and p, b_m and b_r are similarly defined over `<key, [value]>` pairs. Since there are multiple sources of variability in record sizes, we can bound the I/O cost by analyzing these different sources.

A lower bound can be achieved when keys are variable size and the on-the-fly aggregation strategy results in a single scalar value. This case is similar to WordCount benchmark in Table 2.2. The size of index information maintained by InfiniMem for a `<key, [value]>` pair is smaller than the size of the pair itself. By using B_i as the average size of disk block transfer in the unit of index information, we can compute the total number of block transfers for index information for each phase as $(n/p)/B_i$. Also, InfiniMem performs a batch write (and read) of index information for every batch write (and read) of the record pairs. This means, the number of non-sequential disk seeks double compared to Eq. 2.1. Hence, the lower bound on total I/O cost C_B^L can be computed as:

$$C_B^L = \underbrace{2 \cdot \frac{n/p}{B}}_{\substack{\text{record} \\ \text{block transfers}}} + \underbrace{2 \cdot \frac{n/p}{B_i}}_{\substack{\text{index} \\ \text{block transfers}}} + \underbrace{2 \cdot \left(\frac{n/p}{b_m} + \frac{n/p}{b_r}\right)}_{\substack{\text{record \& index} \\ \text{disk seeks}}} \tag{2.2}$$

It is important to note that $B_i \geq B$ which makes C_B^L linear in $(n/p)/B$.

The upper bound can be achieved when on-the-fly aggregation strategy results in vector of values such that it retains each of the generated value. This case is similar to SelfJoin benchmark in Table 2.2. While the I/O costs for disk seeks and index block transfers remain same as in Eq. 2.2, block transfers for records themselves can be computed by explicitly using sizes as parameters. Let S_p and S_k be the average sizes of a `<key, value>` record and key respectively. Since aggregation eliminates redundant keys, we need to subtract sizes of these redundant keys from total size of all pairs, i.e., $n \times S_p$. The total number of batches written during map phase is $(n/p)/b_m$ and for each of these batches, $p-1$ redundant keys are removed by on-the-fly aggregation. Hence, the upper bound on total I/O cost C_B^U can be computed as:

$$C_B^U = \underbrace{\frac{n \cdot S_p - \{(\frac{n}{p \cdot b_m}) \cdot (p-1)\} \cdot S_k}{B \cdot S_p}}_{\substack{\text{record} \\ \text{block transfers}}} + \underbrace{2 \cdot \frac{n/p}{B_i}}_{\substack{\text{index} \\ \text{block transfers}}} + \underbrace{2 \cdot \left(\frac{n/p}{b_m} + \frac{n/p}{b_r}\right)}_{\substack{\text{record \& index} \\ \text{disk seeks}}} \tag{2.3}$$

Using Eq. 2.2 and Eq. 2.3, we bound the total I/O cost C_B as:

$$C_B^L \leq C_B \leq C_B^U \tag{2.4}$$

making C_B linear in n/B (from C_B^U in Eq. 2.3).

C) Non-Sequential Disk Seeks: Finally, Eq. 2.1, Eq. 2.2 and Eq. 2.3 identify that non-sequential disk seeks can be reduced by increasing b_m and b_r. Variable b_m is controlled by the `threshold` parameter in Algorithm 2 whereas b_r is controlled by the `k` parameter in Algorithm 3. Hence, by increasing `threshold`, we utilize a larger portion of main memory during the map phase which effectively delays writes to disk as much as possible. Similarly, by increasing `k`, we utilize larger portion of main memory during the reduce phase by reading larger batches of records and hence, delaying reads from disk as much as possible.

2.5 Evaluation

2.5.1 Experimental Setup

We developed OMR using the InfiniMem I/O runtime [11] to leverage its support for seamless batch disk I/O for fixed size and variable size records. OMR is a generic framework that accepts custom key and value types, making it easier for the user to express a wide variety of MapReduce algorithms. Both, variable size and fixed size types are directly expressed using Protocol Buffers [3] which provides efficient serialization/deserialization. For fixed size `string` types we use C++ structs.

Table 2.3 shows the MapReduce algorithms used from [2] to evaluate our OMR system. While WC, II, SC and RII use `string` types making them variable size, we also evaluate them using fixed size records by bounding the word size to 32 characters as a conservative estimate [16]. Since SC and RII operate on tuples of continuous words and file names, their record sizes are larger compared to WC and II. We used input datasets from [2] to create inputs of sizes between 8GB to 80GB.

Table 2.3: MapReduce algorithms.

Algorithm	**Record Type**	**Fixed Record Size**
WordCount (WC)	Variable+Fixed	40 bytes
InvertedIndex (II)	Variable+Fixed	64 bytes
SequenceCount (SC)	Variable+Fixed	136 bytes
RankedInvertedIndex (RII)	Variable+Fixed	228 bytes
MovieRatings (MR)	Fixed	12 bytes
DegreeCount (DC)	Fixed	12 bytes
AdjacencyList (AL)	Variable	N/A
SelfJoin (SJ)	Variable	N/A

All experiments were conducted on a single machine with 8 cores and 16GB main memory, equipped with 500GB SSD and running 64-bit Ubuntu 14.04. The sequential read and write bandwidth of the SSD is up to 550MB/s and 520MB/s respectively, whereas the random read and write performance is up to 50K IOPS and 60K IOPS respectively.

2.5.2 Performance

To evaluate our OMR system, we compare the performance of following four single machine MapReduce versions:

- **OMR-VR:** This is our out-of-core MapReduce system using variable size records.
- **OMR-FX:** This is our out-of-core MapReduce system using fixed size records.

- **Hadoop:** This is the standalone single machine setup of Hadoop system [1].
- **Metis:** This is the state of the art, high performance in-memory MapReduce system [14].

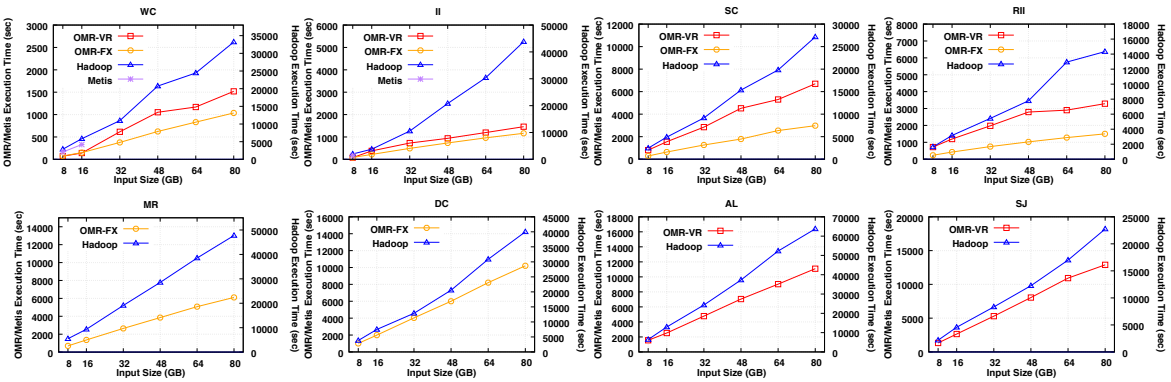


Figure 2.2: Left y-axis represent execution times in seconds for OMR-VR, OMR-FX and Metis. Right y-axis represent execution times in seconds for Hadoop. Majority of Metis datapoints are absent because it could not handle large datasets.

Figure 2.2 presents the execution times with input sizes varying from 8GB to 80GB. The left y-axis represent the execution times for OMR-VR, OMR-FX, and Metis and the right y-axis represent execution times for Hadoop. As we can see, OMR-VR and OMR-FX do not fail even when they process datasets larger than main memory. In contrast, Metis crashes as soon as the dataset size is increased; in fact apart from WC and II, Metis was unable to process 8GB datasets for other benchmarks.

Since Metis operates in memory, we observed that the crashes occurred during the map phase itself as the intermediate data produced grew beyond main memory; the memory overhead due to its hashmap of B+ trees data structure significantly limits the size

Table 2.4: Speedups achieved by OMR over Hadoop: OMR refers to OMR-VR for benchmarks WC, II, SC, RII, AL and SJ and OMR-FX for benchmarks MR and DC.

Input Size		**Benchmarks**							
		WC	**II**	**SC**	**RII**	**MR**	**DC**	**AL**	**SJ**
8 GB	**Hadoop (sec)**	2876	2006	2455	1638	5474	3815	6436	2175
	OMR (sec)	71	78	807	718	693	1010	1553	1326
	Speedup	40.6×	25.8×	3.04×	2.28×	7.90×	3.78×	4.14×	1.64×
16 GB	**Hadoop (sec)**	5844	3778	4907	3168	9346	7481	12932	4566
	OMR (sec)	142	379	1550	1195	1356	1992	2521	2652
	Speedup	41.3×	9.97×	3.17×	2.65×	6.90×	3.76×	5.13×	1.72×
32 GB	**Hadoop (sec)**	10953	10540	9162	5435	19104	12861	24288	8326
	OMR (sec)	616	728	2848	1977	2642	4035	4773	5293
	Speedup	17.8×	14.5×	3.22×	2.75×	7.23×	3.19×	5.09×	1.57×
48 GB	**Hadoop (sec)**	20733	20790	15344	7779	28527	20557	37261	12254
	OMR (sec)	1052	939	4522	2792	3854	6002	7044	8050
	Speedup	19.7×	22.1×	3.39×	2.79×	7.40×	3.43×	5.29×	1.52×
64 GB	**Hadoop (sec)**	24434	30383	19809	12937	38518	30864	52262	16989
	OMR (sec)	1169	1198	5304	2905	5074	8206	9032	10930
	Speedup	20.9×	25.4×	3.74×	4.45×	7.59×	3.76×	5.79×	1.55×
80 GB	**Hadoop (sec)**	33203	43821	27152	14330	47755	40008	63763	22745
	OMR (sec)	1519	1459	6699	3287	6105	10201	11087	12893
	Speedup	22×	30×	4.05×	4.4×	7.82×	3.92×	5.75×	1.76×

of intermediate data that Metis is able to process. Hadoop, on the other hand was able to process all the datasets successfully by writing intermediate results on disk. However, for

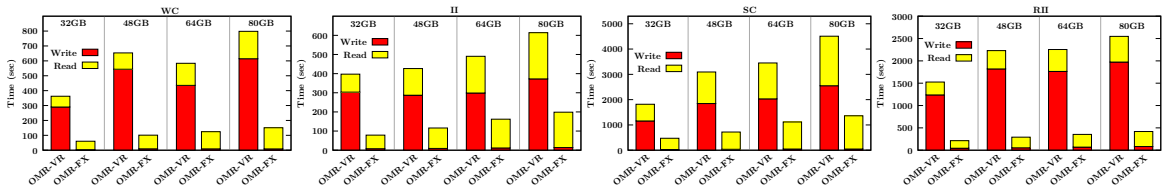


Figure 2.3: Read and write times in seconds.

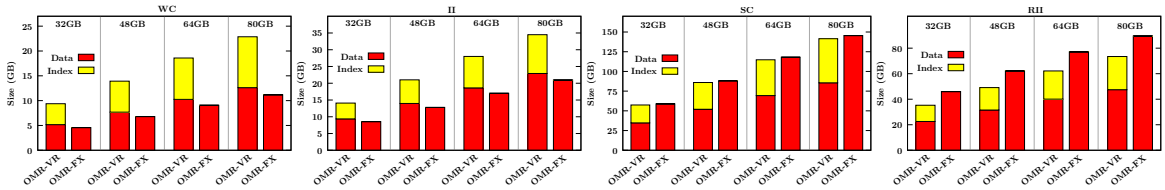


Figure 2.4: Size of intermediate files on disk.

each benchmark executed, it showed a slowdown between 8.7-21.8× in the processing speed with the increase in the input size from 8GB to 80GB.

While Hadoop crashed under the default settings, it was able to process the inputs after tuning a number of parameters. Since most of the crashes occurred during shuffle and sort phase, when the data is read/written from/to the disk to/from in-memory buffers, `mapred.child.java.opts` was varied up to 15536MB and also `mapreduce.task.io.sort.mb` was varied between 100-500MB for each different input to allocate more heap space to the map and reduce tasks and to increase the in-memory buffer size during sorting of the intermediate data respectively. Also, `mapreduce.map.output .compress` was also tuned (`True`/`False`) to shrink the map outputs, thereby reducing the shuffle time and making disk spills faster by writing less amount of data. We also tuned `io.file.buffer.size` (4-128KB), `mapreduce.reduce.shuffle.input.buffer.percent` (0.7-0.8), `mapreduce.input.fileinputformat.split.minsize` (0-256MB), `mapreduce.reduce.shuffle.parallel-`

Table 2.5: Execution times (in seconds) on small datasets. OMR refers to OMR-VR for benchmarks WC, II, SC, RII, AL and SJ and OMR-FX for benchmarks MR and DC. ✗ [k GB] means that the dataset could not be processed due to out-of-memory crashes and the size of the largest dataset that could be processed is k GB.

Dataset	Version	WC	II	SC	RII	DC	MR	AL	SJ
1GB	OMR	8.51	8.60	16.17	35.07	118.62	45.51	121.79	27.83
	Metis	20.74	22.26	33.66	21.40	24.13	46.86	28.45	✗ [0.94GB]
	Hadoop	275	173	230	95	508	670	666	241
2GB	OMR	15.88	17.79	36.54	68.71	236.09	97.19	242.37	50.01
	Metis	44.88	48.50	✗ [1.7GB]	45.32	13.88	✗ [1.9GB]	61.87	✗ [0.94GB]
	Hadoop	511	292	436	188	883	1366	1684	511
4GB	OMR	30.40	35.11	78.05	133.25	469.00	204.40	487.17	467.52
	Metis	95.00	98.73	✗ [1.7GB]	94.36	✗ [3.8GB]	✗ [1.9GB]	✗ [2.5GB]	✗ [0.94GB]
	Hadoop	1081	629	928	334	1780	2923	3406	1036

`copies` (5-10) to bring down the execution times. However, even after changing these parameters and gaining better performance, as shown in Table 2.4, OMR greatly outperforms Hadoop across all the benchmarks.

In Table 2.4, we compare the performance of OMR and Hadoop for the different input datasets (8GB-80GB). Apart from MR and DC which used OMR-FX, the speedup of all the other benchmarks has been calculated using OMR-VR. For 16GB (which equals the size of available memory) dataset OMR-VR shows speed up of 41.3× over Hadoop for WC benchmark since OMR is able to process the entire 16GB input in memory while Hadoop had to spill intermediate data on disk due to its in-memory data-structure and Java runtime overheads. For other benchmarks where the 16GB datasets are processed using disk, OMR performs 1.7-9.9× faster than Hadoop. With datasets larger than 16GB, OMR outperforms

Hadoop by up to 25.4× since OMR primarily achieves minimal disk I/O via partial ordering as opposed to Hadoop (as explained in Section 2.2).

The performance of OMR-VR and OMR-FX scales linearly with input size. With a 10× increase in dataset size, the increase in execution times is between 4.6-10.3× for all benchmarks except WC and II. For WC (II), processing of datasets up to 16GB (8GB) happens entirely in memory and hence, its linear scale begins after 16GB (8GB). We observe plateaus for OMR-VR in WC and RII at 48-64GB which represent faster execution for 64GB datasets; this is because the I/O times for 64GB datasets in those benchmarks do not increase as sharply from 48GB datasets, as shown in Figure 2.3.

We also compare the raw file sizes for intermediate data generated by map phase in Figure 2.4. As we can see, the total size of data records read/written to disk for WC and II is similar for OMR-FX and OMR-VR; however, OMR-VR additionally maintains the index information which increases the overall amount of disk I/O that is performed. Even though record sizes are large in SC and RII (see Table 2.3) due to keys representing multiple words and filenames, the absence of index information offsets the increase in key sizes, making the overall file sizes (including index information) comparable for OMR-FX and OMR-VR. It is interesting to note that even though file sizes for RII are slightly higher for OMR-FX than OMR-VR, Figure 2.3 shows significant reduction in disk I/O time for RII using OMR-FX.

Furthermore, as seen for WC, II, SC, and RII, OMR-FX consistently outperforms OMR-VR while processing using disks. This is because the batch writing process for OMR-VR requires InfiniMem to create an index that captures the variable record sizes. This

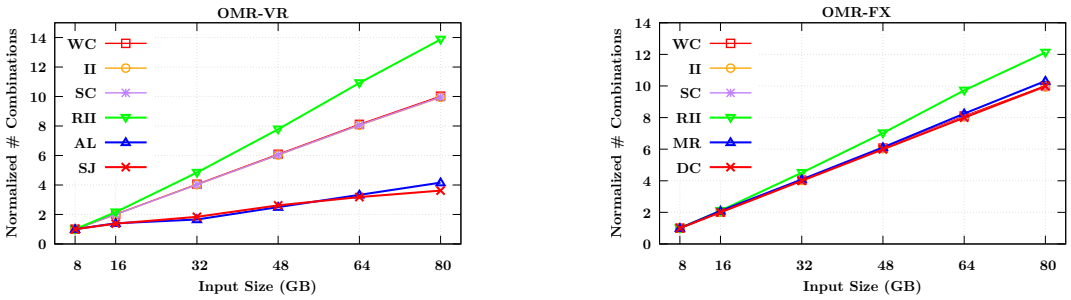


Figure 2.5: On-the-fly aggregation during map phase.

process significantly increases the overall batch writing time for InfiniMem compared to that for OMR-FX (see Figure 2.3). While such index maintenance also impacts the batch reading process, the increase in reading time is less significant since parsing the records requires a single read-only pass over the index information.

Finally, Figure 2.5 compares the on-the-fly aggregations that take place for OMR-VR and OMR-FX. As the input size increases, the number of values combined during the map phase steadily increase. Since combinations in AL and SJ occur by simply joining the value vectors in `<key, [value]>` pairs, the aggregation does not lead to reduction in memory footprint that is as large as that in other benchmarks. This means, the in-memory buffers for AL and SJ are flushed off to disk more often during the map phase, which in turn limits the possibility of aggregating subsequent `<key, value>` pairs. Hence, the increase in number of combinations is slower for AL and SJ compared to other benchmarks.

Impact of Record Sizes: Figure 2.6 shows the impact of record size on performance for WC on 48GB dataset by increasing the fixed size records up to 100×, from 40 bytes to 4K bytes. As we can see, the execution time increases linearly with intermediate record size. This is because the I/O cost incurred in OMR is linear in total size of `<key, value>` pairs

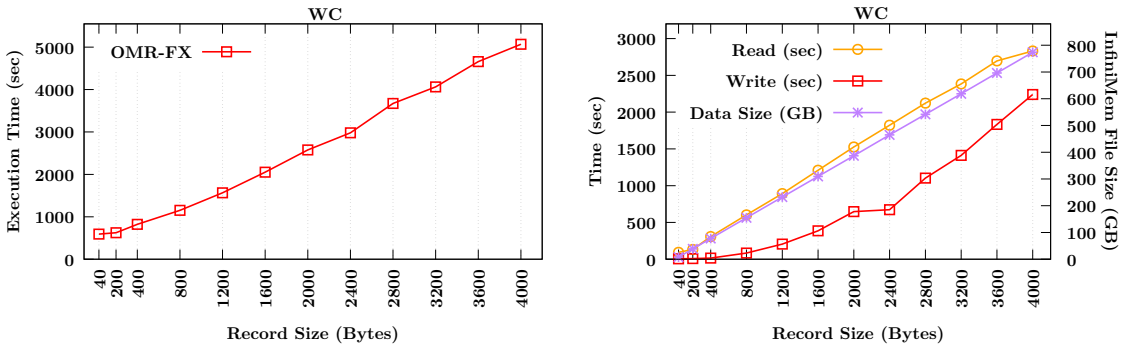


Figure 2.6: Varying record size.

generated by `map()` function. Since record size has a direct impact on the overall data size, the InfiniMem file sizes also increase linearly with record size. This in turn increases the read and write times, as shown in Figure 2.6.

Comparing the execution time with OMR-VR in Figure 2.2, the performance while using fixed size records is better for record sizes up to 400 bytes. This gives a large enough window for leveraging from OMR-FX as words or tuples of consecutive words typically do not grow this large.

Comparison on Small Datasets: While OMR scales well on large datasets, we compared its performance with Hadoop and Metis on datasets that are small enough to be processed on 16GB main memory. Table 2.5 shows the performance over various datasets on OMR-VR (OMR-FX for DC and MR), Metis, and Hadoop. As we can see, OMR-VR (OMR-FX) can efficiently process small datasets in memory; in fact, OMR-VR (OMR-FX) outperforms Hadoop across all benchmarks, and compares favorably with Metis. This is because of our highly concurrent architecture using simple ordered buffers which enables fast on-the-fly aggregation. Metis's hash table and B+ tree based data structure forces it to perform multiple memory accesses per `<key, value>` pair which increases it's execu-

tion time. Hadoop which is originally designed for the distributed systems, on the other hand, is able to fully process all the small datasets as opposed to Metis because it spills data on disk as needed. The execution times for RII, DC and AL are higher for OMR-VR (OMR-FX) compared to Metis since we employ line-by-line processing (to adhere with original MapReduce programming standard) which is slower for input files that only contain short lines. Metis, on the other hand, allows `map()` function to directly process very large partitions of file, hence eliminating the overheads.

Finally, it is important to note that nine out of twenty four cases crashed for Metis because it required more main memory. On the other hand, processing for OMR-VR (OMR-FX) successfully finished in memory for all cases, except for SJ on 4GB dataset where the runtime automatically resorted to using InfiniMem due to constrained memory. For benchmarks that Metis could not process, Table 2.5 shows the largest dataset size that could be handled by Metis; as we can see, the largest size varies across benchmarks due to variations in characteristics like key and value sizes, frequencies, etc.

2.6 Summary

In this chapter, we presented an out-of-core MapReduce system that can process datasets far larger than the size of the memory. It achieves this via - 1) Memory Constrained Processing, 2) Sequential Block Disk Accesses via Ordered Records, and 3) I/O Reduction via On-the-fly aggregation. Our evaluation results show that in contrast to OMR, both Metis [14] an in-memory MapReduce system and a standalone single-machine setup of the Hadoop [1] system either experience out-of-memory crashes or poorer performance.

Moreover, even when datasets are small enough to fit in main memory, OMR outperforms Hadoop across all benchmarks while its performance compares favorably with Metis.

Chapter 3

Out-of-core Graph Partitioner

Graph analytics is employed in many domains to gain insights by analyzing large graphs representing entities and interactions among them. Real world graphs often contain millions of vertices and billions of edges (see Table 3.1), and iterative graph analytics queries require multiple passes over the graph until convergence. Therefore, there has been a great deal of interest in developing scalable graph analytics systems that exploit parallelism available on distributed systems [28, 30, 38, 39]) as well as shared memory systems [40, 43]. In this chapter, we describe an effiicent single-level out-of-core graph partitioner that can partition large graphs in just two passes.

3.1 Background and Motivation

Before parallel or distributed analytics on large graphs can be performed, they typically must first be partitioned. In context of distributed systems the graph is partitioned across multiple machines such that each machine is primarily responsible for computations

Table 3.1: Input Graphs of Varying Sizes: Flickr (FL), PokeC (PK), LiveJournal (LJ), Orkut (OK), UKdomain2002 (UK02), Wikipedia-eng (WK), Twitter-WWW (TW), Twitter-MPI (TM), and UKdomain-2007 (UK07). [35, 60]

Graph G	**Vertices** $\mid V \mid$	**Edges** $\mid E \mid$	**Graph Size** $\mid E \mid + \mid V \mid$	$\frac{\mid E \mid}{\mid V \mid}$
FL	1,715,255	15,551,249	17.3 million	9.1
PK	1,632,803	30,622,564	32.3 million	18.7
LJ	4,036,537	34,681,189	38.7 million	8.6
OK	3,072,441	117,185,083	120.3 million	38.1
UK02	18,520,486	261,787,258	280.3 million	14.1
WK	12,150,976	378,142,420	390.3 million	31.1
TW	41,652,230	1,202,513,195	1,244.2 million	28.9
TM	999,999,987	1,614,106,343	2,614.1 million	1.6
UK07	105,153,952	3,301,876,564	3,407.0 million	31.4

that operate on its assigned partition. In context of a single shared-memory machine, the graph is partitioned to enable *out-of-core* or disk-based processing of a large graph on a single machine [13, 44, 45, 47]. When the graph is too large to fit in the memory available on the machine, it is divided into smaller partitions and stored on disk. This enables partitions to be loaded one at a time into memory and processed. For superior performance, partitioning algorithms endeavor to create partitions that are well balanced and minimize edgecuts (i.e., number of edges that cross partition boundaries).

Although the problem of graph partitioning is known to be NP-hard [27], highly effective *multilevel graph partitioning* algorithms have been developed and are widely used [31, 32, 46]. A large graph goes through a series of coarsening phases which, by merging vertices, produces significantly smaller graphs at each subsequent level. Once a sufficiently

small graph is obtained, it is partitioned. Next this partitioning is projected to the coarsened graph at the next level to obtain its partitioning that is further refined using the Kernighan-Lin algorithm [34] to reduce edgecuts. This preceding step is repeated through the levels eventually producing a partitioning for the original full graph. Many implementation frameworks for multilevel graph partitioning are available for distributed systems (e.g., ParMetis [33], Scotch [41], KaFFPa [42]). These frameworks enable *end-to-end processing* of large graphs on distributed systems as both partitioning and subsequent analytics tasks can be performed on the same distributed platform.

Although a framework that implements multilevel graph partitioning on a single shared-machine, called Mt-Metis [36, 37], is also available, it does not enable *end-to-end* processing of large graphs on a single machine because Mt-Metis requires that not only the input graph be held in memory but also the coarsened graphs. Since the out-of-core processing of large graphs is required in the first place because the entire graph does not fit in memory, executing Mt-Metis to partition such a graph fails as it runs out of memory. Therefore current out-of-core graph processing systems employ very simple partitioning schemes to distribute vertices among partitions [44]. Consequently, end-to-end processing of large graphs on a single machine using sophisticated partitioners is an open problem.

3.1.1 Limitation of Multilevel Partitioning

Given a graph $G = (V, E)$, the goal of *k-way* graph partitioning is to partition V into k non-empty disjoint subsets $V_1, V_2, \cdots, V_k$. In general all vertices and edges in the graph have a weight but to simplify the discussion let us assume all weights are one. The quality of resulting partition is measured by its *balance* and *edgecuts*. Balance is defined as

$k\frac{MAX_i(|V_i|)}{|V|}$, which is ideally 1, and when it exceeds 1, the greater the value the greater is the degree of imbalance. Edgecuts corresponds to the number of edges that connect vertices from different partitions. A partitioning with lower edgecuts is preferred.

The multilevel graph partitioning scheme employed by Mt-Metis [36, 37] has three phases. The *Coarsening Phase* transforms the given graph G into a sequence of smaller graphs $G_1, G_2, \cdots, G_L$ such that $|V| > |V_1| > |V_2| > \cdots > |V_L|$. The *Partitioning Phase* generates k-way partitioning P_L of G_L. The *Uncoarsening Phase* projects P_L back to partitioning P of G going through intermediate partitions $P_{L-1}, P_{L-2}, \cdots, P_1$.

A detailed evaluation of Mt-Metis carried out by Lasalle and Karypis [36] shows that in comparison to ParMetis [33] and Scotch [41] which are both distributed implementations, the memory requirements of Mt-Metis is significantly lower. In fact, the memory requirements of Mt-Metis are only slightly higher than the serial implementation KMetis [32]

Table 3.2: Comparison of serial implementation KMetis with the multithreaded Mt-Metis in terms of the number of Edgecuts, Memory Consumption (GB) and Execution Time (sec). 8 partitions produced for each input graph on a machine with $425GB$ main memory.

Input	KMetis			Mt-Metis		
Graph	**Edgecuts**	**Mem**	**Time**	**Edgecuts**	**Mem**	**Time**
FL	3,371,075	1.70	18.0	3,974,999	2.30	6.0
PK	4,351,130	2.20	25.0	4,196,212	4.00	8.0
LJ	7,377,230	4.5	58.0	7,563,933	6.4	15.0
OK	24,257,372	11.10	103.0	24,163,859	22.40	32.0
UK02	2,107,793	10.4	62.0	2,241,490	15.4	34.0
WK	45,803,435	25.5	269.0	44,993,565	55.1	121.8

– the extra memory is needed primarily for thread local data structures and it increases with the number of threads. Moreover, Mt-Metis is optimized to work for irregular graphs [37]. Table 3.2 compares Mt-Metis and KMetis in terms of edgecuts, peak memory used, and execution times on $425GB$ machine.

Multilevel algorithms generally have a high memory requirement because in addition to holding G in memory, the coarsened graphs $G_1, G_2, \cdots, G_L$ must also be held in memory. While the original graph size is $|E| + |V|$, the cumulative graph size of coarsened graphs is $\sum_{i=1}^{L} |E_i| + |V_i|$. Thus, the combined size of the original graph and the coarsened graphs is $1 + \sum_{i=1}^{L} \frac{|E_i|+|V_i|}{|E|+|V|}$ times the size of the original graph. We collected the values of this ratio for the sample graphs in Table 3.1 by running Mt-Metis and the results are presented in Table 3.3. All experiments in this paper were performed on a machine with 32 cores (2 sockets, each with 16 cores) with Intel Xeon Processor E5-2683 v4 processors, 425 GB memory, 1TB SATA Drives, and running CentOS Linux 7. We observe that the ratio varies from **4.8**× to **13.8**× for the first six graphs. The number of levels of coarsening L is also given for each run. We could not collect the data for the three largest graphs because, on the machine used, Mt-Metis ran out of memory.

Next we present GO, a two-phase algorithm that does not employ multilevel partitioning, and can partition the larger graphs even when provided with memory less than what is needed to hold the original graph.

Table 3.3: Given Original Graph of Size $\mid E \mid + \mid V \mid$ and $(+L)$ Coarsened Graphs Generated by Mt-Metis: Ratio is the times by which Cumulative Graph Size of Mt-Metis is Greater than the Original Graph Size.

Input Graph	**Ratio**: $1 + \sum_{i=1}^{L} \frac{\mid E_i \mid + \mid V_i \mid}{\mid E \mid + \mid V \mid} \times$ **Coarsened Graph Levels**: (+L)					
	k=2	k=4	k=8	k=16	k=24	k=32
FL	9.8× (+6)	11.5× (+8)	12.2× (+9)	12.9× (+10)	13.5× (+11)	13.5× (+11)
PK	6.0× (+4)	6.9× (+5)	8.3× (+7)	8.2× (+7)	8.9× (+8)	8.9× (+8)
LJ	10.9× (+5)	13.0× (+7)	12.9× (+7)	13.7× (+8)	13.7× (+8)	13.8× (+8)
OK	9.3× (+5)	10.2× (+6)	10.1× (+6)	10.9× (+7)	10.1× (+6)	10.1× (+6)
UK02	4.8× (+6)	4.9× (+8)	4.9× (+8)	4.9× (+9)	4.9× (+9)	4.9× (+9)
WK	7.0× (+6)	8.0× (+8)	7.9× (+8)	8.2× (+9)	8.2× (+9)	8.2× (+9)
TW, TM, UK07	✗	✗	✗	✗	✗	✗

3.2 GO Overview

We present GO, an Out-of-core Graph Partitioner, that given a fixed amount of memory on a machine, successfully partitions large graphs that cannot be held in the given amount of memory. GO performs just two passes over the entire input graph, the *partition creation pass* that creates *balanced* partitions and the *partition refinement pass* that reduces *edgecuts*. Both passes are designed to function in a memory constrained manner. During the *partition creation* phase parallel threads read the graph from disk and assign vertices, along with their adjacency lists, to different partitions. Once the available memory is full, the subpartitions created thus far are written to disk to free up the memory. Thus, the threads can resume reading the remainder of the graph from disk and partitioning it. This process produces an initial partitioning that resides on disk. Next the *partition refinement* phase reads portions of all partitions into memory, refines these subpartitions against each other using the KL-algorithm [34], and maintains the refined partitioning. This process is

repeated until entire initial partitioning has been read and processed to generate the final refined partitioning.

Since GO performs partitioning without creating coarsened graphs, it is a *single level* partitioner with greatly reduced memory requirement. In other words, given a graph that can be held in available memory, GO can partition the graph without requiring the initial partitioning to be written to and then read back from disk during refinement. That is, entire partitioning can be performed in-memory. On the other hand, since Mt-Metis requires additional memory to hold coarsened graphs, total memory that it requires to hold all graphs is several times (4.8× to 13.8×) the memory needed to simply hold the original graph. Thus, Mt-Metis cannot partition the graph successfully even if enough memory is available to hold the entire input graph in memory. The quality of partitions produced by GO were found to be superior to those produced by Mt-Metis both in the terms of *balance* and *edgecuts*. This is due to careful design of initial partitioning algorithm and our modified application of KL-algorithm.

Our experiments with GO prototype on nine input graphs of varying sizes show that GO can successfully partition large graphs for which Mt-Metis runs out of memory on the machine used. For graphs that can be successfully partitioned by **Mt-Metis** on a single machine, **GO** produces balanced 8-way partitions with 11.8× to 76.2× fewer edgecuts using 1.9× to 8.3× less memory and comparable runtime.

3.3 GO: Out-of-core Graph Partitioner

Given a limited amount of memory that cannot even hold the input graph in adjacency list format, GO uses the given memory to form memory buffers that are used by multiple threads to perform parallel graph partitioning in two phases: *Initial Partition Creation*; and *Partition Refinement to Create Final Partitions*. Both phases are designed to function in *memory constrained* manner, i.e. they successfully execute using the given memory that cannot hold the large input graph.

Figure 3.1 provides an overview of GO. The input graph is stored on disk in adjacency list format. The available memory is organized as multiple in-memory buffers. During the first phase threads read the graph from disk in parallel, assigning vertices to create balanced initial partitions and accumulating their adjacency lists in the same buffers. When a buffer is full, it is written to disk so that processing of rest of the graph can continue. At the end of the first phase, the initial partitions are created and each partition is written to disk (Infinimem object store [11]) as an ordered sequence of batches. In the second

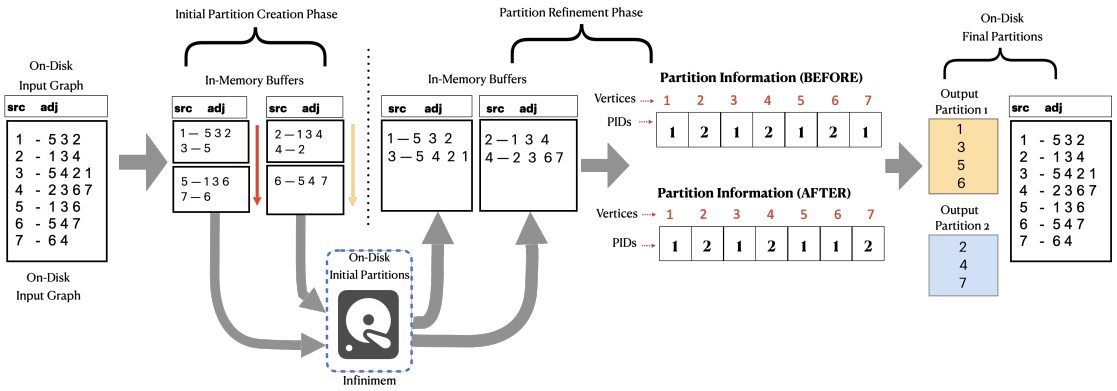


Figure 3.1: Overview of Out-of-Core GO Graph Partitioning.

phase once again the in-memory buffers are created, one for each partition. Portions of partitions from Infinimem are read into these buffers until the buffers are full, and then the in-memory portions of the partitions are refined against each other. Once initial partitions of the graph have gone through the in-memory buffers, the final refined partitions are available and written to the disk. Very high degree vertices are treated differently from other vertices during the above phases to obtain partitioning with good balance and low edgecuts.

Note that if the buffers are large enough to accommodate the entire graph during the first phase, then the initial partitions are not written to disk and re-read for refinement. Instead the refinement is also fully carried out in memory and final partitions are finally output to the disk. In subsequent subsections we present each of these phases in detail.

3.3.1 Memory Constrained Initial Partition Creation

For k-way partitioning of graph G, the memory available to the GO partitioner is divided into k buffers $B_1, B_2 \cdots B_k$, one for each partition. The graph is read using blocked serial reads from disk. Each source vertex v is assigned to some partition P_i, and the vertex v and its adjacency list $Adj(v)$ are stored in buffer B_i. By using a simple hashing function for assigning partitions to source vertices, a *balanced* partitioning is ensured. The above process is repeated as long as there is room in the buffers to accommodate more of the graph. Once some buffer B_* is full, its contents are written to the disk, that is, B_* is emptied and processing of the graph is resumed. Emptying of B_* is referred to as writing of a *batch* to disk. When the entire graph has been processed, all buffers are emptied and the partitioned graph is available on disk. On disk the graph is now organized according to *partitions*, where each partition is made up of series of *batches*. Consequently, after

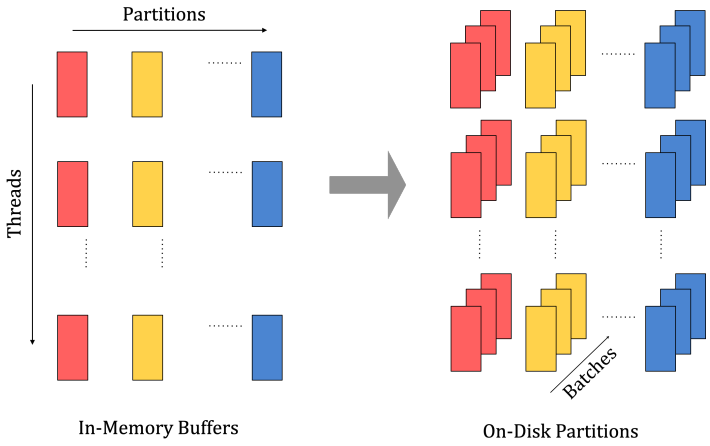
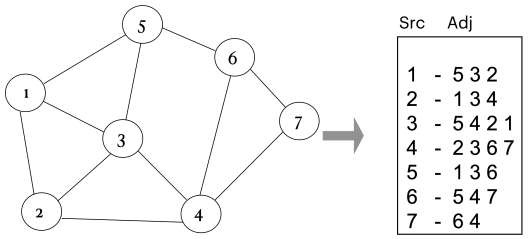


Figure 3.2: Organizing Memory into Buffers and Disk Usage.

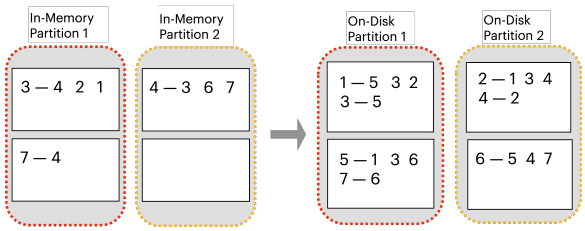
this phase, the next phase (refinement) can read each partition using blocked serial reads of its batches from the disk. While the partitioned graph is written to disk, an auxiliary in-memory PID array, indexed by vertex id, remembers the partition ids of all vertices.

Parallelization To parallelize the I/O and processing, we use t threads that read from disk in parallel and create the partitions in parallel. To ensure that the t threads do not have to synchronize with each other when updating the buffers, each buffer is subdivided into t sub-buffers, one for each thread. This leads to the organization of the graph as shown in Figure 3.2 where the number of in-memory buffers is $t \times k$. Corresponding to each of $t \times k$ buffers, the disk contains a series of batches that are written to disk when the buffers are emptied.

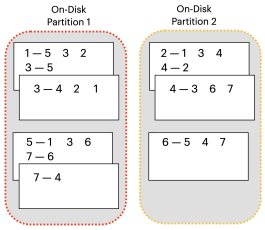
Vertex-Ordered Representation Note that the source vertices in the original graph, and corresponding adjacency lists, are organized on disk in the order of source *vertex ids*. Thus, they are read in the order of source vertex ids by the above partitioning phase, they



(a) Example Input Graph.



(b) Partitioning in progress: In-Memory & On-Disk.



(c) Completed Initial Partitioning.

Figure 3.3: Representation of example graph in memory and on disk where $k = 2$ and $t = 2$.

appear in the order of source vertex ids in B_*'s, and thus appear in order inside each batch written to disk. To illustrate this phase let us consider the example shown in Figure 3.3. In Figure 3.3(a) an example graph and its adjacency list representation is shown. Let us assume that we are carrying out 2-way partitioning using 2 threads and hence the memory is organized into 4 buffers. Figure 3.3(b) shows the representation of the graph where all of the graph has been read and processed, part of it has been written to the disk in its batched partition form while part resides in memory buffers. Here each buffer has been emptied once. Once the buffers are emptied, the completed initial partitioning of the graph that resides on the disk is shown in Figure 3.3(c). Note that the contents of in-memory buffers and on-disk batches are all ordered according to vertex ids of source vertices in them.

Split Adjacency Lists for Irregular Graphs In out-of-core processing by systems such as GraphChi [13] and GridGraph [47], the graph partitions are represented as a subset of edges from the graph. The edges present in the subgraph corresponding to a partition are essentially contained in the partition's representation. Balancing edges across partitions is important because the work performed by analytics tasks (e.g., single source shortest paths) is proportional to the number of edges. Since power-law graphs, due to their irregular skewed degree distribution, contain vertices with very high degrees, we obtain a balanced partitioning by distributing the edges incident to such vertices across the partitions. This can be carried out simply by adding a threshold parameter δ such that when source vertex v is processed by a thread, after placing δ edges in its current partition, the partition is changed causing the next δ edges to be placed in a different partition. This approach enables the edges for vertex v to be split across all k partitions in batches of δ.

Note that it is also possible for the adjacency list of some vertex v to get split because the buffer to which the list is being written to has become full. This can be observed in Figure 3.3(c) where vertex 7 is assigned to partition 1 and its adjacency list consisting of two vertices, 6 and 4, are split across two consecutive batches on disk. However, note that in this kind of splitting, the adjacency list is not split across partitions but rather it is split across batches that belong to the same partition. Moreover, as we will show later, when partitions are read from their batches in the refinement phase, these split adjacency lists will be merged together.

3.3.2 Memory Constrained Partition Refinement

For a k-way partitioning, the refinement phase is executed in parallel by k threads where each thread is assigned the task of refining a single partition. The memory available is divided into k buffers of equal size. All threads (T_is), in parallel, load as much of the subgraphs (wP_is) of their respective assigned initial partitions (P_is) into their buffers ($Buffer(T_i)$s). After refining the loaded wP_i's against each other, the buffers are loaded again and refined. This process continues until the entire graph has been refined and final partitioning has been found.

Parallel Loading When buffer $Buffer(T_i)$ is loaded from partition P_i the vertices assigned to P_i are loaded in increasing order of vertex id. This is achieved by performing a merge sort across batches written to P_i by different threads. Note that when vertices and their adjacency lists are loaded into $Buffer(T_i)$ in this fashion, if the adjacency list of a vertex v in P_i had been split across different batches due to emptying of filled buffers in the initial partitioning phase, then the spilt adjacency list of v will get merged during loading. Consequently, after loading, the contents of $Buffer(T_i)$ are organized as follows. A low degree vertex v_l that is assigned to a unique partition P_i, appears in $Buffer(T_i)$ along with $Adj(v_l)$, its *complete* adjacency list. A high degree vertex v_h whose edges are spread across all k partitions appears in each $Buffer(T_*)$ along with the *complete* subset of its adjacency list assigned to each partition P_*.

Refinement The goal of refinement using the Kernighan-Lin (KL) algorithm [34] is to reduce edgecuts in the existing partitioning by swapping pairs of vertices between partitions.

Doing so does not alter the *balance* of the partitions that is achieved during the initial partitioning phase, the swap operations are chosen merely to reduce edgecuts. The initial partitioning is held in memory in PID[*] array where using vertex id to index the array, we can read the partition id of the vertex. When swap operations are applied, PID[*] contents are modified to reflect the change in partitioning.

Algorithm 4 Interval-based KL Algorithm

1: $P_1 \cdots P_k$ – Subgraphs Representing k Partitions
2: PID[*] – Initial Partition Ids of Vertices
3: NUMINTS – Number of Intervals
4: **for all** threads $T_i \in \{T_1, \cdots, T_k\}$ **do**
5: **do**
6: ▷ Load Partitions
7: Read Subgraph wP_i of P_i into BUFFER(T_i)
8: merge sorting over source vertex ids causing
9: split adjacency lists to be merged
10: Identify Boundary Vertices in wP_i using PID[*]
11: Divide Boundary Vertices into NUMINTS Intervals
12: ▷ Refine Partition wP_i against partitions wP_j, $j > i$
13: REFINEPARTITION ($wP_i \subseteq P_i$)
14: **while** (P_i is exhausted)
15: **end for**

Algorithm 5 Refine Partition Function using KL Algorithm

```
1: ▷ Refine Partition Pairs
2: function RefinePartition( wP_i )
3:     for wP_j = wP_{i+1} ··· wP_k do
4:         ▷ Refine wP_i wrt wP_j
5:         for interval id x = 1 ··· numInts do
6:             ▷ refine interval pair (I_x^i ∈ wP_i, I_x^j ∈ wP_j)
7:             loop ▷ Making a Pass
8:                 (g, (v, w)) ← FindMaxGainPair(I_x^i, I_x^j)
9:                 break when g = 0
10:                Add (v, w) to SwapSet
11:            forever
12:            ▷ Update Partitions using SwapSet
13:            for each (v, w) ∈ SwapSet do
14:                Swap ( Pid[v], Pid[w] )
15:            end for
16:        end for
17:    end for
18: end function
```

Algorithm 6 Finding Pair with Maximum Gain using KL Algorithm

```
1: ▷ Find Pair of Vertices to Swap
2: function FindMaxGainPair(I_x^i, I_x^j)
3:     MaxGain ← 0; MaxPair ← null
4:     for v ∈ I_x^i do
5:         for w ∈ I_x^j do
6:             D(v) ← EC(v)\I_x^j − IC(v)\I_x^i
7:             D(w) ← EC(w)\I_x^i − IC(w)\I_x^j
8:             if D(v) ≥ 0 ∧ D(w) ≥ 0 then
9:                 ThisGain ← D(v) + D(w) − 2 × Edge(v, w)
10:            end if
11:            if ThisGain > MaxGain then
12:                MaxGain ← ThisGain;
13:                MaxPair ← (v, w)
14:            end if
15:        end for
16:    end for
17:    return (MaxGain, MaxPair)
18: end function
```

Given a pair of vertices (v, w) from two different partitions, Pid$[v]$ and Pid$[w]$, the decision to swap v and w between the two partitions is based upon the extent to which the swap reduces edgecuts, which is also called the Gain. The *external cost* of v with respect to Pid$[w]$ (i.e., $EC(v)\backslash PID[w]$) is the number of edges from v to vertices in Pid$[w]$. The *internal cost* of v, $IC(v)$, is the number of edges from v to vertices in Pid$[v]$. If the difference

between external and internal costs of v and w, $D(v)$ and $D(w)$, are both positive, then swapping of the vertices will definitely reduce edgecuts. The GAIN is the sum $D(v)$ and $D(w)$ if v and w are not directly connected by an edge; otherwise it is $D(v) + D(w) - 2$.

When refining a pair of partitions against each other, all pairs of vertices are considered and the one that provides the highest Gain are chosen for swapping. This process is repeated – each application is called a *Pass* that finds one pair to swap – until no more pairs with positive Gain are available.

Taming KL-algorithm's Complexity for Large Graphs In large graphs with millions of vertices in each partition, it is not practical to consider every pair of vertices from every pair of partitions and consider them for swapping. To reduce the complexity we first observe that only *boundary vertices* – vertices that have edges cut by the initial partition – for a pair of partitions need to be considered during refinement. To further lower the complexity of refinement we limit the scope of refinement by dividing boundary vertices belonging to each partition into equal-sized NUMINTS *intervals*. When refining two partitions with respect to each other, only all pairs of vertices from corresponding intervals in the two partitions are considered. This greatly reduces the pairs considered and hence the cost of refinement. Note that high-degree vertices are not included in such pairs as, having partitioned their edges across partitions, their is no benefit from swapping them. The low degree vertices are plentiful in a large irregular graph and hence refinement of intervals that are smaller subgraphs is still highly effective as will be observed from our experimental results for GO in comparison to Mt-Metis.

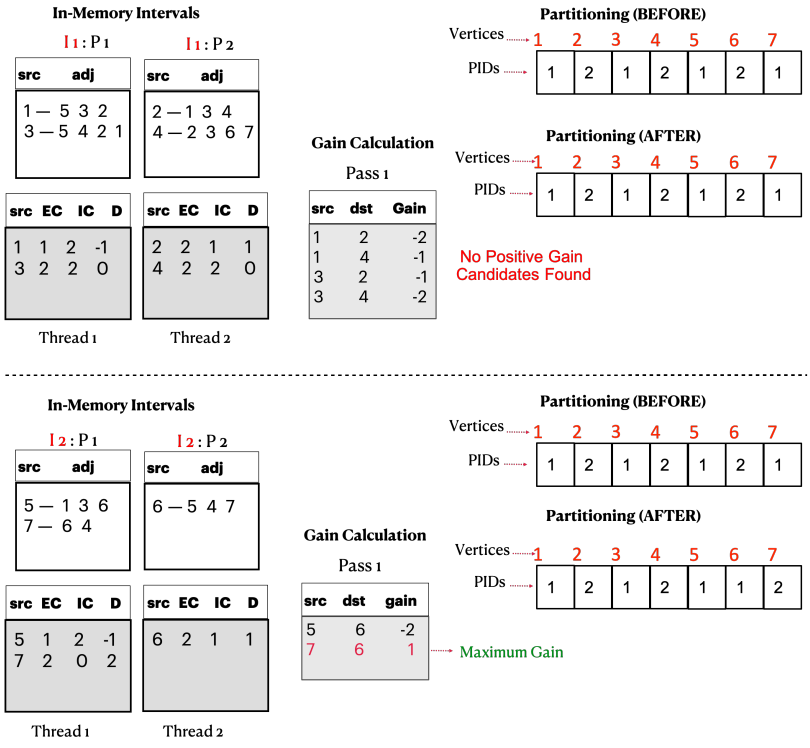


Figure 3.4: Illustration of Refinement Algorithm.

Interval-based Algorithm and Example Algorithm 4 presents Interval-based algorithm of our refinement phase. Lines 4-15 show threads loading portions of partitions into buffers in parallel, identifying boundary vertices, dividing them into intervals, and calling RefinePartition to refine partitions. Lines 2-18 in Algorithm 5 show the function including how a partition refines itself against all other partitions (Line 3), considering pairs of intervals (Line 5), making multiple *passes* (Lines 7-11), and finally applying atomic Swap operations that update partition ids stored in array Pid. When considering a pair of intervals, function FindMaxGainPair defined in Algorithm 6 is used to consider all relevant vertex pairs in those intervals and finds the pair with the highest gain and returns that pair.

Figure 3.4 illustrates the above algorithm. We assume two partitions with two intervals each. The top part of the figure refines the first intervals from both partitions against each other and does not swap any vertices as no vertex pair with a positive gain is found after a pass. The bottom part refines second intervals from both partitions and this time the first pass identifies a vertex pair with positive gain and in the second pass (not shown) none is found. The figure shows how the partitioning as expressed in PID[*] array changes. The EC, IC, and D values of the source vertices belonging to intervals are shown. The gains computed in each pass are also shown. However, note that the tables that show these are for illustration as no tables are maintained by the algorithm, only the pair with maximum gain is remembered.

3.4 GO Prototype and its Evaluation

GO is implemented in C++ and it uses the Infinimem I/O runtime [11] to leverage its support for seamless batch disk I/O for variable size records. The records are expressed using Protocol Buffers [3] which provide efficient serialization/deserialization. The graph is divided among multiple threads for parallel processing where each thread reads its assigned part of the graph and puts it into the in-memory buffers. The graph is represented in memory using the adjacency list format and written to disk by the threads to which vertices are assigned using Infinimem's batch I/O.

The goal of our evaluation is to compare the quality and cost of graph partitioning as performed by GO and Mt-Metis. We study the scalability of GO with increasing graph size and varying memory availability. For modest sized graphs that could be successfully

partitioned by both GO and Mt-Metis, we compare the partitions produced by both the systems in terms of number of edgecuts, balance and peak memory used in handling graphs of varying sizes.

The graph partitioning experiments were performed on a machine with 32 cores (2 sockets, each with 16 cores) with Intel Xeon Processor E5-2683 v4 processors, 425 GB memory, 1TB SATA Drives and running CentOS Linux 7. Table 3.1 includes graphs of varying sizes ranging from 15.5 million edges to 3.3 billion edges. This allows us to compare the scalability of partitioning algorithms and demonstrate that eventually for large graphs Mt-Metis runs out of memory while GO can successfully partition them even when only given part of the memory is available on the machine used in the experiments.

Our evaluation considers following partitioning algorithms:

- GO-100 corresponds to the amount of memory so the graph fits in memory and initial partition is not written to and re-read from disk;
- GO-75, GO-50, and GO-25 correspond to running GO with 75%, 50% and 25% of graph in memory. GO will need to write and re-read the initial partition from disk.
- Mt-Metis is a *multithreaded* version of Metis [36]. We compare GO's performance with Mt-Metis 0.6.0 that incorporates enhanced coarsening scheme [37] for graphs with highly variable degree distribution (e.g., power-law).

Table 3.4: Number of Edgecuts for GO-100 and Relative Number for Mt-Metis and Other GO Configurations.

k	Algo.	Input Graphs								
		FL	**PK**	**LJ**	**OK**	**UK02**	**WK**	**TW**	**TM**	**UK07**
8	**GO-100**	337,908	173,738	452,443	316,918	113,265	655,057	5,837,535	112,048	6,760,828
	Mt-Metis	11.8×	24.2×	16.7×	76.2×	19.8×	68.7×	✗	✗	✗
	GO-75	2.4×	0.8×	0.4×	1.1×	1.4×	0.5×	1.5×	1.6×	0.6×
	GO-50	2.4×	1.3×	0.5×	1.4×	1.2×	0.9×	1.6×	2.5×	0.7×
	GO-25	2.8×	1.7×	0.6×	2.4×	2.2×	1.1×	1.7×	4.3×	0.6×
16	**GO-100**	1,145,151	366,450	493,762	2,640,637	649,569	1,923,263	14,249,266	1,078,944	10,342,321
	Mt-Metis	4.2×	15.2×	20.3×	12.5×	4.0×	28.9×	✗	✗	✗
	GO-75	2.1×	1.3×	0.9×	0.4×	0.4×	0.8×	1.9×	0.9×	0.5×
	GO-50	2.1×	1.5×	1.1×	0.4×	0.3×	1.0×	1.9×	0.8×	1.1×
	GO-25	2.2×	2.1×	1.4×	0.6×	0.7×	1.1×	1.9×	1.0×	1.0×
24	**GO-100**	2,469,269	1,672,698	2,260,283	9,278,356	869,908	6,232,550	22,486,575	3,923,882	13,110,702
	Mt-Metis	2.1×	3.8×	5.3×	3.8×	3.3×	10.3×	✗	✗	✗
	GO-75	2.0×	0.5×	0.4×	0.2×	0.6×	0.6×	1.7×	0.2×	1.8×
	GO-50	2.0×	0.7×	0.5×	0.2×	0.4×	0.7×	1.7×	0.3×	1.8×
	GO-25	2.1×	0.7×	0.6×	0.3×	1.1×	0.8×	1.8×	0.6×	1.7×

3.4.1 Quality of Partitions: Edgecuts and Balance

We demonstrate that GO can successfully partition all graphs in Table 3.1 using varying amounts of memory that holds 100%, 75%, 50%, or 25% of the graph. Since different configurations have different amounts of memory available, the partitionings they produce are different. We compare the quality of partitions produced using Mt-Metis with the various GO configurations. Table 3.4 presents *edgecuts* data while Table 3.5 presents the *balance* data for partitionings produced. For Mt-Metis no data is presented for the three

Table 3.5: *Balance* of Partitions – GO Configurations vs. Mt-Metis: Values are $MAX_i(|\ V_i\ |)$ as a percentage of $|\ V_i\ |$. The ideal balance percentage for 8, 16 and 24 partitions is 12.50%, 6.25% and 4.17% respectively.

k	Algo.	Input Graphs								
		FL	**PK**	**LJ**	**OK**	**UK02**	**WK**	**TW**	**TM**	**UK07**
8	**GO-100**	12.50%	12.50%	16.12%	12.50%	12.50%	12.50%	12.50%	12.51%	12.50%
	Mt-Metis	13.76%	13.70%	16.65%	13.70%	12.54%	14.61%	✗	✗	✗
	GO-75	12.50%	12.50%	16.12%	12.50%	12.50%	12.50%	12.50%	12.50%	12.50%
	GO-50	12.51%	12.51%	16.12%	12.51%	12.50%	12.50%	12.50%	12.51%	12.50%
	GO-25	12.52%	12.52%	16.12%	12.51%	12.50%	12.50%	12.50%	12.51%	12.50%
16	**GO-100**	6.25%	6.25%	8.06%	6.25%	6.25%	6.25%	6.25%	6.26%	6.25%
	Mt-Metis	7.62%	6.76%	8.76%	7.21%	6.28%	6.67%	✗	✗	✗
	GO-75	6.25%	6.25%	8.06%	6.25%	6.25%	6.25%	6.25%	6.26%	6.25%
	GO-50	6.27%	6.27%	8.07%	6.26%	6.25%	6.25%	6.25%	6.26%	6.25%
	GO-25	6.30%	6.31%	8.08%	6.29%	6.26%	6.26%	6.25%	6.26%	6.25%
24	**GO-100**	4.17%	4.17%	5.37%	4.17%	4.17%	4.17%	4.17%	4.17%	4.17%
	Mt-Metis	4.86%	4.82%	5.70%	4.48%	4.19%	5.14%	✗	✗	✗
	GO-75	4.17%	4.17%	5.37%	4.17%	4.17%	4.17%	4.17%	4.17%	4.17%
	GO-50	4.21%	4.23%	5.40%	4.19%	4.17%	4.18%	4.17%	4.17%	4.17%
	GO-25	4.22%	4.24%	5.41%	4.24%	4.18%	4.19%	4.17%	4.17%	4.17%

largest graphs (TW, TM and UK07) because it ran out of memory in the coarsening phase and terminated (indicated by ✗ in the tables).

Edgecuts From number of edgecuts given in Table 3.4, we observe that the number of edgecuts produced by all GO configurations are typically far fewer than Mt-Metis, i.e. irrespective of the amount of memory available to GO. This is because the interval size employed by the KL algorithm during the refinement is similar for all the GO configurations. The

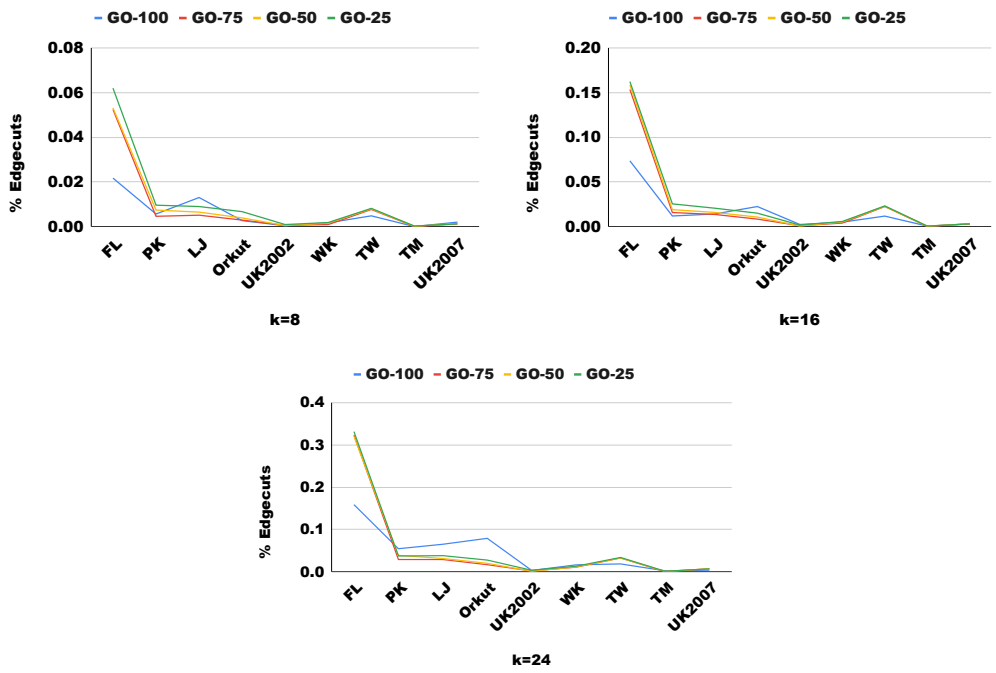


Figure 3.5: Edgecuts as a percentage of total number of edges for GO configurations.

variation in edgecuts for different GO configurations is due to the difference in refinement intervals caused by batches merged during the *partition refinement phase*. Mt-Metis creates 8-way/16-way/24-way partitioning with 11.8/4.2/2.1× to 76.2/28.9/10.3× more edgecuts than GO-100. And this is in spite of GO-100 using much less memory as, being a single-level algorithm, it does not create coarsened graphs. As expected, the percentage of edges that are cut increases with number of partitions.

When we look at the ratio of edgecuts for MT-Metis and GO-100, we also make another observation. The higher the $\frac{|E|}{|V|}$ ratio for the input graph, the higher is the edgecuts ratio of Mt-Metis over GO-100. Among the six graphs that are successfully handled by Mt-Metis, the three largest $\frac{|E|}{|V|}$ ratios are 38.1 for OK, 31.1 for WK, and 18.7 for PK. Their

Table 3.6: Peak Memory in **GB** for GO configurations vs. Mt-Metis.

k	Algo.	FL	PK	LJ	OK	UK02	WK	TW	TM	UK07
8	**Mt-Metis**	2.30	4.00	6.4	22.40	15.4	55.1	✗	✗	✗
	GO-100	0.61	0.72	1.7	2.70	8.0	7.6	29.8	72.9	79.1
	GO-75	0.27	0.31	0.8	0.96	3.3	2.5	10.0	52.0	26.5
	GO-50	0.26	0.27	0.6	0.93	2.6	2.4	9.8	46.0	26.1
	GO-25	0.14	0.16	0.4	0.48	1.6	1.5	6.4	40.4	13.4
16	**Mt-Metis**	2.40	2.70	7.8	20.50	16.2	53.1	✗	✗	✗
	GO-100	0.67	0.78	1.9	2.80	8.6	8.0	31.2	104.2	82.4
	GO-75	0.37	0.38	0.9	1.00	3.8	2.9	11.0	83.0	29.0
	GO-50	0.31	0.32	0.8	0.99	3.2	2.8	10.3	77.8	28.0
	GO-25	0.23	0.22	0.6	0.52	2.2	1.9	7.9	71.4	17.2
24	**Mt-Metis**	2.60	3.70	8.1	17.40	15.9	47.1	✗	✗	✗
	GO-100	0.45	0.50	2.1	2.90	9.2	8.4	32.6	135.6	85.7
	GO-75	0.41	0.45	1.0	1.10	4.2	3.3	12.0	114.0	32.0
	GO-50	0.38	0.40	0.9	1.00	3.8	3.1	11.5	109.0	31.0
	GO-25	0.35	0.30	0.8	0.64	2.8	2.1	8.8	104.0	19.3

Mt-Metis over GO-100 edgecut ratios are the worst – 76.2× for OK, 68.7× for WK, and 24.2× for PK. In other words, GO is more effective in dealing with irregular graphs.

Finally, Figure 3.5 presents the percentage of edges that are cut by partitioning. As we can see, the percentage is generally greater for smaller graphs than for larger graphs, the edgecut percentage increases with the number of partitions.

3.4.2 Memory Usage

Table 3.6 shows the peak memory used by GO and Mt-Metis for varying the number of partitions. The peak memory is noted using the *top* utility on Linux. For the first six graphs, since Mt-Metis could successfully partition the graphs, we calculate the ratio between peak memory used by Mt-Metis versus that used by GO configurations.

From the data in Table 3.6 we observe the following. As expected, GO-100 is far more memory efficient than Mt-Metis as GO is a single level algorithm while Mt-Metis is a multilevel algorithm that must additionally hold coarsened graphs in memory. For first six graphs, GO-100 uses 1.9× to 8.3× lesser amount of memory than Mt-Metis for an 8-way partitioning. For producing greater number of partitions (i.e., 16 and 24), both GO-100 and Mt-Metis require greater amount of memory although this increase is modest (less than 10%). For the largest three TW, TM, and UK2007 graphs, GO-100 uses 29.8-32.6GB, 72.9-135.6GB, and 79-85.7GB of memory respectively. Given that typically Mt-Metis uses many times the memory used by GO-100, it is not surprising that Mt-Metis runs out of memory and fails to partition the graphs.

The $\frac{|E|}{|V|}$ ratio impacts the memory used by the algorithms, much more so for Mt-Metis than for GO-100 resulting in the following observations. Although the sizes of WK and UK2002 are fairly close, their $\frac{|E|}{|V|}$ ratios are quite different – 31.1 for WK versus 14.1 for UK2002. This impacts the relative memory usage of Mt-Metis and GO. MT-Metis uses 6.6× for WK and 1.9× for UK2002 in comparison to GO-100. This is because the coarsened graphs for Mt-Metis contain greater numbers of edges for WK than for UK2002 causing greater need for additional memory by WK than UK2002.

The memory used by GO-25 is at least four times less than the memory used by GO-100. The exception is the TM input graph for which GO-100 uses around 1.5× to 2× the memory used by GO-25. This is because TM has the smallest $\frac{|E|}{|V|}$ ratio of 1.6 and hence edges account for smaller fraction of memory needs. In fact the auxiliary array for holding partition ids of vertices accounts for significant fraction of memory. Thus, reducing buffer memory impacts peak memory consumption far less than for all other graphs.

3.4.3 Execution Times

Table 3.7 shows the execution time in seconds for different GO configurations - GO-100, GO-75, GO-50, GO-25, and Mt-Metis. The execution times of GO-100 compare well with Mt-Metis and better for graphs with higher $\frac{|E|}{|V|}$ ratio (PK, OK WK) with speedups ranging from 1.3× to 1.7×. For the two larger graphs the results are quite different – for UK2002 the execution times for Mt-Metis are roughly 2× faster than GO-100 while for WK, Mt-Metis runs 1.6× slower. Nevertheless, GO-100 produces partitioning with fewer edgecuts than Mt-Metis for both UK2002 and WK.

Balance of partitions Next we examine how well balanced are the partitions that are produced. We present the percentage of vertices that belong to the largest partition in a partitioning in Table 3.5. Note that the ideal percentage for best balance is 12.5% for 8-way, 6.25% for 16-way, and 4.17% for 24-way partitioning. We observe that for six graphs – FL, PK, OK, WK, TW, and UK2007 – GO-100 produces almost perfect partitioning. On the other hand, with Mt-Metis the largest partitions vary from 12.54% to 16.65% in size.

Table 3.7: Execution Times in Seconds for GO Configurations vs. Mt-Metis.

k	GO-Mem Config.	Input Graphs								
		FL	**PK**	**LJ**	**OK**	**UK02**	**WK**	**TW**	**TM**	**UK07**
8	**GO-100**	7.4	6.2	18.9	18.2	70.9	71.9	397.1	1,722.0	1,026.0
	Mt-Metis	6.0	8.0	15.0	32.0	34.0	121.8	✗	✗	✗
	GO-75	7.5	6.5	19.1	20.1	83.4	75.1	426.8	1,794.0	1,057.0
	GO-50	7.9	6.6	19.2	20.2	84.1	78.0	450.1	1,814.0	1,073.0
	GO-25	8.0	8.2	19.4	20.4	87.6	79.9	453.3	2,068.0	1,113.0
16	**GO-100**	7.2	5.8	17.4	17.5	65.1	66.2	396.1	1,763.0	965.2
	Mt-Metis	7.0	7.0	19.0	34.0	34.0	123.6	✗	✗	✗
	GO-75	7.4	6.9	17.7	19.1	65.7	69.1	405.2	1,789.0	1,020.0
	GO-50	8.2	8.2	18.5	20.4	68.3	70.6	412.5	1,811.0	1,049.0
	GO-25	9.4	8.5	19.2	21.0	69.5	74.1	431.9	1,929.0	1,094.0
24	**GO-100**	7.4	6.1	18.6	19.9	60.4	69.0	424.2	1,869.0	1,028.0
	Mt-Metis	6.0	8.0	17.0	30.0	35.0	126.6	✗	✗	✗
	GO-75	7.9	6.5	19.1	21.2	64.8	71.7	425.1	1,911.0	1,075.0
	GO-50	8.2	6.6	19.1	21.9	70.6	73.7	435.4	1,937.0	1,080.0
	GO-25	9.0	7.5	19.2	29.0	71.9	79.2	462.3	1,948.0	1,095.0

Thus, the partitions generated by GO-100 are more balanced and have fewer edgecuts than partitions produced by Mt-Metis. When GO is given less memory, balance of partitions is not adversely impacted but only very slightly. The highly balanced partitions produced by GO is due to its superior initial partitioning phase. This is because the swapping of vertices performed during refinement, reduce edgecuts without altering the balance of partitions.

Table 3.8: I/O Time in Seconds for GO Configurations.

k	GO-Mem	Input Graphs								
	Config.	FL	PK	LJ	OK	UK02	WK	TW	TM	UK07
8	GO-100	0 (4.7)	0 (3.7)	0 (11.9)	0 (8.9)	0 (47.0)	0 (42.9)	0 (200.5)	0 (1187.0)	0 (478.4)
	GO-75	1.02 (5.1)	0.6 (4.2)	3.1 (12.4)	1.2 (9.6)	14.5 (50.6)	5.4 (46.1)	26.6 (493.3)	50.5 (1281.5)	68.6 (499.9)
	GO-50	1.52 (8.7)	1.2 (5.0)	5.0 (12.8)	1.5 (10.7)	15.3 (60.7)	8.2 (47.7)	34.2 (509.8)	53.7 (1473.7)	82.5 (523.3)
	GO-25	2.63 (6.3)	1.4 (5.7)	5.4 (14.7)	1.6 (11.9)	20.8 (61.4)	8.7 (53.8)	45.4 (572.6)	69.2 (1502.8)	47.9 (622.9)
16	GO-100	0 (4.6)	0 (3.5)	0 (11.0)	0 (8.5)	0 (40.4)	0 (34.8)	0 (173.5)	0 (1216.4)	0 (413.6)
	GO-75	1.23 (4.6)	0.41 (5.5)	2.1 (11.4)	1.0 (9.0)	10.7 (41.4)	5.1 (42.3)	26.6 (514.8)	31.9 (1224.6)	49.0 (441.0)
	GO-50	1.33 (5.8)	0.82 (5.7)	3.9 (33.7)	1.6 (10.9)	12.7 (42.1)	7.2 (42.9)	34.2 (541.7)	39.0 (1409.2)	62.4 (497.2)
	GO-25	1.42 (6.0)	0.83 (6.8)	4.2 (14.1)	1.7 (11.7)	17.5 (45.6)	8.4 (44.2)	45.4 (584.7)	77.0 (1471.0)	64.7 (683.8)
24	GO-100	0 (4.6)	0 (3.8)	0 (11.2)	0 (9.2)	0 (33.4)	0 (34.8)	0 (163.3)	0 (1179.7)	0 (302.5)
	GO-75	1.04 (5.2)	0.3 (3.9)	1.3 (11.9)	0.9 (13.9)	5.0 (36.6)	6.4 (122.2)	10.0 (452.4)	36.1 (1257.9)	24.8 (355.3)
	GO-50	1.44 (6.2)	0.7 (4.2)	3.4 (12.6)	1.4 (17.4)	6.4 (37.3)	4.0 (120.3)	15.7 (533.4)	49.4 (1463.1)	42.0 (365.0)
	GO-25	1.84 (7.4)	0.9 (5.5)	3.6 (14.3)	1.5 (19.3)	7.9 (45.7)	6.2 (138.4)	17.1 (534.8)	87.0 (1499.7)	46.0 (391.1)

The runtimes of GO scale with graph size ranging from few seconds for the smallest graph to roughly 32 minutes for large graphs. Let us compare the execution times of GO-100 with GO-25. We observe that typically GO-25 exceeds the execution times of GO-100 by less than 50%. For the largest graph of UK2007 with over 3.3 billion edges, the execution time of GO-25 exceeds that of GO-100 by 8.5%, 13.4%, and 6.5% for 8-way, 16-way, and 24-way partitionings. Thus, we see that the runtimes of GO scale well with graph size, number of partitions, and amount of memory available to run.

Finally, we observed that the I/O times of GO-100 differ from the I/O times of GO-25 only by a small amount because the I/O performed by the out-of-core feature is highly efficient. Table 3.8 presents the I/O times of writing/reading of initial partitioning to/from Infinimem is small compared to rest of the I/O time (numbers in parenthesis in the table) for reading the initial graph and writing out the final partitioning.

3.4.4 GridGraph Performance vs. GO Partitioning

Next we study the impact of partitions produced on runtimes of graph algorithm on a state-of-the-art out-of-core system. We executed two graph algorithms, **PageRank** and **WCC** (*Weakly Connected Components*), on the GridGraph [47] out-of-core system using the partitions produced by GO-100, GO-75, GO-50 and GO-25. The execution times of GridGraph are given in Table 3.9. As we can seen from this table, the execution times of all the GO configurations are comparable. In fact, the runtimes are same for most of the GO configurations. In other words, reducing memory to 25% does not result in any slowdowns on GridGraph as the quality of partitions produced does not change significantly from GO-100 to GO-25. Therefore we can conclude that the quality of partitions produced by GO

Table 3.9: Scalability of GO Configurations: Execution Times in Seconds for PageRank and WCC on GridGraph.

k	Part. Algo.	PageRank			WCC		
		OK	WK	TW	OK	WK	TW
8	**GO-100**	6	19	128	3	7	48
	GO-75	7	22	134	4	8	48
	GO-50	7	21	128	4	7	49
	GO-25	8	20	126	5	7	44
16	**GO-100**	6	21	96	4	7	37
	GO-75	7	23	100	6	8	39
	GO-50	8	20	100	5	7	39
	GO-25	7	21	102	6	7	38
24	**GO-100**	8	16	100	5	6	33
	GO-75	7	19	110	6	7	34
	GO-50	8	20	99	6	8	37
	GO-25	7	18	109	7	7	40

is fairly insensitive to the aount of memory provided to GO. The above results are to be expected because, as we reduce the memory available to GO, the number of edgecuts does not change significantly.

In order to compare the effectiveness of GO's partitioning with Mt-Metis partitioning and other simple partitioning schemes, we ran the workloads of *PageRank* and *WCC* on GridGraph. We compare the running times of GridGraph on OK, WK and TW input graphs for multiple partitioning schemes. In addition to Mt-Metis and GO, we also collected the running times of GridGraph when using simple partitioning strategies, cyclic and block-

Table 3.10: Execution Times in Seconds for GO-100, Mt-Metis, Cyclic, and Block-Cyclic Partitionings on Medium Sized Graphs OK, WK and Large Graph TW for PageRank and Weakly Connected Components (WCC) on GridGraph.

k	Part. Algo.	PageRank			WCC		
		OK	WK	TW	OK	WK	TW
8	Mt-Metis	7	21	✗	4	8	✗
	GO-100	**6**	**19**	**128**	**3**	8	**48**
	Cyclic	8	**19**	133	5	8	50
	Block-Cyclic	8	24	130	5	**7**	49
16	Mt-Metis	9	22	✗	5	14	✗
	GO-100	**8**	**21**	**96**	**4**	**12**	**37**
	Cyclic	9	23	109	5	15	39
	Block-Cyclic	**8**	29	99	**4**	17	**37**
24	Mt-Metis	7	18	✗	**6**	9	✗
	GO-100	**6**	**16**	**100**	**6**	**7**	**33**
	Cyclic	7	20	101	8	9	44
	Block-Cyclic	7	21	105	9	11	35

cyclic, as these strategies have been used by existing out-of-core systems for evaluating graph queries to circumvent the memory intensive nature of graph partitioning. The results shown in Table 3.10 demonstrate that running times for GO-100 partitioning are same or less than that for Mt-Metis, Cyclic and Block-Cyclic partitionings with the exception of *WCC* for k=8. Note that in Table 3.10 the minimum execution times are shown in bold. This is to be expected because GO-100 always produces partitioning that is superior to the partitioning generated by other methods.

3.5 Summary

In this chapter we presented the GO out-of-core graph partitioner that can function within the memory constraints imposed by the machine and successfully partition graphs that far exceed the size of a graph that can be held in memory. The execution time and memory requirements scale well with the graph size. For graphs that can be successfully partitioned using the in-memory Mt-Metis graph partitioner, GO produces high quality partitioning in terms of edgecuts and balance. In contrast to multilevel partitioning scheme used by Mt-Metis, GO is single level and it partitions the graph in two highly memory efficient parallel passes. For graphs that can be successfully partitioned by Mt-Metis on a single machine, GO produces balanced 8-way partitions with 11.8 to 76.2 fewer edgecuts using 1.9 to 8.3 lesser memory in comparable runtime.

Chapter 4

OMRGx: Extended OMR for Graph Partitioning and Processing

We already discussed the growing popularity of single machine analytics systems and described existing systems for MapReduce and iterative Graph Analytics. In Chapter 2 we studied how a MapReduce system like OMR [48] can be efficiently used on a single machine to process huge datasets in parallel with the optimized IO operations when using disk. Such a framework enables programmers to easily express the processing logic using the simple APIs while runtime system transparently manages parallelism, memory, and IO. In Chapter 3, we studied how similar challenges can be addressed for partitioning large graphs while maintaining its balance and minimizing the edgecuts. The ease of expressing wide variety of algorithms using our highly optimized MapReduce system motivated us to address the following challenge - *using MapReduce system for partitioning and end-to-end processing of large graphs.*

In this Chapter, we present a *general purpose* MapReduce framework which supports *graph partitioning* and performs *partition-based graph processing*. We show the efficacy of our system by showing its programmability, performance and scalability by running the programmed applications on five input graphs of medium and large sizes.

4.1 MapReduce for Graphs

A common concern across graph processing systems is the nature of consistency semantics they offer for programmer to accurately express graph algorithms. Consistency semantics in the context of iterative graph processing fundamentally decide when should a vertex's value (that is computed in a given iteration) become visible to its outgoing neighbors. The most popular consistency semantics is offered by the Bulk Synchronous Parallel (BSP) [49] model (hereafter called synchronous processing semantics) that separates computations across iterations such that vertex values computed in a given iteration become visible to their outgoing neighbors in the next iteration, i.e., values in a given iteration are computed based on values from the previous iteration. Such clear separation between values being generated versus values being used allows programmers to clearly reason about the important convergence and correctness properties. Hence, synchronous processing semantics often becomes a preferred choice for large-scale graph processing [39, 47, 51, 52].

Originally, MapReduce systems were designed to work in a distributed manner where the workload was distributed amongst the multiple nodes. Therefore, expressing graph algorithms for distributed MapReduce systems will end up having to take multiple iterations of MapReduce which requires the output of one iteration to be fed as the input to

another. This, in turn, will slow down the entire process as data has to be written to files at the end of every *map*() and *reduce*() operation and moved between nodes. The entire state of the graph needs to be transmitted in each stage, which will require a lot more communication overhead. Bringing the MapReduce framework on a single machine where the vertices and edges remain on the machine that performs the computation mitigates the communication cost, unlike performing MapReduce operations on distributed systems where communication overhead between the nodes takes over the advantages of using MapReduce systems to express different computations. Therefore, MapReduce systems on a single machine can easily simulate the Bulk Synchronous Parallel (BSP) model by splitting the computations in parallel across multiple reducers which are responsible for performing the computation on the assigned vertices and then communicate through shared-memory structures and wait for all the reducers to finish the computations of the assigned tasks. This implies that we can exploit the features provided by the runtime to easily express graphs in a MapReduce model where a key can be specified as a vertex/node and the edges will be the value for that specific key. The graph algorithms can be easily expressed using the *map*() and *reduce*() functions where the vertex-centric computations can be performed based on the assigned vertices to different mapper and reducer threads, which can operate independently.

Another challenge in handling graphs is that they usually have a specific structure which makes them to be expressed using highly specialized systems. However, recent systems like OMR and Infinimem [11] have shown the applicability of Protocol buffers to further extend the capabilities of the system to express custom input formats which makes it easier to express the graph structure.

4.2 Extending OMR: An Implementation Choice

In the previous section, we discussed how a single machine MapReduce system can overcome the challenges in large scale processing of the graphs by simulating the BSP model and avoiding the communication costs that incur in a distributed environment. Such a system is already equipped with efficient engine to leverage parallelism, handle data efficiently in memory and perform optimized IO operations. Therefore, a *highly tailored* single machine MapReduce system becomes a good choice to partition and process large graphs that often need to be written off to disk due to their iterative nature and BSP model semantics. Hence, we can use the simple APIs provided by the MapReduce system to express the vertex-centric computations. Such a framework is also *ideal* to partition the graphs by mapping the data to different partitions and assigning each reducer the task of refining each partition. This *motivated* us to take a step forward and extend our existing MapReduce system - *OMR* to support Graph Partitioning and Processing.

Therefore, we present *OMRG*x, an out-of-core graph partitioning and processing system by extending OMR's API to support graphs. OMRGx allows programmers to express a wide variety of graph partitioners from simple partitioners like cyclic, block cyclic, hash to more sophisticated partitioners like GO [50], Mt-Metis [36, 37]. Programmers can then perform *partition-based* graph processing based on the partitioning strategy of their choice. *OMRG*x provides a simple yet all-inclusive API which makes it easier to program different graph *partitioning* algorithms: GO, MtMetis, hash and *processing* algorithms: GraphChi [13]. We further demonstrate the *versatility* of *OMRG*x by implementing different partitioners and graph processing frameworks: GO partitioner with GraphChi; simple

Hash partitioner with GraphChi; and MtMetis partitioner with GraphChi. *OMRG*x runs the PageRank algorithm, using the different graph partitioners and processing frameworks implemented in *OMRG*x, on medium and large sized input graphs by producing a number of partitions of the input graphs and performing the *partition-based* processing. This demonstrates the *scalability* of *OMRG*x. We also run the default processing of *OMRG*x to show how the framework itself can be used to process large graphs using simple *map*() and *reduce*() functions making users *oblivious* about Out-Of-Core.

4.3 The *OMRG*x Programming Interface

*OMRG*x provides a rich programming interface that allows programmers to be *oblivious* about Out-of-Core and easily express different graph partitioning and processing algorithms using simple *map*() and *reduce*() functions provided by the system. Programmers just need to specify the partitioning/processing logic and let runtime take care of the entire process of efficiently handling large amounts of data in memory and optimizing IO operations to store data on disk whenever needed. This gives programmers the flexibility to program different applications, as seen in Figure 4.1. We now describe *OMRG*xs simple yet all-inclusive application programming interface (API) that enables graph partitioning and processing.

Figure 4.2 shows the high-level API provided by *OMRG*x that allows programmers to express different graph partitioning strategies and also perform partition-based graph processing. This implies that the programmers can use *OMRG*x to express graph

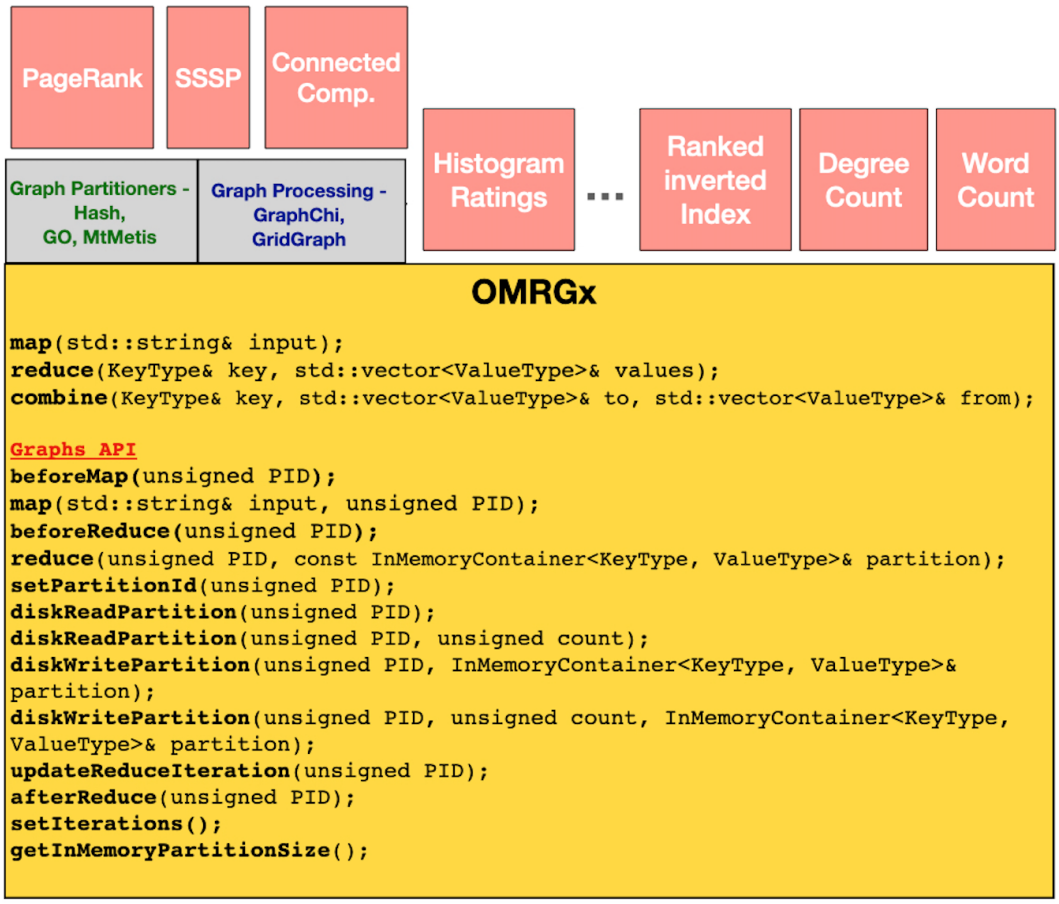


Figure 4.1: OMRGx APIs to support Graph Partitioning and Processing.

partitioning algorithms and partition-based graph processing algorithms. This does not limit the user to any specific partitioner when processing large graphs, rather provides the freedom to implement graph partitioning algorithm of their choice. In addition, the programmer can transparently make use of the out-of-core capabilities of the underlying OMR system.

```
template <typename KeyType, typename ValueType>
class Graph : public MapReduce<KeyType, ValueType>
{
 void* beforeMap(unsigned PID);
 void* map(std::string& input, unsigned PID, unsigned hiDegree);

 unsigned setPartitionId(unsigned PID);

 void* beforeReduce(unsigned PID);
 void* reduce(unsigned PID, InMemoryContainer<KeyType, ValueType>& partition);

 void* updateReduceIteration(unsigned PID);
 void* afterReduce(unsigned PID);

/* Returns entire container */
 InMemoryContainer<KeyType, ValueType>& diskReadPartition(unsigned PID);

/* Returns records specified by count in the container */
 InMemoryContainer<KeyType, ValueType>& diskReadPartition(unsigned PID, unsigned count);

/* Writes entire container */
 void diskWritePartition(unsigned PID);

/* Writes records specified by count */
 void diskWritePartition(unsigned PID, unsigned count, InMemoryContainer<KeyType, ValueType>& partition);

};


int main(int argc, char** argv)
{
 Graph<unsigned, unsigned> graph;

 graph.setInput(folderPath);
 graph.setMappers(nmappers);
 graph.setReducers(nreducers);
 Graph.setIterations(iterations);
 graph.init();
 …
 graph.run();

 return 0;
}
```

Figure 4.2: Programming with *OMRGx*: the *map*() and *reduce*() APIs are used for specifying the processing logic; *diskReadPartition* and *diskWritePartition* APIs used for storing the entire or part of the partition on disk.

4.3.1 Graph Partitioning Algorithms

In *OMRGx*, the programmer can implement their graph partitioning algorithm using the *map* and *reduce* APIs. The programmer specified *map* function is used to split the input into $key - value$ pairs. Here *keys* are represented by *vertices*/*nodes* and *values* represent *adjacencylist* for the vertex. The *setPartitionId* API is used to set the PID in memory for the input $key - value$ pair which is then passed to the *map* function to map the *key* to the specified partition. If during the *map* phase, the in-memory partitions reach

```
template <typename KeyType, typename ValueType>
class Hash : public MapReduce<KeyType, ValueType>
{
 void* map(std::string& input, unsigned PID, unsigned hiDegree){
   foreach(v, e)
     writeToBuffer(v, e, PID, hiDegree);
 }

 unsigned setPartitionId(unsigned PID){
   return hashKey(getkey(PID)) % nParts;
 }


 void* reduce(unsigned PID, InMemoryContainer<KeyType, ValueType>& partition)
{
   writeFinalPartition(partition);
 }
};
```

Figure 4.3: Hash partitioner programmed in *OMRG*x using its high-level API; showcasing the ease and versatility of programming with *OMRG*x.

```
template <typename KeyType, typename ValueType>
class Go : public MapReduce<KeyType, ValueType>
{
 void* map(std::string& input, unsigned PID, unsigned hiDegree){
     foreach(v, e)
       writeToBuffer(v, e, PID, hiDegree);
 }

 unsigned setPartitionId(unsigned PID){
     return hashKey(getkey(PID)) % nParts;
 }


 void* reduce(unsigned PID, InMemoryContainer<KeyType, ValueType>& partition){
      /* Refine this Partition with all other Partitions i */
     foreach(PID, PIDi){
         /* Compute Edgecuts */
       ComputeBEdgeCut(PID, PIDi, partition);

       /* Compute Gain using KL Algorithm */
       ComputeGain(PID, PIDi, partition)
     }
 }
};
```

Figure 4.4: GO partitioner programmed in *OMRG*x using its high-level API; showcasing the ease and versatility of programming with *OMRG*x.

```
template <typename KeyType, typename ValueType>
class MtMetis : public MapReduce<KeyType, ValueType>
{
 void* map(std::string& input, unsigned PID, unsigned hiDegree){
   foreach(v, e)
     writeToBuffer(v, e, PID, hiDegree);
 }

 unsigned setPartitionId(unsigned PID){
   return hashKey(getkey(PID)) % nParts;
 }

 void* reduce(unsigned PID, InMemoryContainer<KeyType, ValueType>& partition){
   coarsened_graph = coarsen(PID, partition);
   diskWritePartition(PID, count, InMemoryContainer<KeyType, ValueType>& coarsened_graph);

   init_partition(PID, coarsest_graph);
   foreach(coarsened_graph){
     refine_partition(PID, coarsened_graph);
     /* Reads records specified by count */
     finer_graph = diskReadPartition(PID, count);
   }
 }

};
```

Figure 4.5: MtMetis partitioner programmed in *OMRG*x using its high-level API; showcasing the ease and versatility of programming with *OMRG*x.

their capacity, the contents are serialized into a batch of contiguous records and written off to disk. The *reduce* function processes all the vertices in the partitions on disk in parallel based on the logic provided by the application programmer and emits the final partitions. This process repeats until the entire graph is read and processed.

Figure 4.3, Figure 4.4 and Figure 4.5 illustrate the different graph partitioners programmed in *OMRG*x. Hash partitioner, as demonstrated in Figure 4.3, is a simple partitioner which hashes the vertices into different partitions based on their keys during the *map* phase and simply writes out the final partitions during the *reduce* phase. On the other hand, GO and MtMetis (Figure 4.4 and Figure 4.5), being the sophisticated partitioners, refine the partitions during the *reduce* phase and hence perform additional processing. Once all the partitions are read from disk and refined during the *reduce* phase, the final partitions are available at the end which can be used for further processing by the graph algorithms.

Since MtMetis employs *multilevel* graph partitioning strategy where a graph is transformed into a sequence of smaller graphs during the *coarsening* phase. The *Partitioning* phase generates the initial partitioning of the coarsest graph. The *Uncoarsening* phase refines the partitions produced and projects them to their finer level graph and all the way to the original graph. Figure 4.5 shows the ease of programming *multilevel* algorithm using the simple APIs provided by *OMRG*x. If the input graph is large, MtMetis stores all its *coarsened* graphs on disk using *OMRG*x's $diskWritePartition$ API by specifying the *count* for the $key - value$ pairs to be stored on disk. During the *uncoarsening* phase, it simply reads the *finer* level graphs using the $diskReadPartition$ API provided by *OMRG*x.

4.3.2 Partition-based Graph Processing Algorithms

The graph processing algorithms tend to be iterative in nature, therefore, *OMRG*x provides $setIterations$ API to set the number of iterations for successful convergence and correctness. The $beforeMap$, $beforeReduce$ and $afterReduce$ APIs are used to set/clear the graph related structures. Based on the set number of iterations, the *reduce* operation is performed. The $updateReduceIteration$ API is used to update the computation data structures to prepare for the next iteration. For the iterative algorithms, the entire graph is read from or written off to disk multiple times using $diskReadPartition$ and $diskWritePartition$ APIs to propagate the updated values for the next iteration. The programmers have the choice to read/write the entire partition or a part of it based on their implementation.

As an example, Figure 4.6 shows the implementation of PageRank in *OMRG*x. During the *map* phase, vertex v and all its corresponding edges e are written off to buffers

in memory. The Protocol Buffers enable the use of the custom format to store the extra information like rank, number of neighbors etc along with the incoming edge e. If the partition to which this incoming $vertex-edge$ pair is written is not provided, then $OMRG$x chooses its default partitioning strategy which is *hashing* the $key-value$ pairs based on their vertex IDs. During the *reduce* phase, all the edges corresponding to each vertex are merged together by the reducer and available in the *partition*. Therefore, for each vertex v, all of

```
template <typename KeyType, typename ValueType>
class PageRank : public MapReduce<KeyType, ValueType>
{
 void* map(std::string& input, unsigned PID, unsigned hiDegree){
     foreach(v, e) /* Vertex v and Edge e */
       Edge e;
       e.src = e;
       e.dst = v;
       e.rank = 1.0/nvertices;  // nvertices - number of vertices
       e.vRank = 1.0/nvertices;
       e.numNeighbors = edges.size();
       writeToBuffer(v, e, PID, hiDegree);
 }

 unsigned setPartitionId(unsigned PID){
     /* set -1 to return default partitioning used by engine */
     return -1;
 }


 void* reduce(unsigned PID, InMemoryContainer<KeyType, ValueType>& partition){
     foreach(vertex v){
       // Process Neighbors
     }

    // Calculate rank
     float rank = (DAMPING_FACTOR * sum) + (1 - DAMPING_FACTOR);

     foreach(neighbor n){
      // update rank
         n->vRank = rank;
     }
 }

 void* updateReduceIter(const unsigned tid) {
   ++iteration;
 }
};
```

Figure 4.6: Graph Processing Algorithm - PageRank programmed in $OMRG$x using its high-level API; showcasing the ease and versatility of programming with $OMRG$x.

its neighbors are processed based on which the *pagerank* is calculated and then updated for all the neighbors of vertex v. The *updateReduceIter* API is used to increment to the next iteration. It is important to note that *OMRG*x internally uses its $diskWritePartition$ API to propagate the updated values for the next iteration. The entire process of reading and writing graphs from disk in parallel and optimized IO operations is *oblivious* to the application programmer.

4.3.3 Default Processing

*OMRG*x is extended from our *OMR* system which includes all the capabilities of the engine including the *mapping* of all the $key - value$ pairs and outputting the final $key - value$ pairs through the reduce phase. Therefore, the extended *OMRG*x system which includes the capabilities to partition and process large graphs can *itself* be used for the *default* processing without using any formal graph processing framework. Our experiments in Section 4.4 compares the performance of the *default OMRG*x system with the specialized Graph Processing frameworks like *GraphChi*.

4.4 Implementation and Evaluation

*OMRG*x is embedded in OMR and built by extending the API provided by OMR to support both graph partitioning and processing. It is implemented in C++. *OMRG*x can be used to *partition* large graphs and then perform the *partition-based* graph processing. Users can easily implement simple partitioners like cyclic, block-cyclic, hash etc as well as sophisticated partitioners like GO, MtMetis without worrying about the complex details

like memory used, parallelism, IO. Users can then feed these partitions to graph processing algorithms like pagrank, sssp to perform end-to-end processing of graphs. The entire process is seamless to the user.

We evaluate the programmability, scalability and performance of the *OMRG*x system. The evaluation is based on two classes of algorithms: graph partitioning and *partition-based* graph processing. We programmed different graph partitioning and processing algorithms using *OMRG*x's simple API. The *goal* of the experiments is to compare the programming effort, performance and scalability of the graph analytics systems implemented in *OMRG*x with their specialized standalone versions. Our evaluation considers following partitioning and processing algorithms:

- OMRGx-Hash, OMRGx-GO, and OMRGx-Mt correspond to the Hash, GO, and MtMetis partitioners implemented using OMRGx's API.
- GO-S and MtMetis-S refer to standalone implementation of GO and MtMetis.
- OMRGx-D and OMRGx-GC correspond to the OMRGx default and GraphChi implementations in OMRGx.
- GraphChi-S corresponds to running the standalone implementation of GraphChi.

4.5 Programmability

Writing graph partitioning and processing algorithms using OMRGx's API is simple. The programmer needs to only: (a) set the initial partitioning ID of the input vertices which is input to the *map* function, (b) specify the format (*adjacencylist* or *edges*) for

Table 4.1: Lines of Code needed to program the graph *partitioning* and *processing* algorithms in *OMRG*x vs the lines of code in the corresponding standalone systems.

Partitioning	**Framework**		**Processing**	**Framework**	
Algorithm	**OMRGx**	**Standalone**	**Algorithm**	**OMRGx**	**Standalone**
Hash	10		GraphChi	83	1323*
GO	176	1300	OMRGx-D	29	
MtMetis	200	22489			

the values to be stored corresponding to each *key*, (c) use the *default* processing engine or provide a *custom* processing algorithm to the *reduce* function. Table 4.1 quantifies the ease of programming with *OMRG*x by listing the lines of code for different graph *partitioning* and *processing* algorithms using OMRGx's APIs and compares it with their *standalone* implementations.

Programmability of Graph Partitioning Algorithms A simple hash partitioner can be implemented with a fewer lines of code. It is implemented with around 10 lines of code in OMRGx. As noted earlier, it does not need to refine the partitions during the *reduce* phase and hence, outputs the entire partition. The standalone version of GO is implemented with around 1300 lines of code whereas only 176 lines of code are needed to implement GO in OMRGx. This is because OMRGx's engine does all the heavy lifting with around 900 lines of code, all of which hides the complexity of parallelizing tasks, efficiently managing memory and optimizing IO from the user. The standalone implementation of MtMetis uses around $22,489$ lines of code just to express the coarsening and uncoarsening heuristics for the multilevel algorithm. On the other hand, this same algorithm can be programmed in

OMRGx with around 200 lines of code; rest of the complex details of handling out-of-core structures is managed by OMRGx.

Programmability of Graph Processing Algorithms Implementing GraphChi in OMRGx was relatively easy with less than 100 lines of code including loading memory and corresponding sliding shards, building subgraph in memory, and processing the subgraph to run *PageRank* algorithm. On the other hand, the standalone implementation of GraphChi is done with around 1323 lines of code[1]. It is important to note that the extra lines of code for preprocessing the shards, building subgraphs and meta data have not been added. OMRGx-D is the default processing provided by OMRGx which uses a hash partitioner to partition the $vertex - edge$ pairs based on their vertex ID and runs *PageRank* on the partitioned graph. It is implemented in about 29 lines of code.

Table 4.2: Input Graphs: Orkut (OK), Wikipedia-eng (WK), Twitter-WWW (TW), Twitter-MPI (TM), and UKdomain-2007 (UK). [35, 60] used in the evaluation.

Graph G	**Vertices** $\mid V \mid$	**Edges** $\mid E \mid$	**Graph Size** $\mid E \mid + \mid V \mid$	$\frac{\mid E \mid}{\mid V \mid}$
OK	3,072,441	117,185,083	120.3 million	38.1
WK	12,150,976	378,142,420	390.3 million	31.1
TW	41,652,230	1,202,513,195	1,244.2 million	28.9
TM	999,999,987	1,614,106,343	2,614.1 million	1.6
UK	105,153,952	3,301,876,564	3,407.0 million	31.4

[1]The lines of code provided in Table 4.1 for standalone implementation of GraphChi is just for the engine program which provides the code for processing the shards. The extra lines of code for preprocessing the shards, building subgraphs and meta data have not been added

4.6 Experimental Setup

All the experiments were performed on a machine with 32 cores (2 sockets, each with 16 cores) with Intel Xeon Processor $E5-2683\ v4$ processors, $425GB$ memory, $1TB$ SATA Drives, and running CentOS Linux 7. The input graph datasets consists of varying vertices and edges, listed in Table 4.2; ranging from medium sized graphs - OK, WK with around $120M$ - $378M$ edges to large sized graphs - TW, TM, UK consisting of edges between $1,244M - 3,407M$. UK is the largest graph at $55GB$ on disk, $3,407M$ edges and $105M$ vertices.

4.7 Performance

We now present the runtime performance of applications programmed with OMRGx in terms of number of partitions produced and the size of the input graph datasets. We compared the performance of algorithms programmed in *OMRG*x with their *standalone* versions.

First, we discuss the performance of graph partitioning algorithms programmed in OMRGx - OMRGx-Hash, OMRGx-GO in terms of number of partitions produced and the size of the input graph datasets. OMRGx-Hash is a simple partitioner implemented in OMRGx by hashing the vertices to different partitions while reading the input during the *map* phase and then it combines the $adjacency-lists$ for each vertex during the *reduce* phase to emit the final partitions without performing any refinement. On the other hand, OMRGx-GO and GO-S refine the reduced partitions before writing them off to disk. Therefore, the overall execution time for OMRGx-GO and GO-S is higher than OMRGx-Hash as seen in Figure 4.7.

It is interesting to note that the execution time for all the algorithms is comparable for the largest UK graph. This shows that the performance of the algorithms implemented in OMRGx is not adversely impacted with the increase in the size of input graph. Furthermore, the performance of OMRGx-GO compares well with the highly optimized standalone graph partitioner GO-S.

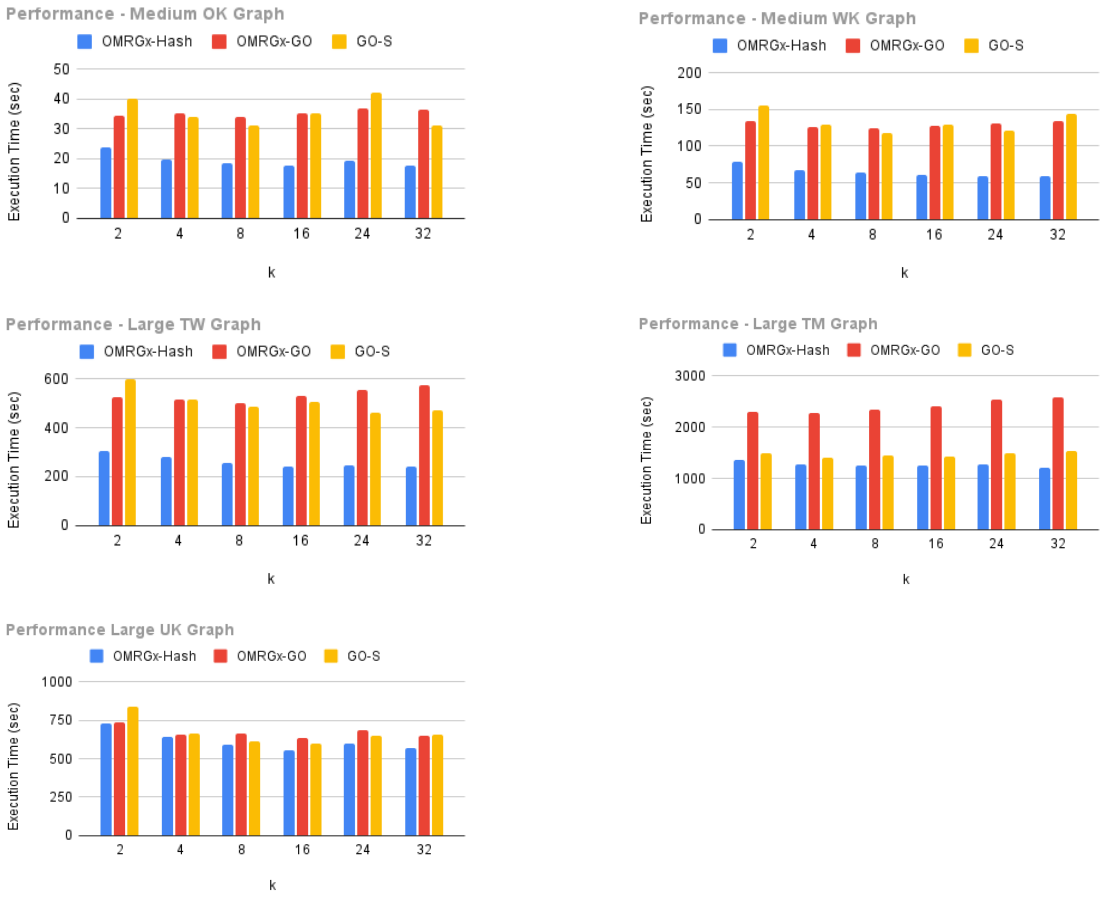


Figure 4.7: Comparison of Execution Times (seconds) for Graph Partitioning Algorithms implemented in *OMRG*x vs. their standalone implementations using input graphs of varying sizes and different number of partitions.

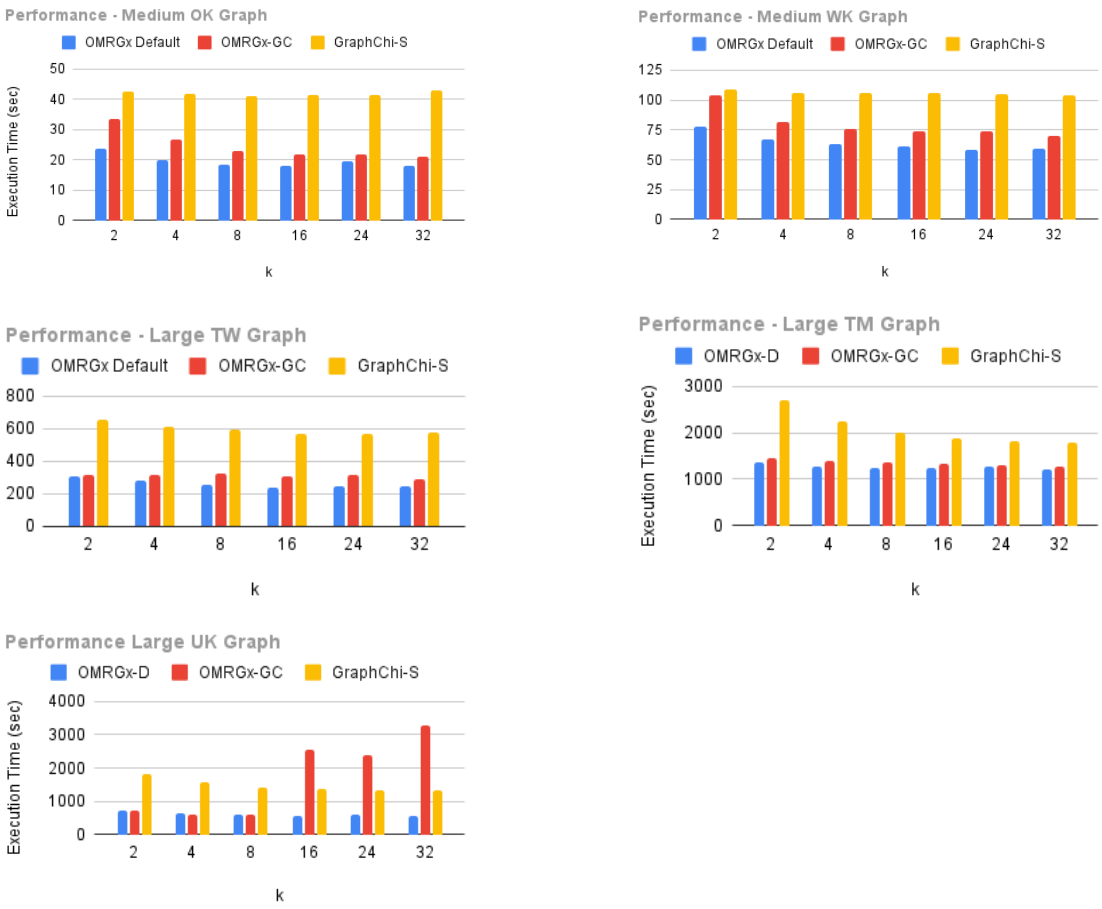


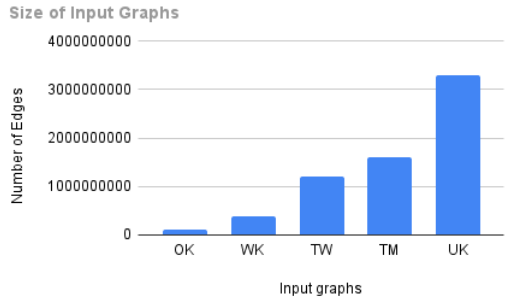
Figure 4.8: Comparison of Execution Times (seconds) for *Partition-based* Graph Processing Algorithms implemented in *OMRG*x vs. their standalone implementations using input graphs of varying sizes and different number of partitions.

Next, we compare the runtime performance for the graph processing frameworks implemented in OMRGx with their standalone counterparts. We ran two iterations of PageRank with all the graph processing algorithms used in our evaluation. OMRGx-D is OMRGx's default processing framework implemented using a *hash* partitioner. For the all the input graphs (except UK), both OMRGx-D and OMRGx-GC perform far *better* than well-tuned and hand-optimized GraphChi-S. This is due to the highly optimized *reduce* phase in OMRGx which maximizes sequential disk accesses when processing input from disk. With

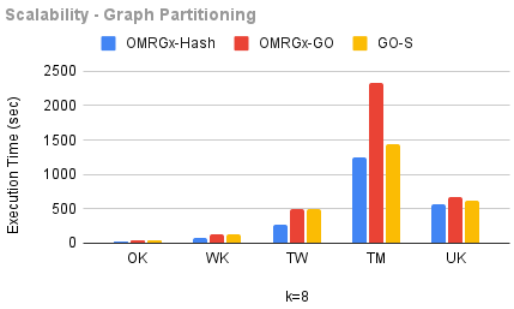
the largest input graph UK, our OMRGx-GC implementation runs slower as the runtime switches to using *disk* with the number of partitions increasing beyond 8. As we noticed in Table 4.2, the E/V ratio forUK is high and we store the other information along with the *edges* using the protocol buffers, as seen in Figure 4.8. Furthermore, the implementation of *GraphChi* specific data structures to store the sub-graph in memory and building GraphChi meta data for the sliding shards needs more memory for the large input graph showing the overall slowdown. However, the default processing algorithm OMRGx-D which uses the available memory and does not need additional data structures is able to run the input UK graph entirely in memory without using disk and hence performs far better.

4.8 Scalability

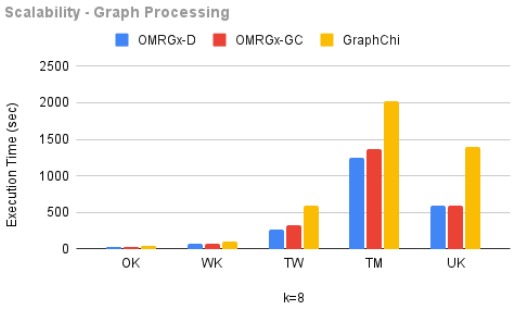
Next, we study the scalability of our graph partitioning and processing algorithms. Figure 4.9a shows the size of the input graphs in terms of number of edges. UK has the highest number of edges. Figure 4.9 shows the scalability data for both graph partitioning and processing algorithms for creating 8 partitions of each input graph. As seen in Figure 4.9b, both our graph partitioning algorithms OMRGx-Hash and OMRGx-GO scale well with the size of the input graph. We see that the execution time for TM graph is more than UK graph when we compare the corresponding number of edges for each graph. This is because the TM graph has highest number of vertices with E/V ratio of 1.6. Since we perform vertex-based processing in memory, the overall execution time for TM graph increases. Nevertheless, both OMRGx-Hash and OMRGx-GO were able to process the entire graph in memory without the need to store the intermediate results on disk.



(a) Size of Input Graphs in terms of number of edges.



(b) Showing Scalability of Graph Partitioning Algorithms.



(c) Showing Scalability of Graph Processing Algorithms.

Figure 4.9: Scalability for Graph Partitioning and Processing Algorithms w.r.t the size of the input graphs for the number of partitions k = 8.

For our graph processing algorithms, we ran two iterations of PageRank implemented on top of all the graph partitioning algorithms used in the evaluation. As shown in Figure 4.9c our OMRGx-D and OMRGx-GC implementations scale well for both medium and large input graphs used in the experiments. For the largest UK graph, both OMRGx-D and OMRGx-GC show a speedup *greater than* 50 when compared with the finely-tuned standalone implementation GraphChi-S. As noted earlier, the execution time for TM graph is more than UK graph due to low E/V ratio which implies more time is spent in performing vertex-based computations in memory i.e., calculating the PageRank for each vertex.

It is important to note that OMRGx-D and OMRGx-GC perform consistently better with the speedups ranging from 2.2× for the medium sized OK graph to 2.4× for the largest UK graph when compared with the standalone implementation GraphChi-S.

4.9 Summary

In this chapter, we presented our extended out-of-core MapReduce system which can perform graph analytics. OMRGx can be easily programmed by using its simple API to express the processing logic while runtime takes care of all the complex details like parallelism, memory management and IO. Through our evaluation we showed that our system can be easily used to 1) partition large graphs and 2) perform the partition-based graph processing of the input graphs, making the user *oblivious* about out-of-core and focus more on the programming logic.

Chapter 5

Related Work

This chapter summarizes various prior research in domains and problems addressed by this thesis. We first summarize the existing solutions for single machine MapReduce systems. Next, we present the related work for out-of-core graph partitioner for single machine. Finally, we summarize the work related to out-of-core graph partitioning and partition-based graph processing using MapReduce.

5.1 Mapreduce on a Single Machine

Various single machine based MapReduce solutions have been developed over the past decade [4, 10, 14, 17, 21, 25] that efficiently utilize the processing capabilities of a single machine (i.e., cores, caches and memory) to perform in-memory data-processing. In other words, none of the systems actively employ disks for processing, and hence they assume that the entire data being processed must fit in the main-memory of a single machine.

Metis [14] is the state of the art single machine map-reduce system that primarily addresses the performance bottlenecks in the data-structure that groups intermediate key-value pairs. It develops different strategies based on workload characteristics like number of keys used by the application and the frequency of repetition of keys. It mainly relies on a combination of hash table and B+ tree data structures where each entry of the hash table is a B+ tree entry. However, for workloads with unexpected key distributions, it falls back to the simplified B+ tree data structure. To further accelerate processing, it uses the Streamflow [18] memory allocator.

Pheonix [17] showed that the MapReduce model could be used on shared-memory machines, with scalability comparable to hand-coded Pthreads solutions. It is optimized for a class of workloads that feature high per task computation and a large, unknown number of keys. Pheonix rebirth [25] optimizes Pheonix for a quad-chip, 32-core, 256-thread UltraSPARC T2+ system with NUMA characteristics. It uses a multi-layered approach that comprises optimizations on the algorithm, implementation, and OS interaction to achieve up to 19× speedup over 256 threads. Pheonix++ [21] is a rewrite of Pheonix to address its various performance issues including uniform intermediate storage, combiner implementation, and poor task overhead amortization. However, it uses expensive copy operation for resizing its hash table and grouping values with the same key across threads.

MATE [10] explores the effectiveness of traditional MapReduce API to produce efficient implementations of a subclass of data-intensive applications. It further extends the API by including support for programmer-managed reduction object, which results in lower memory requirements at runtime; and operates on top of Pheonix. [4] divides a large

MapReduce job into a number of small sub-jobs and iteratively processes one sub-job at a time. It also incorporates several optimizing techniques targeting multicore, including the intermediate data-structure reuse, a NUCA/NUMA-aware scheduler, and pipelining reduce phases with successive map phases.

Hone [12] scales down Hadoop to run on shared-memory machines. It efficiently executes an existing Hadoop jar on a multi-core shared memory machine, enabling existing Hadoop algorithms to run on most suitable runtime environment on datasets of varying sizes. Google's MR4C [8] enables running native code within Hadoop [1] and HDFS [19] to seamlessly scale over distributed setting. Hadoop also supports single machine based execution for debugging and processing smaller datasets; as discussed in Section 2.5, it often requires main-memory to perform its sorting and shuffling phase which limits the amount of data that can be processed in a single machine setting.

MapReduce implementations have also been widely explored beyond single machine based processing, including in-memory execution on supercomputers using MPI [7] and GPU based processing [9, 20]. Finally, [13, 45] demonstrate scalable single machine out-of-core graph processing solutions comparable to distributed in-memory processing like [22–24], which are based on a different execution and programming model compared to the generalized MapReduce model. Recent works like [15] also argue about single-machine designs due to their promising performance.

5.2 Graph Partitioning on a Single Machine

It is important to note that there is no related paper on out-of-core graph partitioning. While there is a single machine multilevel graph partitioner like Mt-Metis [36], however, it cannot generate partitions when presented with large graphs. Therefore, end-to-end graph partitioning using the sophisticated partitioners is still an open problem. Existing out-of-core graph processing systems employ light-weight partitioning strategies like cyclic hashing, chunking, and random partitioning based upon vertex ids. Similarly, LUMOS [44] uses an equally simple vertex-degree based partitioning strategy that has synergy with its intra- and inter-partition dependence-aware value propagation strategy. LUMOS and other systems cannot use sophisticated partitioners like Metis [32], or its adaptations [36, 37], because Metis would require much more memory than the size of the graph and thus could not run on a machine where the graph does not fit in memory. Our work makes it possible for out-of-core systems to use a sophisticated general partitioning algorithm as is embodied in GO that minimizes inter-partition edges and balances partition sizes.

5.3 Out-of-Core Graph Processing

With the popularity of single machine Graph Analytics, many graph processing frameworks [13, 43, 44, 47, 51, 53–59] have been developed that allow disk based processing of large graphs to scale beyond the available main memory. To enable out-of-core processing of large graphs, these systems first divide the graph into partitions that reside on disk, and then process these partitions one-by-one by streaming through them, i.e., by sequentially loading them in memory and immediately processing them. The preprocessing of the partitions is

dependent on the graph processing framework. For example, GraphChi [13] allows the processing of large graphs by creating the shards in specific format. Gridgraph [47], on the other hand, requires the input graph to be preprocessed into a binary format. Nonetheless, there is no out-of-core system which can provide application programmers with the flexibility to use a graph partitioner of their choice paired up with the graph processing framework.

Chapter 6

Conclusions and Future Work

6.1 Contributions

This dissertation's contributions address the memory problems faced by single machine systems when processing large datasets. The contributions are divided into three parts: 1) an efficient approach to perform out-of-core MapReduce by maximizing *sequential disk access* when handling large datasets; 2) a *single-level* graph partitioner which can efficiently partition large graphs in a *memory constrained* manner; and 3) an extension of a single machine MapReduce system to perform single machine graph analytics, including both graph partitioning and graph processing.

6.1.1 OMR: Out-of-core MapReduce for Large Datasets

We presented a single machine out-of-core MapReduce system to process datasets that are larger than main memory using a memory constrained processing model. OMR actively minimizes disk I/O operations by enabling On-the-fly aggregation of the intermediate

values and sequential block disk accesses via ordered batches. Therefore, OMR guarantees linear scaling with growing data sizes. Moreover, OMR optimizes data management via fixed sized key-value pairs by eliminating the need to maintain indexing information on disk, which further reduces random disk accesses. Finally, to maintain efficient in-memory execution, OMR employs lockless processing that eliminates thread synchronization within map and reduce phases.

6.1.2 GO: Out-of-core Graph Partitioner for Large Graphs

We showed the efficacy of using a single-level graph partitioner versus multilevel graph partitioning strategy employed by the popular Mt-Metis system. GO is a single-level out-of-core graph partitioner that can function within the memory constraints imposed by the machine and successfully partition graphs that far exceed the size of a graph that can be held in memory. GO performs just two passes over the entire input graph, the *partition creation pass* that creates *balanced* partitions and the *partition refinement pass* that reduces *edgecuts*. Both passes are designed to function in a memory constrained manner.

6.1.3 OMRGx: MapReduce for Graph Partitioning and Processing

Finally we presented OMRGx, a MapReduce system that can partition graphs and perform partition-based processing of large input graphs. OMRGx has been built by extending the API of our OMR system which supports out-of-core MapReduce applications. We showed how a single machine MapReduce system can overcome the challenges in large scale processing of graphs by simulating the BSP model and avoiding the communication costs that are incurred in a distributed environment. We leveraged the parallelism, effi-

cient memory management and optimized IO provided by our single machine out-of-core MapReduce system making it a good choice to be used as a graph partitioning and processing system. We demonstrated the efficacy of our OMRGx system by programming different graph partitioning and processing algorithms.

6.2 Future Work

This thesis demonstrated the effectiveness of leveraging the runtime provided by well-optimized out-of-core single machine systems to program graph processing frameworks and algorithms with ease. *OMRG*x can also be used as a tool to perform analytics for social media networks by programming different algorithms to analyze datasets for retrieving useful information. Our *OMR* system demonstrated the applicability of MapReduce in use cases requiring simple aggregations (like counting, joining etc). However, the extended API in *OMRG*x system can be used to program applications for social media datasets like analyzing the age of your followers on instagram or finding people with the same interest etc to yield useful results.

Application in Diverse Domains We would like to explore the use of *OMRG*x system by programming applications in various domains. *OMRG*x can be used to program both compute and data intensive tasks to achieve parallelism by leveraging its runtime while making users *oblivious* about out-of-core. Specifically, domains like bioinformatics and computational genomics involve algorithms that process large connected structures similar to graphs. Therefore, such domains are good candidates to explore first. Future work can explore the applicability and customization of our proposed solutions to specific domains.

Single-level Graph Partitioner Our single-level out-of-core graph partitioner *GO* demonstrated its effectiveness in partitioning large graphs on a *Single-PC* while maintaining balance and low edgecuts. The algorithms works intuitively by creating the balanced partitions at the first place and then refining the partitions to minimize edgecuts. Since the datasets continue to grow larger, this approach can be used as the first step to partition large datasets in general for the data or compute intensive tasks thereby, minimizing the communication costs between the nodes while efficiently performing out-of-core.

Extending to Other Platforms With the tremendous growth of datasets in the past few years, we believe that this work can be extended to a platform where a mobile device serves as the frontend and a server in the cloud serves as the backend. Thus, the power of large scale analytics can be put in hands of users who can carry out analytics tasks at anytime and from anywhere.

Bibliography

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. Puma: Purdue mapreduce benchmarks suite. 2012.

[3] Protocol Buffers. Google's data interchange format, 2011.

[4] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 523–534, 2010.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, pages 10–10, 2004.

[6] L. Fang, K. Nguyen, G. Xu, B. Demsky and S. Lu. Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs In *SOSP '15*, pages 394-409, 2015.

[7] Tao Gao, Yanfei Guo, Boyu Zhang, Pietro Cicotti, Yutong Lu, Pavan Balaji, and Michela Taufer. Mimir: Memory-efficient and scalable mapreduce for large supercomputing systems. In *IEEE Intl. Parallel and Distributed Processing Symposium*, pages 1098–1108, 2017.

[8] Google. MR4C. `https://github.com/google/mr4c`.

[9] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A mapreduce framework on graphics processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[10] Wei Jiang, Vignesh T Ravi, and Gagan Agrawal. A map-reduce system with an alternate api for multi-core environments. In *International Conference on Cluster, Cloud and Grid Computing*, pages 84–93, 2010.

[11] Sai Charan Koduru, Rajiv Gupta, and Iulian Neamtiu. Size oblivious programming with InfiniMem. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 3–19, 2016.

[12] K Ashwin Kumar, Jonathan Gluck, Amol Deshpande, and Jimmy Lin. Hone: Scaling down hadoop on shared-memory systems. *Proceedings of the VLDB Endowment*, 6(12):1354–1357, 2013.

[13] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *USENIX OSDI*, pages 31–46, 2012.

[14] Yandong Mao, Robert Morris, and M Frans Kaashoek. Optimizing mapreduce for multicore architectures. In *Computer Science and Artificial Intelligence Laboratory, MIT Technical Reports*, 2010.

[15] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at what COST? *15th Workshop on Hot Topics in Operating Systems* (HotOS), 2015.

[16] G.A. Miller, E.B. Newman, and E.A. Friedman. Length-frequency statistics for written english. *Information and Control*, 1(4), 1958.

[17] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *IEEE International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[18] Scott Schneider, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *International Symposium on Memory Management*, pages 84–94. ACM, 2006.

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2010.

[20] Jeff A. Stuart and John D. Owens. Multi-gpu mapreduce on gpu clusters. In *IEEE International Parallel & Distributed Processing Symposium*, pages 1068–1079, 2011.

[21] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *International Workshop on MapReduce and Its Applications*, pages 9–16, 2011.

[22] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 237-251, 2017.

[23] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223-236, 2017.

[24] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms using a Relaxed Consistency based DSM. In *International Conference on Object Oriented Programming Systems, Languages and Applications* (OOPSLA), pages 861-878, 2014.

[25] Richard M Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *IEEE International Symposium on Workload Characterization*, pages 198–207, 2009.

[26] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *USENIX HotCloud*, 10(10-10):95, 2010.

[27] T. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, pages 153-159, 1992.

[28] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing* (TOPC), 5(3), 13, 2019.

[29] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *IJPP*, (16):498-513, 1987.

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

[31] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *ACM/IEEE conference on Supercomputing*, 1995.

[32] G. Karypis, and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. In *J. Parallel Distributed Comp.*, 48(1):96-129, 1998.

[33] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *ACM/IEEE Conference on Supercomputing*, 1996.

[34] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49: 291?307, 1970.

[35] J. Leskovec. "Stanford large network dataset collection," *http://snap.stanford.edu/data/index.html*, 2011.

[36] D. LaSalle, and G. Karypis. Multithreaded Graph Partitioning. In *IEEE International Parallel and Distributed Processing Symposium*, 2013.

[37] D. LaSalle, Md M. A. Patwary, N. Satish, N. Sundaram, G. Karypis and P. Dubey. Improving Graph Partitioning for Modern Graphs and Architectures. In *IA3 Workshop*, 2015.

[38] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *Proceedings of the VLDB Endowment 5*, 8 (2012), 716-727.

[39] G. Malewicz, M.H. Austern, A.J.C Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Intl Conf. on Management of Data*, pages 135-146, 2010.

[40] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *ACM SOSP*, pages 456-471, 2013.

[41] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proc. International Conference and Exhibition on High-Performance Computing and Networking*, pages 493-498, 1996.

[42] P. Sanders and C. Schulz. Distributed evolutionary graph partitioning. *CoRR*, vol. abs/1110.0477, 2011.

[43] J. Shun and G. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, pages 135-146, 2013.

[44] K. Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *USENIX Annual Technical Conference*, pages 429-442, 2019.

[45] K. Vora, G. Xu, and R. Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *USENIX Annual Technical Conference*, pages 507–522, 2016.

[46] C. Walshaw and M. Cross. Parallel Optimization Algorithms for Multilevel Mesh Partitioning. In *Parallel Comp.*, 26(12):1635-60, 2000.

[47] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX Annual Technical Conference*, pages 375–386, 2015.

[48] G Kaur, K Vora, S.C. Koduru and R Gupta. OMR: Out-of-Core MapReduce for Large Datasets. Proceedings of the *ACM SIGPLAN International Symposium on Memory Management*, pages 71-83, 2018.

[49] L. G. Valiant. A Bridging Model for Parallel Computation. In *Communications of the ACM*, 33(8):103111, 1990.

[50] G Kaur and R Gupta. GO: Out-of-Core Partitioning of Large Irregular Graphs. In *15th IEEE International Conference on Networking, Architecture and Storage*, 2021.

[51] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *USENIX SOSP*, pages 472488, 2013.

[52] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX OSDI*, pages 301316, 2016.

[53] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li and H. Yang. NXgraph: An efficient graph processing system on a single machine. In *IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 409-420, 2016.

[54] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen. Wonderland: A Novel Abstraction-Based OutOf-Core Graph Processing System. In *ASPLOS*, pages 608621. ACM, 2018.

[55] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In *KDD*, pages 7785, 2013.

[56] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *USENIX SOSP*, pages 410424, 2015.

[57] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O In *USENIX ATC*, pages 125137, 2017

[58] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai. Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication. In *USENIX ATC*, pages 195207, 2017.

[59] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *EuroSys*, pages 527543. ACM 2017.

[60] http://konect.cc/networks/