OMR: Out-of-Core MapReduce for Large Data Sets

Gurneet Kaur
University of California Riverside
Riverside, CA, USA
gkaur007@cs.ucr.edu

Sai Charan Koduru
University of California Riverside
Riverside, CA, USA
scharan@cs.ucr.edu

Keval Vora
Simon Fraser University
British Columbia, Canada
keval@cs.sfu.ca

Rajiv Gupta
University of California Riverside
Riverside, CA, USA
gupta@cs.ucr.edu

Abstract
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. Our experience shows that the state-of-the-art single-machine in-memory MapReduce system Metis frequently experiences out-of-memory crashes. 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, the single-machine setup of the Hadoop system performs much slower when it is presented with the datasets which are larger than the main memory. Moreover, such frameworks also require tuning a lot of parameters which puts an added burden on the programmer. In this paper we present OMR, an Out-of-core MapReduce system that not only successfully handles datasets that are far larger than the size of main memory, it also 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 OMR’s linear scalability and empirically demonstrate it by processing datasets that are up to 5x 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 in contrast to Metis, OMR avoids out-of-memory crashes for large datasets as well as delivers higher performance when datasets are small enough to fit in main memory.

CCS Concepts • Information systems → MapReduce-based systems; • Computing methodologies → Parallel computing methodologies; Parallel programming languages;

Keywords Out-of-Core, MapReduce, Single Machine, Fixed/Variable Sized Records, Lockless Memory Constrained Processing, Parallel Programming, Data Processing

ACM Reference Format:

1 Introduction
The prevalence of large datasets has led to development of various efficient big data processing tools like Spark [27], MapReduce [5], PowerGraph [7], 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, 26] 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISMM’18, June 18, 2018, Philadelphia, PA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5801-9/18/06... $15.00
https://doi.org/10.1145/3210563.3210568
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, 22] 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 InfiniMen [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.

We present 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 get maximized whenever disk operations are required. 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 consistent 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 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, Figure 3, and Figure 4 show that OMR efficiently processes datasets that are up to 5x larger than main memory, proving the linear scalability of OMR as analyzed in Section 4. In contrast, Metis fails to process large data sets. Furthermore, results in Table 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. While after tuning many parameters available on Hadoop, Hadoop is able to process large data sets, OMR outperforms Hadoop by 1.5-41.2x for data sets ranging from 8GB to 80GB in size.

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.

Algorithm 1 Word count example in MapReduce

1: function map(line)  6: function reduce(<key,[value]>)
2: for word \( \in \) line do  7: frequency ← sum([value])
3: emit(<word, 1>)  8: output(key, frequency)
4: end for        9: end function
5: end function

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.

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.

### Table 1. Performance of Metis [14] on various MapReduce algorithms – ✓ represents out-of-memory crashes.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Key Size</th>
<th>Value Size</th>
<th>Dataset</th>
<th>Metis</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount</td>
<td>1-32 bytes</td>
<td>8 bytes</td>
<td>8GB</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16GB</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32GB</td>
<td>✗</td>
</tr>
<tr>
<td>InvertedIndex</td>
<td>1-32 bytes</td>
<td>32 bytes</td>
<td>8GB</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16GB</td>
<td>✗</td>
</tr>
<tr>
<td>SequenceCount</td>
<td>7-132 bytes</td>
<td>4 bytes</td>
<td>8GB</td>
<td>✗</td>
</tr>
<tr>
<td>RankedInvertedIndex</td>
<td>5-98 bytes</td>
<td>130 bytes</td>
<td>8GB</td>
<td>✗</td>
</tr>
<tr>
<td>MovieRatings</td>
<td>8 bytes</td>
<td>4 bytes</td>
<td>8GB</td>
<td>✗</td>
</tr>
<tr>
<td>DegreeCount</td>
<td>8 bytes</td>
<td>4 bytes</td>
<td>8GB</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16GB</td>
<td>✗</td>
</tr>
<tr>
<td>AdjacencyList</td>
<td>8 bytes</td>
<td>Unbounded</td>
<td>8GB</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16GB</td>
<td>✗</td>
</tr>
<tr>
<td>SelfJoin</td>
<td>1-32 bytes</td>
<td>Unbounded</td>
<td>8GB</td>
<td>✗</td>
</tr>
</tbody>
</table>

Another option available is to make use of a single machine MapReduce framework [4, 10, 14, 17, 21, 26] 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 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, 22] 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 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
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.

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.

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.

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 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 1b. During map phase, each thread owns a row of in-memory buffers and the <key, 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 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.

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
**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

**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
buffer from which they are serialized. Since records within the batch are contiguous, as shown in Figure 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 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.

### 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 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 deserialized (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 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).

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 (as shown in Section 3.3).

### 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, 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 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.
Table 2. Size types for keys and values of various MapReduce algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Key/Size</th>
<th>Value/Size</th>
<th>Combine/Size</th>
<th>Fixed Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MovieRatings</td>
<td>Integer/Fixed</td>
<td>Integer/Fixed</td>
<td>Integer/Fixed</td>
<td>Yes</td>
</tr>
<tr>
<td>HistogramMovies</td>
<td>Integer/Fixed</td>
<td>Integer/Fixed</td>
<td>Integer/Fixed</td>
<td></td>
</tr>
<tr>
<td>DegreeCount</td>
<td>Integer/Fixed</td>
<td>Integer Pair/Fixed</td>
<td>Integer Pair/Fixed</td>
<td></td>
</tr>
<tr>
<td>WordCount</td>
<td>String/Variable</td>
<td>Integer/Fixed</td>
<td>Integer/Fixed</td>
<td></td>
</tr>
<tr>
<td>InvertedIndex</td>
<td>String/Variable</td>
<td>Bit Vector/Fixed</td>
<td>Bit Vector/Fixed</td>
<td></td>
</tr>
<tr>
<td>SequenceCount</td>
<td>String/Variable</td>
<td>Integer/Fixed</td>
<td>Integer/Fixed</td>
<td></td>
</tr>
<tr>
<td>RankedInvertedIndex</td>
<td>String/Variable</td>
<td>Bit Vector/Fixed</td>
<td>Bit Vector/Fixed</td>
<td></td>
</tr>
<tr>
<td>Grep</td>
<td>Integer/Fixed</td>
<td>String/Variable</td>
<td>String Vector/Variable</td>
<td>No</td>
</tr>
<tr>
<td>SelfJoin</td>
<td>String/Variable</td>
<td>String/Variable</td>
<td>String Vector/Variable</td>
<td></td>
</tr>
<tr>
<td>AdjacencyList</td>
<td>Integer/Fixed</td>
<td>Pair of Integers/Variable</td>
<td>Vector of Pair of Integers/Variable</td>
<td></td>
</tr>
</tbody>
</table>

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 \(<key, \text{value}>\) pairs generated by \(\text{map}()\) 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 \(<key, \text{value}>\) pairs. With the total number of \(<key, \text{value}>\) pairs generated by \(\text{map}()\) 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 \(<key, \text{value}>\) 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:

\[
C_B = 2 \cdot \frac{n/p}{B} + \frac{n/p}{b_m} + \frac{n/p}{b_r} \quad (1)
\]

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 \(<key, [\text{value}]>\) pairs, and \(p, b_m\) and \(b_r\) are similarly defined over \(<key, [\text{value}]>\) 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. The size of index information maintained by InfiniMem for a \(<key, [\text{value}]>\) 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. 1. Hence, the lower bound on total I/O cost \(C^L_B\) can be computed as:

\[
C^L_B = 2 \cdot \frac{n/p}{B} + 2 \cdot \frac{n/p}{B_i} + 2 \cdot \left( \frac{n/p}{b_m} + \frac{n/p}{b_r} \right) \quad (2)
\]

It is important to note that \(B_i \geq B\) which makes \(C^L_B\) 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. While the I/O costs for disk seeks and index block transfers remain same as in Eq. 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 \(<key, \text{value}>\) 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:

$$C_B^U = \frac{n \cdot S_p - \left( \frac{n}{p} b_m \right) \cdot (p - 1) \cdot S_k}{B \cdot S_p} + \frac{2 \cdot n/p}{B_i} + 2 \cdot \frac{n/p + n/p}{b_m}$$

Using Eq. 2 and Eq. 3, we bound the total I/O cost $C_B$ as:

$$C_B^L \leq C_B \leq C_B^U$$

making $C_B$ linear in $n/B$ (from $C_B^U$ in Eq. 3).

**C) Non-Sequential Disk Seeks:** Finally, Eq. 1, Eq. 2 and Eq. 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.

### 5 Evaluation

#### 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 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.

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.

#### 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 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 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 each benchmark executed, it showed a slowdown between 8.7-21.8x 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 15336MB 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.inputfileinputformat.split.minsize (0-256MB), mapreduce.reduce.shuffle.parallelcopies (5-10) to bring down the execution times. However, even after changing these parameters and gaining better performance, as shown in Table 4, OMR greatly outperforms Hadoop across all the benchmarks.

In Table 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×

Table 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.

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Benchmarks</th>
<th>WC</th>
<th>II</th>
<th>SC</th>
<th>RII</th>
<th>MR</th>
<th>DC</th>
<th>AL</th>
<th>SJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 GB</td>
<td>Hadoop (sec)</td>
<td>2876</td>
<td>2006</td>
<td>2455</td>
<td>1638</td>
<td>5474</td>
<td>3815</td>
<td>6436</td>
<td>2175</td>
</tr>
<tr>
<td></td>
<td>OMR (sec)</td>
<td>71</td>
<td>78</td>
<td>807</td>
<td>718</td>
<td>693</td>
<td>1010</td>
<td>1553</td>
<td>1326</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>40.6×</td>
<td>25.8×</td>
<td>3.04×</td>
<td>2.28×</td>
<td>7.90×</td>
<td>3.78×</td>
<td>4.14×</td>
<td>1.64×</td>
</tr>
<tr>
<td>16 GB</td>
<td>Hadoop (sec)</td>
<td>5844</td>
<td>3778</td>
<td>4907</td>
<td>3168</td>
<td>9346</td>
<td>7481</td>
<td>12932</td>
<td>4566</td>
</tr>
<tr>
<td></td>
<td>OMR (sec)</td>
<td>142</td>
<td>379</td>
<td>1550</td>
<td>1195</td>
<td>1356</td>
<td>1992</td>
<td>2521</td>
<td>2652</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>41.3×</td>
<td>9.97×</td>
<td>3.17×</td>
<td>2.65×</td>
<td>6.90×</td>
<td>3.76×</td>
<td>5.13×</td>
<td>1.72×</td>
</tr>
<tr>
<td>32 GB</td>
<td>Hadoop (sec)</td>
<td>10953</td>
<td>10540</td>
<td>9162</td>
<td>5435</td>
<td>19104</td>
<td>12861</td>
<td>24288</td>
<td>8326</td>
</tr>
<tr>
<td></td>
<td>OMR (sec)</td>
<td>616</td>
<td>728</td>
<td>2848</td>
<td>1977</td>
<td>2642</td>
<td>4035</td>
<td>4773</td>
<td>5293</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>17.8×</td>
<td>14.5×</td>
<td>3.22×</td>
<td>2.75×</td>
<td>7.23×</td>
<td>3.19×</td>
<td>5.09×</td>
<td>1.57×</td>
</tr>
<tr>
<td>48 GB</td>
<td>Hadoop (sec)</td>
<td>20733</td>
<td>20790</td>
<td>15344</td>
<td>7779</td>
<td>28527</td>
<td>20557</td>
<td>37261</td>
<td>12254</td>
</tr>
<tr>
<td></td>
<td>OMR (sec)</td>
<td>1052</td>
<td>939</td>
<td>4522</td>
<td>2792</td>
<td>3854</td>
<td>6002</td>
<td>7044</td>
<td>8050</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>19.7×</td>
<td>22.1×</td>
<td>3.39×</td>
<td>2.79×</td>
<td>7.40×</td>
<td>3.43×</td>
<td>5.29×</td>
<td>1.52×</td>
</tr>
<tr>
<td>64 GB</td>
<td>Hadoop (sec)</td>
<td>24344</td>
<td>30833</td>
<td>19809</td>
<td>12937</td>
<td>38518</td>
<td>30864</td>
<td>52262</td>
<td>16989</td>
</tr>
<tr>
<td></td>
<td>OMR (sec)</td>
<td>1169</td>
<td>1198</td>
<td>5304</td>
<td>2905</td>
<td>5074</td>
<td>8206</td>
<td>9032</td>
<td>10930</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>20.9×</td>
<td>25.4×</td>
<td>3.74×</td>
<td>4.45×</td>
<td>7.59×</td>
<td>3.76×</td>
<td>5.79×</td>
<td>1.55×</td>
</tr>
<tr>
<td>80 GB</td>
<td>Hadoop (sec)</td>
<td>33293</td>
<td>43821</td>
<td>27152</td>
<td>14330</td>
<td>47755</td>
<td>40008</td>
<td>63763</td>
<td>22745</td>
</tr>
<tr>
<td></td>
<td>OMR (sec)</td>
<td>1519</td>
<td>1459</td>
<td>6699</td>
<td>3287</td>
<td>6105</td>
<td>10201</td>
<td>11087</td>
<td>12893</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>22×</td>
<td>30×</td>
<td>4.05×</td>
<td>4.4×</td>
<td>7.82×</td>
<td>3.92×</td>
<td>5.75×</td>
<td>1.76×</td>
</tr>
</tbody>
</table>

Figure 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.
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).

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 3.

We also compare the raw file sizes for intermediate data generated by map phase in Figure 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 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 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 process significantly increases the overall batch writing time for InfiniMem compared to that for OMR-FX (see Figure 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 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
Table 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. \(X\) \([k\ GB]\) 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 \(k\ GB\).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Version</th>
<th>WC</th>
<th>II</th>
<th>SC</th>
<th>RII</th>
<th>DC</th>
<th>MR</th>
<th>AL</th>
<th>SJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1GB</td>
<td>OMR</td>
<td>8.51</td>
<td>8.60</td>
<td>16.17</td>
<td>35.07</td>
<td>118.62</td>
<td>45.51</td>
<td>27.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Metis</td>
<td>20.74</td>
<td>22.26</td>
<td>33.66</td>
<td>21.40</td>
<td>24.13</td>
<td>46.86</td>
<td>28.45</td>
<td>241</td>
</tr>
<tr>
<td></td>
<td>Hadoop</td>
<td>275</td>
<td>173</td>
<td>230</td>
<td>95</td>
<td>508</td>
<td>670</td>
<td>666</td>
<td>511</td>
</tr>
<tr>
<td>2GB</td>
<td>OMR</td>
<td>15.88</td>
<td>17.79</td>
<td>36.54</td>
<td>68.71</td>
<td>236.09</td>
<td>97.19</td>
<td>242.37</td>
<td>50.01</td>
</tr>
<tr>
<td></td>
<td>Metis</td>
<td>44.88</td>
<td>48.50</td>
<td>(X) [1.7GB]</td>
<td>45.32</td>
<td>13.88</td>
<td>(X) [1.9GB]</td>
<td>61.87</td>
<td>5.01</td>
</tr>
<tr>
<td></td>
<td>Hadoop</td>
<td>511</td>
<td>292</td>
<td>436</td>
<td>188</td>
<td>883</td>
<td>1366</td>
<td>1684</td>
<td>511</td>
</tr>
<tr>
<td>4GB</td>
<td>OMR</td>
<td>30.40</td>
<td>35.11</td>
<td>78.05</td>
<td>133.25</td>
<td>469.00</td>
<td>204.40</td>
<td>487.17</td>
<td>467.52</td>
</tr>
<tr>
<td></td>
<td>Metis</td>
<td>95.00</td>
<td>98.73</td>
<td>(X) [1.7GB]</td>
<td>94.36</td>
<td>3.8GB</td>
<td>(X) [1.9GB]</td>
<td>1780</td>
<td>3406</td>
</tr>
<tr>
<td></td>
<td>Hadoop</td>
<td>1081</td>
<td>629</td>
<td>928</td>
<td>334</td>
<td>2923</td>
<td>3406</td>
<td>1036</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6. Varying record size.

the possibility of aggregating subsequent \(<\text{key}, \text{value}>\) pairs. Hence, the increase in number of combinations is slower for AL and SJ compared to other benchmarks. Impact of Record Sizes: Figure 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 \(<\text{key}, \text{value}>\) pairs 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 6.

Comparing the execution time with OMR-VR in Figure 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 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 \(<\text{key}, \text{value}>\) pair which increases its execution 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 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.
6 Related Work

Various single machine based MapReduce solutions have been developed over the past decade [4, 10, 14, 17, 21, 26] 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 [26] 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 19x 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. [12] scales down Hadoop to run on shared-memory machines. It efficiently executes an existing Hadoop jar on a multicore 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 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 [6] and GPU based processing [9, 20]. Finally, [13, 22] demonstrate scalable single machine out-of-core graph processing solutions comparable to distributed in-memory processing like [23–25], 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.

7 Conclusion

We presented OMR, a single machine out-of-core MapReduce system to process datasets that are larger than main memory. OMR actively minimizes disk I/O operations and guarantees linear scaling with growing data sizes. In our experiments with eight MapReduce algorithms, we theoretically proved OMR’s linear scalability and demonstrated it by processing datasets that are up to 5x larger than main memory as shown in Section 5. Our experiments 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 the Metis system.

Acknowledgments

This work is partially supported by National Science Foundation grant CCF-1524852 to University of California Riverside.

References