ZIPTHRU: A SOFTWARE ARCHITECTURE THAT EXPLOITS ZIPFIAN SKEW IN DATASETS FOR ACCELERATING BIG DATA ANALYSIS

by

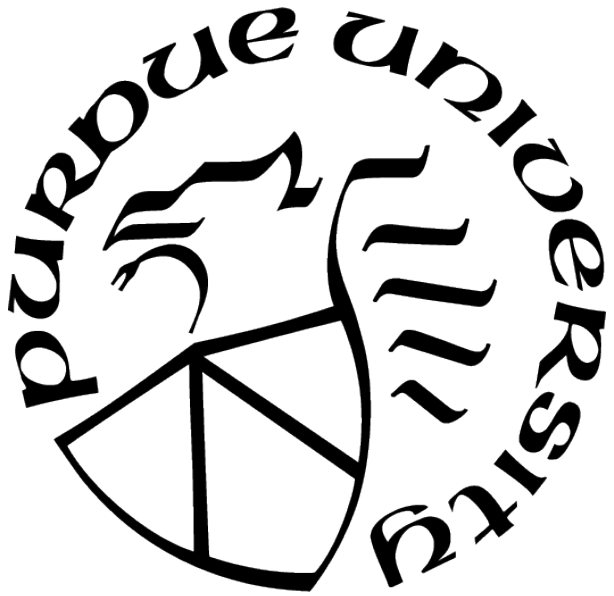
Ejebagom John Ojogbo

A dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



Electrical and Computer Engineering

West Lafayette, Indiana

December 2020

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. T.N. Vijaykumar, Co-chair

School of Electrical and Computer Engineering

Dr. Mithuna S. Thottethodi, Co-chair

School of Electrical and Computer Engineering

Dr. Milind Kulkarni

School of Electrical and Computer Engineering

Dr. Samuel P. Midkiff

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

To my parents, friends, and advisors. I could not have done any of this without your support.

ACKNOWLEDGMENTS

I would like to acknowledge all those that helped and supported me throughout the research process, especially my advisors Profs T.N. Vijaykumar and Mithuna Thottethodi, and the rest of my committee. Thank you all for your help, advice, and guidance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In the past decade, Big Data analysis has become a central part of many industries including entertainment, social networking, and online commerce. MapReduce, pioneered by Google, is a popular programming model for Big Data analysis, famous for its easy programmability due to automatic data partitioning, fault tolerance, and high performance. Majority of MapReduce workloads are summarizations, where the final output is a per-key "reduced" version of the input, highlighting a shared property of each key in the input dataset.

While MapReduce was originally proposed for massive data analyses on networked clusters, the model is also applicable to datasets small enough to be analyzed on a single server. In this single-server context the intermediate tuple state generated by mappers is saved to memory, and only after all Map tasks have finished are reducers allowed to process it. This Map-then-Reduce sequential mode of execution leads to distant reuse of the intermediate state, resulting in poor locality for memory accesses. In addition the size of the intermediate state is often too large to fit in the on-chip caches, leading to numerous cache misses as the state grows during execution, further degrading performance. It is well known, however, that many large datasets used in these workloads possess a Zipfian/Power Law skew, where a minority of keys (e.g., 10%) appear in a majority of tuples/records (e.g., 70%).

I propose ZipThru, a novel MapReduce software architecture that exploits this skew to keep the tuples for the popular keys on-chip, processing them on the fly and thus improving reuse of their intermediate state and curtailing off-chip misses. ZipThru achieves this using four key mechanisms: 1) Concurrent execution of both Map and Reduce phases; 2) Holding only the small, reduced state of the minority of popular keys on-chip during execution; 3) Using a lookup table built from pre-processing a subset of the input to distinguish between popular and unpopular keys; and 4) Load balancing the concurrently executing Map and Reduce phases to efficiently share on-chip resources.

Evaluations using Phoenix, a shared-memory MapReduce implementation, on 16- and 32-core servers reveal that ZipThru incurs 72% fewer cache misses on average over tradi-

tional MapReduce while achieving average speedups of 2.75x and 1.73x on both machines respectively.

1. INTRODUCTION

Big Data analysis has become a cornerstone sector in computing in the past decade. As the number of devices and services on the Internet has grown, the amount of data generated by consumers using these devices and services has increased tremendously. This data provides much needed information, context, and value for products in many fields, from entertainment to online commerce, social networking, and more, and its unabated growth has pushed demand for quicker and more efficient means of analyzing, organizing, and extracting useful information from the data.

In response the computing industry has come up with a variety of tools to tackle Big Data analysis. Because of the vast amounts of data and its disparate sources most solutions have been built around distributed computing on networked clusters, usually following the MapReduce paradigm[1] and other related variants (e.g., Dryad[2]).

The MapReduce programming model, pioneered by Google and inspired by the Map and Reduce functions used in functional programming, is popular mainly due to its high programmability. This programmability is facilitated by the framework's built-in support for task management, data management, and fault tolerance, features that are indispensable for distributed analyses on networked machines. The most widespread MapReduce framework is Apache Hadoop[3].

In MapReduce execution is split into two phases: a Map phase and a Reduce phase. The Map phase produces *key-value* tuples from the input data (e.g., <word, 1> where the word is the key and the value is 1). Each tuple is then sent to a specific reducer based on its key, which performs a user-defined Reduce operation on all the tuples for a given key (e.g., a simple summation to determine each word's count).

While the MapReduce model supports any program whose work can be split into Map and Reduce phases, majority of the workloads performed with the paradigm are summarization workloads where the final output is a per-key "reduced" version of the input data, like the word count example above. The goal in such workloads is to extract some information on each key in dataset, producing a smaller, compressed version of the input data that highlights the target property.

Other distributed computing frameworks/models used for Big Data analysis include the Bulk-Sychronous Parallel Model[4], which focuses on streamlining communication and synchronization between nodes in a distributed network; and Apache Spark[5], built for in-memory computing and geared mainly towards both iterative and query-based programming models for use in neural nets and databases.

Despite these frameworks' focus on analyzing huge amounts of data across distributed clusters, many real-world analyses are performed on (relatively) smaller datasets that can be processed efficiently on a single machine. These single-server workloads still benefit from the features of the MapReduce model, however; in fact the computations performed by individual nodes in distributed workloads use MapReduce to process their local data before sending their output on the network for global reduction. Apache Spark, which supports many different execution models, specifically targets such use.

While there has been a plethora of recent work in accelerating the local execution on single servers and individual nodes in distributed clusters, most of this work[6][7] has focused on integrating application-specific, data-agnostic FPGA accelerators and GPUs within the compute process. Because many of these frameworks are written in high-level managed languages and run on general purpose CPUs, researchers have focused on speeding up local execution by allowing the easy integration of accelerators into compute[8], in some cases allowing the user to customize the architectures to their specific computing needs [9] .

However, it is well known that many datasets possess a skewed key distribution, one that is either Zipfian in nature or adheres to some other Power Law formula. In these datasets a minority of keys (e.g., 10%) are present in a majority of tuples (e.g, 70%), leading to a majority of accesses to the tuples with these keys. A prime example of a Zipfian dataset that has uses in many Big Data contexts is words in the English language. Just the top word alone (the) accounts for over 7% of all word occurrences in regular English use [10]. This example illustrates the power of this distribution, as word identification and analysis is used in variety of Big Data contexts such as keyword search, sort, and tagging. Social networks provide another example, where a small minority of more popular users usually possess a disproportionate number of links/edges compared to the majority of users. Algorithms

analyzing graphs in Big Data are ubiquitous and would benefit from any architecture that optimizes for such skew.

In traditional MapReduce programs no special attention is paid to the organization of the data or its inherent properties. Thus during the Map phase the intermediate tuple state for all keys is saved/sent to the reducers and only after all Map tasks have concluded are reducers allowed to proceed. This Map-then-Reduce *sequential* mode of execution means that the tuples are reduced far in the future after being produced by Map, resulting in poor locality due to distant reuse. In addition the intermediate state produced by mappers usually cannot fit in the on-chip caches due to its size. As a result mappers and reducers suffer numerous cache misses and memory accesses as execution progresses, which in turn degrades performance.

To address these issues I propose ZipThru, a novel MapReduce software architecture that exploits the inherent skew in big datasets and processes the state for popular keys on the fly, thus improving reuse, preserving locality, and boosting performance. ZipThru achieves this via four key mechanisms:

- First, instead of the sequential Map-then-Reduce mode of execution which induces distant reuse, ZipThru employs *concurrent* Map-and-Reduce to produce and consume the popular keys' tuples on-chip immediately, facilitating near reuse.

- Second, ZipThru takes advantage of the fact that in summarization workloads the reduced state of the popular keys is a small fraction of the total reduced state of the full dataset. As a result ZipThru can hold this smaller state on-chip as it reduces the tuples for the popular keys, greatly improving locality and curtailing off-chip misses. The tuples/intermediate state for the unpopular keys follow the default MapReduce flow and are reduced later in a separate, final Reduce phase.

- Third, ZipThru employs a lookup table to identify the tuples with popular keys. This lookup table is populated based on an offline, pre-processing step on a subset of the input data, the cost of which is amortized over numerous MapReductions due to the naturally slow-changing nature of popularity.

- And fourth, in contrast with sequential MapReduce which uses all system resources for the Map and then Reduce phases and as a result is naturally load balanced, ZipThru's concurrency introduces the need to efficiently balance system resources between the simultaneously executing Map and Reduce tasks. Thus ZipThru offers two load-balancing schemes: a static scheme that fixes the number of threads that execute each task at the start of execution, and a dynamic scheme that uses all system threads for both tasks, switching modes between Map and Reduce depending on the workload in order to achieve load balance.

ZipThru is completely transparent to existing MapReduce programs and can run them with little modification. In addition, because of the broad applicability of the MapReduce model, ZipThru's optimizations are also applicable to cluster-scale workloads as well as the single-server context that is the focus of this work.

Evaluations using Phoenix[11], a shared-memory MapReduce implementation, on two 16- and 32-core servers reveal that ZipThru incurs an average of 72% fewer cache misses over traditional MapReduce programs, as well as average speedups of 2.75x and 1.73x on both servers respectively.

The rest of this work is structured as follows: First I provide more background on the traditional MapReduce paradigm and the limitations and drawbacks that exist in the current mode of execution. Next I describe ZipThru and the key mechanisms it employs to take advantage of the skew in large datasets in order to improve performance. Then I cover experimental methodology and show comparisons between ZipThru and traditional MapReduce for a number of datasets with varying degrees of inherent skew. Lastly I discuss related works in the field and highlight ZipThru's novelty before concluding.

2. BACKGROUND

2.1 MapReduce

As mentioned in Chapter 1 traditional MapReduce programs have their execution defined in two sequential, non-overlapping phases: a Map phase and a Reduce phase, illustrated in Figure 2.1.

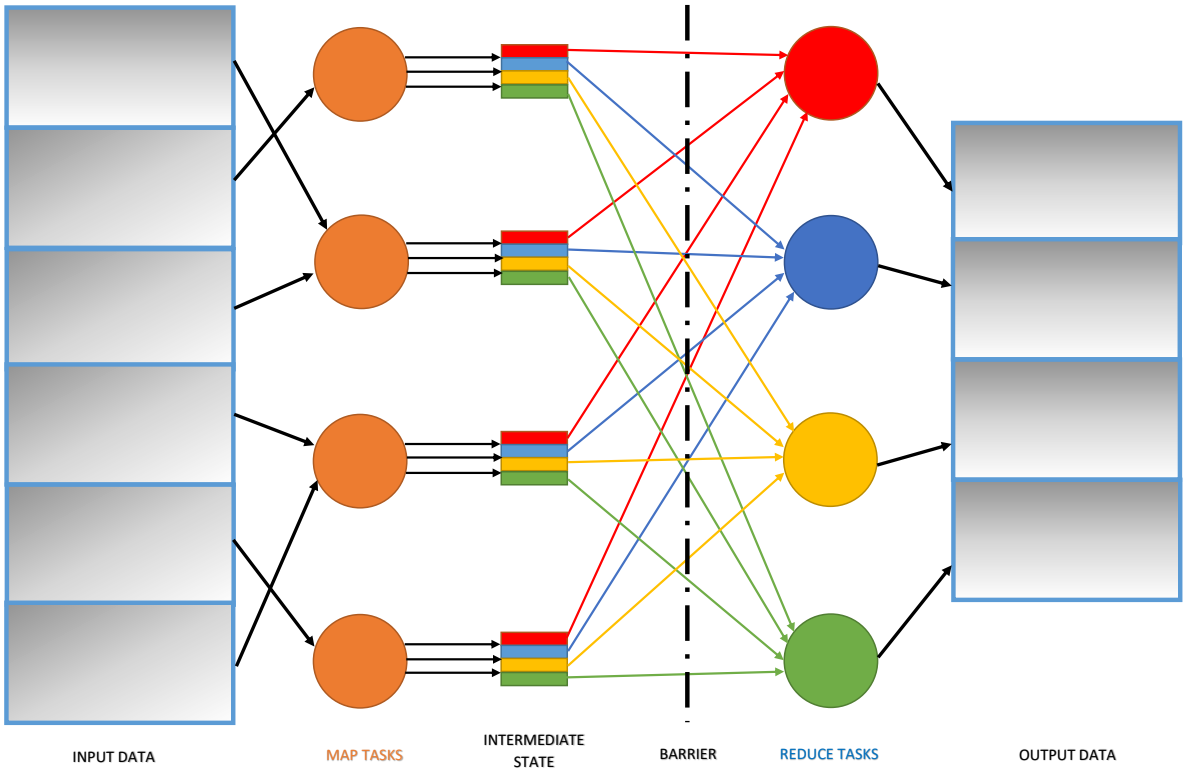


Figure 2.1. Traditional MapReduce workflow

During the Map phase mappers process separate chunks of input data and emit key-value tuples, placing these tuples into individual buckets based on their keys (the intermediate state in Figure 2.1) for future Reduce. Then after the barrier i.e., once the Map tasks have finished processing their inputs, the intermediate state is communicated/shuffled from mappers to reducers and reducers perform a user-defined operation on the tuples of each of their assigned keys, generating their output. A classic example of a MapReduce workload is word count,

where mappers will emit tuples <word, 1> for each word they encounter in the input data, and reducers sum all the tuples for a given word to determine the word's count in the dataset.

Because of the size of the intermediate state generated by mappers while processing the input, summarization MapReduce programs often include a combiner function, which takes data emitted by local Map tasks and "pre-reduces" or combines it in order to pare down its footprint before placing the new data in reducer buckets. This combining reduces pressure on memory in the single-server context and saves network bandwidth in multi-cluster workloads.

Despite the MapReduce model's applicability to both single and multi-cluster execution the challenges faced by programs executing in the two contexts are quite different. In multi-cluster workloads special attention is paid to fault tolerance and task management across multiple machines, along with network optimizations to make communicating tuples between mappers and reducers more efficient.

Single-server workloads do not have to deal with these issues. The built-in multi-threading supported by the local machine's operating system, along with the shared-memory infrastructure used by mappers and reducers for saving/sending tuples, provides automatic support for communication. In addition, the fact that all execution takes place on a single machine obviates the need for fault tolerance.

As a result the key challenges faced by programs in the single-server context mainly deal with efficient use of local CPU resources, particularly the on-chip caches shared by mappers and reducers. This work is focused on boosting performance by optimizing memory access for MapReduce programs operating in this context.

2.2 Memory Access Patterns in Single-Server MapReduce

When running as a multi-threaded program there are three main sources of memory accesses in MapReduce: 1) The input data, which is partitioned and fed into the Map tasks; 2) The intermediate state, which is produced by the Map tasks and read by reducers to generate the final output; and 3) The final reduced state of the data, which reducers produce as they process the intermediate state for each of their assigned keys.

2.2.1 Input Data

In the shared-memory context input data can be represented as a contiguous memory-mapped file. Map tasks read this data sequentially and in general will process each block only once. Even for Map tasks that don't access their input blocks once, temporal locality will still be fairly high during execution. While access patterns in input processing are highly dependent on both the dataset and the specific function the Map tasks are executing, this sequential-access/high-locality assumption is a fundamentally emergent property of the MapReduce paradigm. Programs that don't benefit from implicit partitioning of input data and allow independent functioning of Map tasks already make a poor fit for MapReduce and will be unable to take advantage of its benefits. As a result it is reasonable to expect very simple, high locality memory accesses from Map tasks processing independent chunks of input data.

The upshot of this is that input processing in MapReduce needs very little by way of memory optimizations; sequential, high-locality memory access patterns are already well served by hardware and software prefetching [12].

2.2.2 Intermediate State

Memory accesses to the intermediate state in traditional MapReduce suffer from a major deficiency that hurts performance: poor locality from distant reuse due to the sequential nature of the traditional Map-then-Reduce paradigm. This distant reuse leads to numerous off-chip misses as a result of the sheer size of the intermediate state generated by mappers.

As mappers process their inputs and emit tuples for each key they encounter, they must insert these tuples into reducer-specific queues/buckets for later reduction. Because each mapper operates on independent chunks of input data each mapper maintains private queues for every future reducer, allowing them to insert tuples into these queues without worrying about synchronization and interference from other concurrently executing Map tasks.

For each tuple <K1, V1> encountered for a given key K1, a mapper must insert it into the appropriate queue for K1's reducer. Regardless of the data structure used to hold this queue all mappers will have to contend with the compute cost and memory accesses that come with

handling constantly growing intermediate state. For a sufficiently large and unsorted input each mapper can be expected to see a tuple for almost every key in the dataset, leading to a huge amount of intermediate state stored in each mapper's private queues.

Unlike input processing, where mappers can be expected to make simple high-locality accesses to each input block, the access patterns for the intermediate state are unpredictable and as a result offer poor locality. A mapper accessing the state for key K1 assigned to reducer R1 in no way guarantees that the next tuple it encounters will belong to either K1 or R1. Thus as intermediate state grows during execution, and as mappers encounter tuples belonging to different keys, mappers will make multiple disparate memory accesses to various portions of their queues. If the state cannot be held in the on-chip caches - a certainty when dealing with large datasets - then each mapper will suffer cache misses as various portions of their queues are brought in and out of the cache.

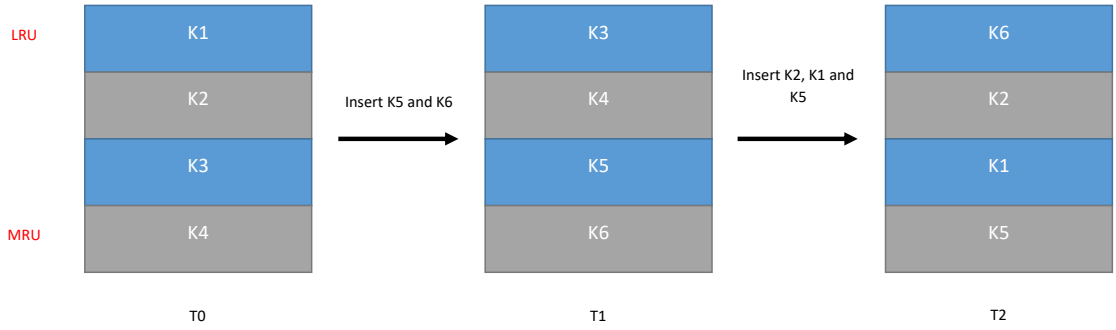


Figure 2.2. Accesses to mapper's intermediate state

Figure 2.2 offers an illustration of how these access patterns would hurt locality. For simplicity it displays the cache state of a single mapper and assumes that the per-key intermediate state doesn't grow. The replacement policy of the cache is LRU.

At T0, the mapper is holding intermediate state for 4 different keys in its cache, K1 through K4. The cache is full. Then at T1 the mapper encounters tuples for keys K5 and K6 and attempts to insert them, leading to evictions of the states of keys K1 and K2, the least recently used blocks in the cache. Because the states for K5 and K6 are off-chip the mapper also incurs slowdowns due to cache misses as it fetches their states from memory in order to update them.

Once again at time T2 the mapper encounters new tuples, this time belonging to K2, K1, and K5. However because K1 and K2 were evicted at T2 due to the cache being at capacity, their accesses now miss in the cache and once again slow down execution.

As mentioned in the previous section, MapReduce programs often include a combiner function used by the framework to pare down intermediate state as it is being created in order to control its size. As illustrated in Figure 2.2 however, even if the per-key state is kept fixed due to aggressive combining, for sufficiently large datasets the full intermediate state would still be too large to fit on-chip and as a result would still suffer slow memory accesses due to off-chip misses.

Once mappers have finished and reducers start executing, each reducer must access the intermediate state held in *all* mappers' private queues in order to collect all the tuples for their assigned keys. Once again, depending on the access patterns in the previous phase and the size of the intermediate state, many of the tuples read by reducers for processing will not be in the on-chip caches. This poor locality due to distant reuse of the tuple state means that reducers will be fetching their state from memory in order to process it, resulting in much slower execution as they wait for their cache misses to be resolved.

These access patterns are illustrated in Figure 2.3. Once again for simplicity the figure only includes the state for a single reducer and assumes its cache is holding state generated by the mapper in Figure 2.2.

At time T3 the reducer attempts to read and reduce the states for keys K3 and K4. Even though those keys' states were in the cache at T1 in Figure 2.2, they have since been evicted to make room for subsequent updates to other keys' intermediate state. The reducer's reuse of their state, being too far in the future to take advantage of their prior presence in the cache, incurs a cache miss and leads to an off-chip memory access.

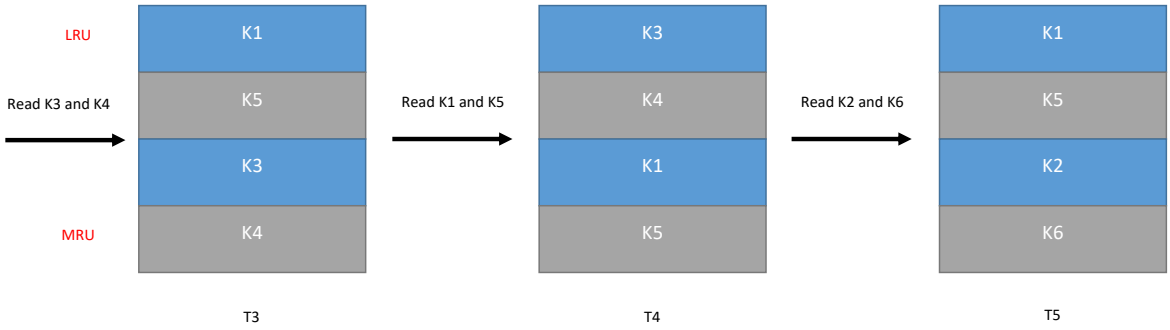


Figure 2.3. Accesses to reducer's intermediate state

At time T4 the reducer attempts to read and reduce the states for K1 and K5. Because these are still on-chip, however, the accesses hit in the cache and the reducer can quickly read and process those keys before moving on.

Finally at time T5 the reducer reads keys K2 and K6 and encounters a similar situation to the one at T3. Both states suffer from distant reuse and as a result are no longer in the cache, leading to cache misses and much slower memory accesses.

2.2.3 Final Reduced State

Much like access patterns for input data, memory accesses to the final reduced state in MapReduce programs can often be expected to be high-locality and as a result fast.

During the Reduce phase, each reducer will read the state for all tuples for given key, process and reduce them into their final state, and then move on to the next assigned key. While the process of reading the intermediate state for each key may generate multiple accesses both on- and off-chip (as illustrated in Figure 2.3), the actual per-key final reduced state will be accessed very frequently as it is computed and updated, keeping it on-chip.

As a result memory accesses to the final reduced state of the data also need little to no optimization. The primary cause of slowdowns rests mainly in both the Map and Reduce phases' interactions with the data's intermediate state.

2.3 Zipfian/Power Law Skew in Big Datasets

While the classic MapReduce paradigm specifies execution phases and has underlying frameworks designed to optimize parallelism and communication, it is largely data-agnostic. Thus little work has been done to optimize performance for datasets that possess certain features and naturally generate certain kinds of memory accesses.

It is well known, however, that many datasets used in Big Data analysis possess a Zipfian/Power Law distribution and would benefit from frameworks specifically designed to take advantage of it. Already mentioned in Chapter 1 is word distribution in the English language, a canonical example of a Zipfian distribution.

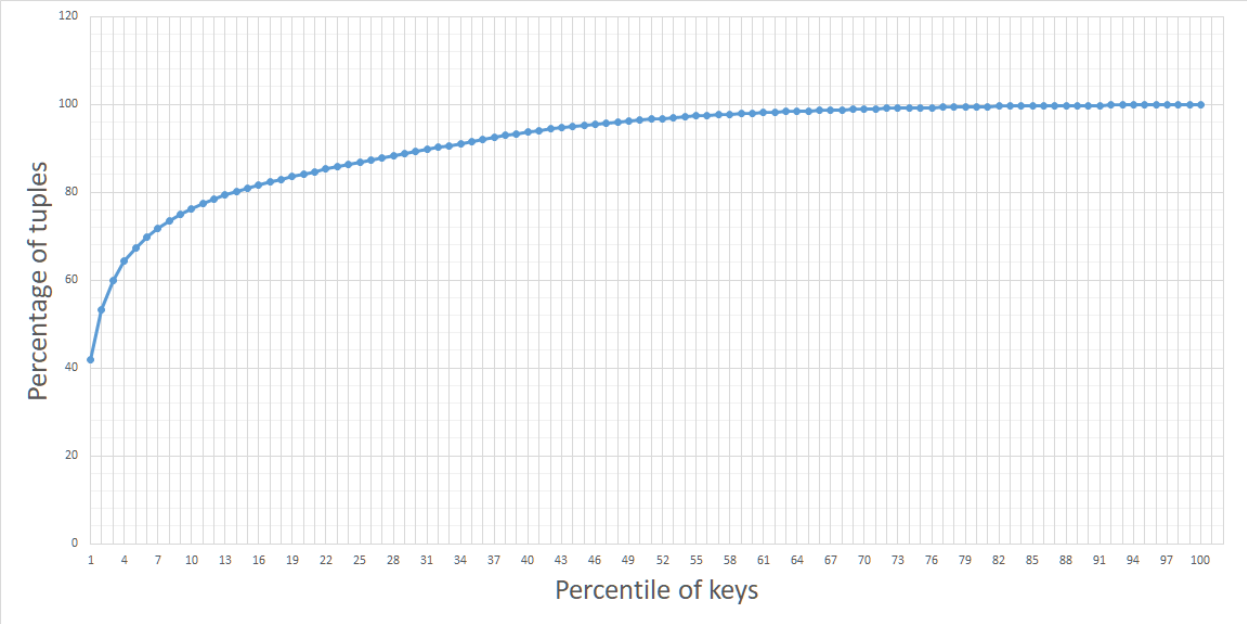


Figure 2.4. Twitter's data distribution

To further this point Figures 2.4 and 2.5 show the data distributions for two social networking datasets, Twitter[13][14] and Friendster[15]. The X-axes in both figures show the percentile of keys in the dataset, while the Y-axes show the percentage of tuples each

percentile is responsible for. While the intensity of the skew differs between the social networks, they both show that a small minority of keys is responsible for a large majority of tuples/connections. For Twitter we can see that the top 10% of users account for about 76% of all connections in the graph, while Friendster has a more modest 10%-59% skew.

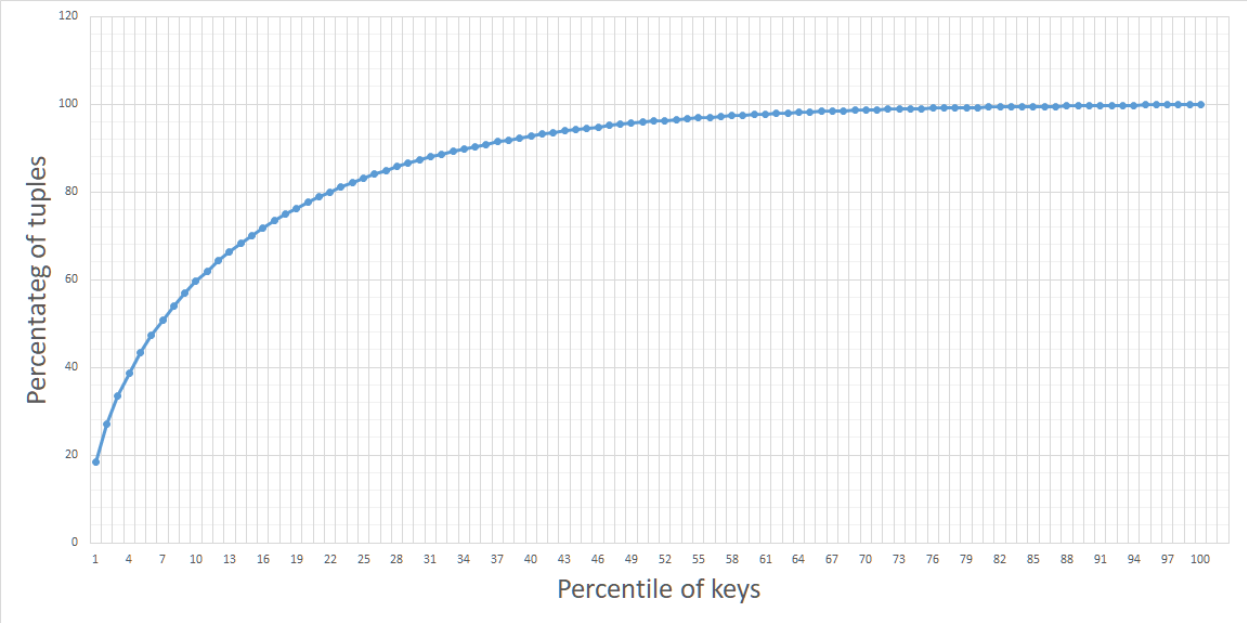


Figure 2.5. Friendster's data distribution

This skew offers huge opportunity for improving the performance of Big Data analysis programs. A small minority of keys showing up in a large majority of tuples/records means that if these tuples can be reduced on the fly the distant reuse suffered by reducers would be eliminated, as the immediate processing would consume the tuples as they are emitted by the mappers. This near reuse of the intermediate state would in turn lead to a reduction in the memory accesses/off-chip misses associated with handling and updating said state, thus boosting performance.

ZipThru exploits this inherent skew to facilitate near reuse and reduce the memory footprint of MapReduce programs analyzing large datasets. I discuss ZipThru and the key mechanisms it uses to achieve this in the next chapter.

3. ZipThru

In order to reduce memory footprint and facilitate near reuse of intermediate state ZipThru extends the traditional MapReduce framework to provide support for immediate processing of popular keys. It does this in a completely transparent fashion, allowing programs designed for the traditional MapReduce paradigm to run as is, with very little modifications. ZipThru achieves it's goals via four key mechanisms:

- Concurrent execution of the Map and Reduce Phases to enable immediate consumption of intermediate state;
- Caching only the small, final reduced state of the popular keys on-chip in order to boost locality and curtail off-chip misses;
- Using a lookup table built from pre-processing the input to identify tuples belonging to popular keys; and
- Load balancing Map and Reduce tasks to make efficient use of on-chip resources during concurrent execution.

I go into further detail about each of these features below.

3.1 Concurrent Execution of Map and Reduce Tasks

In order to ensure that constantly growing intermediate state is kept on-chip it needs to be processed/reduced immediately. To do this ZipThru allows both Map and Reduce tasks to run at the same time. This concurrent execution is in stark contrast with the traditional sequential MapReduce model where, as mentioned in Chapter 2.1, mappers first process the input data and save the intermediate state into private queues and then reducers read these queues and process them once mappers are finished.

ZipThru bypasses this intermediate step by scheduling both Map and Reduce tasks concurrently and by having the mappers send their tuples directly to the reducers for immediate analysis and reduction, illustrated in Figure 3.1.

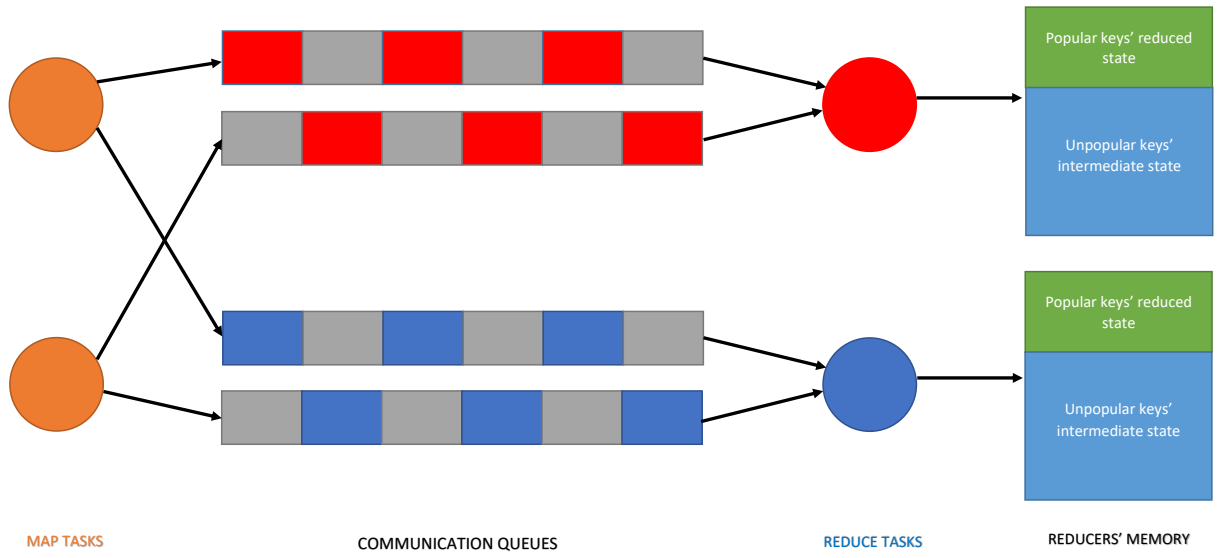


Figure 3.1. Concurrent execution in ZipThru

Figure 3.1 shows a high-level overview of how two mappers and two reducers would operate as they run concurrently. Both types of tasks communicate using shared-memory queues. Much like sequential MapReduce each mapper maintains a private queue for each reducer and each reducer has to read tuples from all its assigned queues. The mapper inserts tuples at one end of the queue and the reducer drains tuples from the other end.

When a reducer reads a tuple from the queue, it checks to see whether or not this tuple belongs to a popular key using a lookup table. I go into more detail on the mechanics of this table lookup later in this chapter (3.3). Tuples belonging to popular keys are reduced immediately and their final state is saved. Tuples belonging to unpopular keys are treated much like the classic MapReduce and are saved into a secondary intermediate state for later reduction.

Because the new communication queues will now be shared by concurrently executing tasks, an emergent challenge is ensuring correctness by avoiding races between the mappers inserting tuples into the queues and the reducers draining tuples from the queues.

A naive way to address this challenge would be the use of synchronization primitives shared by the different threads. Synchronization, however, is bound to slow down execution

considerably, since a mapper would have to acquire and release a lock to insert a tuple and a reducer would have to acquire and release the same lock to drain the tuple.

Such a naive implementation ignores the unique producer-consumer relationship that exists between mappers and reducers. Map tasks only insert and never drain and Reduce tasks only drain and never insert, and so they need only touch completely separate parts of the queue during execution, obviating the need for any kind of coarse-grained synchronization between the threads.

ZipThru takes advantage of this producer-consumer relationship between mappers and reducers to implement lock-free queues for communication between the two task types. In ZipThru, mappers maintain control of pointers to the tail of the queue, where they insert new tuples as they process them. Reducers maintain control of the head of the queue, where they read and remove new tuples for reduction. To limit memory use the queue is circular, and insertion and deletion proceed in a round-robin fashion.

Because of the circular nature of the queue both tasks have to compute the number of elements currently present before modifying, thus avoiding inserts to a full queue and deletes from an empty one. In other words, mappers have to make sure that their tail pointer does not overtake the head and reducers have to make sure that their head pointer does not overtake the tail.

In a normal linear array this can be done by simply subtracting the head from the tail. A simple subtraction, however, would make it impossible to determine the difference between a full queue and an empty one, since both states would produce a value of 0. To distinguish between states ZipThru maintains a one slot gap in the queue. The head and the tail meet only when the queue is empty; if the queue is full the tail remains one slot behind the head.

The various states of the head and tail pointers and what they mean for the length of the queue are shown in Figure 3.2. The last two examples show the difference between a full queue and an empty one. In the full queue the tail is one slot away from the head; as a result a mapper that encounters a queue in this state would be unable to insert a new tuple and move the tail forward. In the empty the queue the tail and the head are equal and a reducer that encounters a queue in this state would be unable to drain any tuples and update the head. All other combinations of the head and tail values allow mappers and

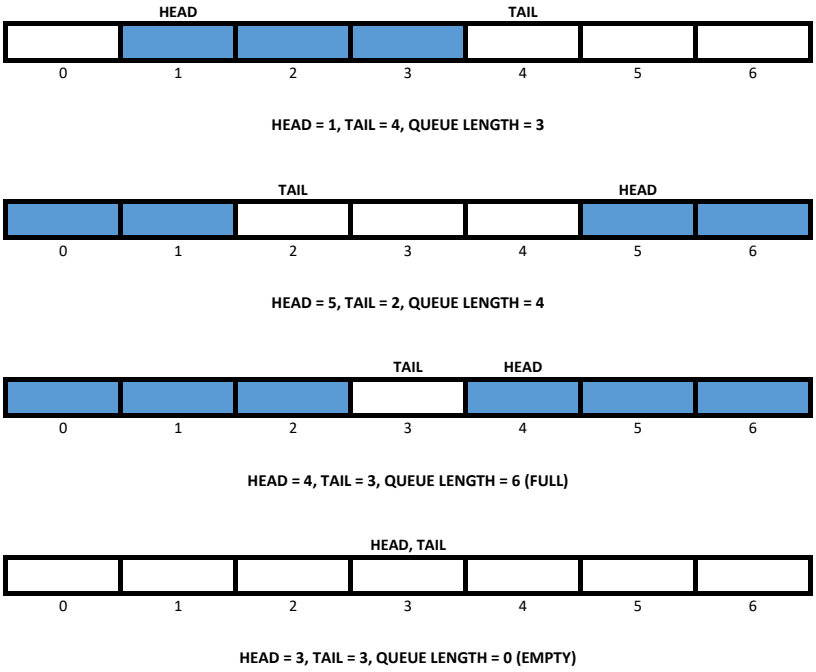


Figure 3.2. Various states of ZipThru's lock-free queues

reducers to insert and drain from the queues at the same without the need for locks and synchronization.

Thus by using lock-free shared-memory queues ZipThru provides mappers and reducers with a fast and accurate communication channel to facilitate concurrent execution.

Once the tuples have arrived at the reducers, reducers have to decide whether to reduce them in-place due to the popularity of their keys or to save their state for later. I discuss the rationale behind this decision in the next section

3.2 Caching Final Reduced State for Popular Keys

In summarization MapReductions the final reduced state of the data is much smaller than the original input data, since the workloads generally aim to compute some aggregate property/properties of each key in the dataset. ZipThru exploits this fact, along with the

fact that the popular keys are a minority of the total key space, to make very efficient use of on-chip cache capacity during concurrent execution. These two features provide ZipThru with two distinct benefits.

First, the frequent accesses to the popular keys' reduced state as reducers read and process their tuples leads to high temporal locality for this state in the caches. This locality ensures that the state is very likely to remain on-chip at the expense of other blocks of memory, guaranteeing fast access as popular keys are encountered and reduced.

Second, because the popular keys constitute a minority of all keys in the dataset, their reduced state will be smaller than the already relatively small final reduced state of the full dataset. This size makes their reduced state more likely to fit in the limited on-chip caches during execution, further reducing the likelihood of the already rare cache misses that would have occurred due to distant reuse of their cache state.

These two advantages make the on the fly reduction of popular keys very fast under ZipThru, boosting performance significantly.

For the other majority of keys with their much fewer tuples ZipThru defaults to the classic MapReduce workflow: tuples belonging to unpopular keys are saved to queues for a separate final reduction. However, because these tuples are a minority of all the tuples in the dataset, this secondary intermediate state is still smaller than the global intermediate state handled by mappers and reducers in the traditional MapReduce, and as a result is less taxing on the cache.

Figure 3.3 illustrates this phenomenon. The figure shows the cache state of a Reduce task reading tuples from a lock-free queue. This task is in charge of keys K1 through K10. Keys K2, K4, and K5 are popular keys and their reduced state is shown in green. The secondary intermediate state for the other unpopular keys is shown in blue.

At time T0 the reducer has K1, K5, K2, and K4 in its cache. The cache is full. Then at T1 it receives tuples for K3, K5, and K4. K5 and K4 are popular keys whose state is already in the cache, and so they are reduced immediately and these accesses are very fast. K3 is unpopular, and so to update its intermediate state the reducer has to go off-chip to fetch it and evict K1 to save it.

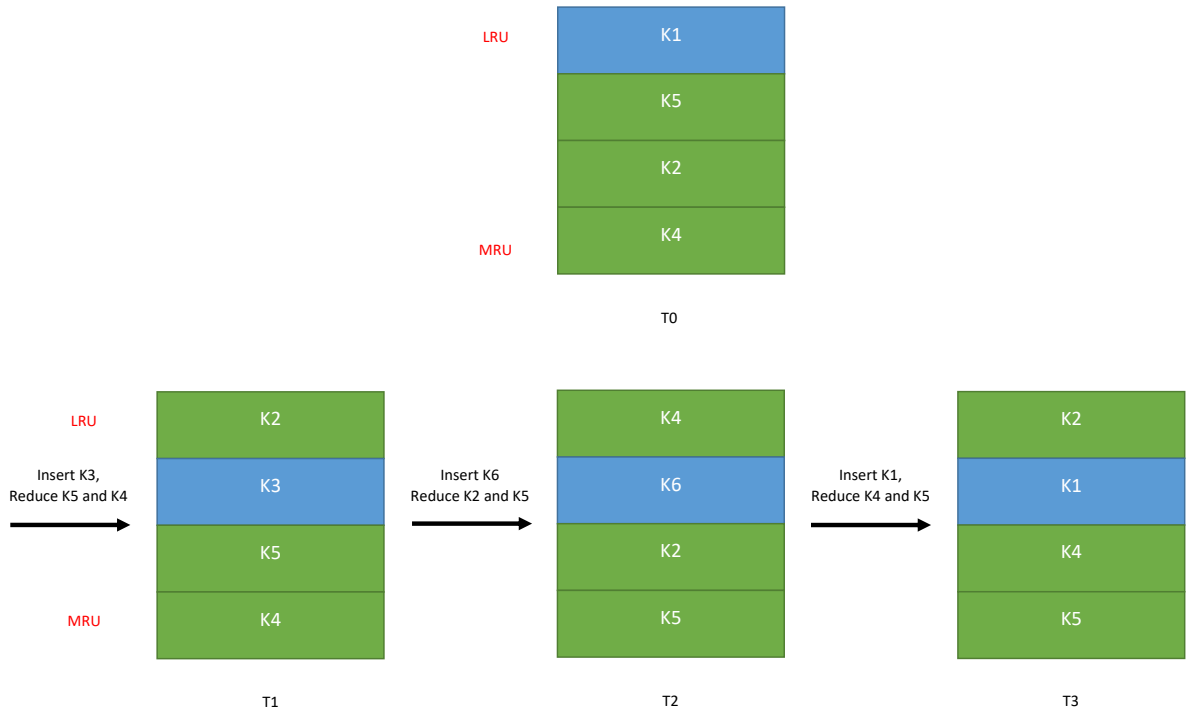


Figure 3.3. Accesses to ZipThru's reduced state during concurrent execution

A similar situation transpires at T2 and T3, where more tuples for the popular keys show up, guaranteeing that those blocks get touched frequently and as a result remain in the cache. Thus even though the reducer suffers the occasional misses for the intermediate state of its unpopular keys, majority of its accesses are cache hits due to the skew that the minority of popular keys enjoys.

In order to take proper advantage of these features, Reduce tasks have to be able to tell popular keys and unpopular keys apart. I discuss how ZipThru makes this distinction in the next section.

3.3 Distinguishing between Popular and Unpopular Keys

To distinguish between popular and unpopular keys, reducers must know which keys are in the popular minority *before* they arrive. This means that some analysis of the dataset must

have been performed prior to the start of execution. ZipThru once again takes advantage of the inherent skew in the dataset to optimize this initial analysis.

Because the popular minority of keys are over-represented in the dataset, a simple sample of a subset of the data is all it takes to reveal the top keys. Prior to execution in ZipThru a traditional MapReduction can be performed on a small sample of the data to determine which keys are over-represented. This pre-processing can even be baked into the process that the creates the dataset for analysis. Collecting all the data into a single unified file for MapReduce will inevitably require database queries to whatever backing store is holding the state of the graph/dataset to be analyzed. By periodically sampling the records as they arrive while building the input a fairly accurate set of the most popular keys can be obtained with little cost/overhead.

Analysis of this tiny sample of the data will take a fraction of the time required for overall execution, and once done will provide a bootstrap list of popular keys for ZipThru to use in its own execution. With the naturally slow changing nature of popularity it is reasonable to assume that across many runs/analyses the top keys in the data will see very little churn. As a result the cost of analyzing this initial sample will be amortized over hundreds of runs before it needs to be performed again.

Once this bootstrap list has been obtained reducers need a way of checking received keys against the keys in the list in order to determine their popularity. ZipThru enables this by way of a lookup table.

3.3.1 Lookup Table for Popular Keys

At the start of execution the user (i.e. the person that wishes to perform analysis on a given dataset) provides the bootstrap list of popular keys from the dataset. ZipThru takes this list and, using the key partitioning algorithm also supplied by the user, determines which of these keys will be sent to which reducers during execution.

Each reducer is then tasked with building its own lookup table of popular keys before it begins draining its queues. Once a tuple has been read from the queues a reducer will hash it to determine its location in the table, and then check if the key stored at that location

corresponds with the key within its received tuple. If the received key is one of the popular keys from the bootstrap list it performs an in-place reduction and saves the updated state.

In using the lookup table ZipThru once again takes advantage of the inherent benefits that come from the skew within the dataset.

First are the advantages of fast lookup conferred by using a fixed-size hash table. Reducers in ZipThru only have to keep track of popular keys and their reduced state. This is in contrast with the continually expanding list of keys and tuples that mappers in traditional MapReduce frameworks must hold on to, the overhead of which will inevitably use more compute resources than a simple hash over a non-expanding list.

Locating the addresses of the final reduced state of the popular keys is also made easier and quicker by the lookup table. Each reducer in ZipThru pre-allocates pointers to the final reduced state for each of their popular keys at the start of execution and includes them in the payload of the lookup table. Thus when a key is looked up not only would a reducer know whether or not it is a popular key, it would also know where to go to fetch its current reduced state and update it.

3.4 Load Balancing Map and Reduce Tasks

Traditional MapReduce frameworks have no need for load balancing of any kind between mappers and reducers because both phases of execution are non-overlapping. Thus at any given point in time all of the machine's resources are fully dedicated to the current phase's needs.

By forcing concurrent execution of Map and Reduce, however, ZipThru must now manage how much CPU time is spent mapping vs reducing as execution progresses. ZipThru achieves this load balancing in two ways: static partitioning and dynamic load balancing.

3.4.1 Static Partitioning of Map and Reduce Tasks

Under static partitioning ZipThru provides the user with the option of specifying how many of the host system's threads are to be used for Map tasks and how many are to be used

for Reduce tasks. As the name implies this partitioning is fixed at the start of execution and will not change throughout the duration of the MapReduction.

Despite its simplicity and low implementation overhead static partitioning has some significant drawbacks. The most obvious is a failure to properly anticipate how many threads are sufficient for each task type. Because the partitioning is fixed at the start of execution it can lead to two different kinds of deficiencies for the two different thread types: mapper blocking and reducer idling.

When mappers wish to send tuples to reducers, they must insert these tuples at the tails of the lock-free queues described in Section 3.1. Because these queues are fixed size however, at some point in execution a mapper is bound to encounter a full queue, and once it does its insert will fail. ZipThru's design is relatively simple on the mapper end; there is no support for saving tuples for full queues so that they can be retried later while the mapper continues processing the input. As a result failure to insert into a queue leads to mapper blocking, where a mapper simply spins in a loop as it attempts to insert until the reducer in charge of draining that queue frees up space and allows the insert to succeed.

If the thread count chosen by the user at the start of execution is too biased in favor of mappers this mapper blocking event will be far more likely and will lead to mappers spending an inordinate amount of execution time just waiting for queues to open up so that they can insert.

Biasing the thread count in favor of the reducers comes with its own issue however. Because the queues used by mappers to send tuples to reducers are private, a reducer will have to drain tuples from as many queues as there are mappers on the system. In order to avoid preference for any mapper reducers in ZipThru drain tuples in a round-robin fashion, constantly cycling through all their queues and checking if there are any tuples in them to reduce. For every queue only one tuple is drained per visit, once again to avoid spending too much time on one mapper at the expense of another.

Too many reducers, while sure to guarantee that instances of mapper blocking are much fewer, will inevitably lead to more situations where a certain reducer has no tuples in its queue for reduction. Again, because the thread assignment is fixed at the start of execution, a reducer that sees that all its queues are empty cannot simply quit and switch to mapping.

It must continue to idle on its empty queues until either a tuple arrives for it to reduce or all mappers signal that they have finished mapping, implying that no new tuples will ever be arriving again.

As a result static partitioning puts pressure on the user to be able to independently gauge how many mappers and reducers to use for a given workload/dataset. For the workloads tested while evaluating ZipThru I discovered that a simple half-way split is usually enough to provide good performance over the baseline. On machines with lower thread counts, or on datasets with a large amount of tuples, a bias in favor of reducers sometimes provided better performance than splitting evenly. Arriving at this point required a lot of experimentation, however, and it's possible that the perfect split for each workload lies in a configuration I did not test.

To avoid making users independently figure out which thread split is optimal for their dataset on their specific hardware ZipThru also includes a dynamic load balancing scheme.

3.4.2 Dynamic Load Balancing of Map and Reduce Tasks

Much like the traditional MapReduce the dynamic scheme in ZipThru allows all threads on a given system to execute both Map and Reduce tasks. The high-level benefits of such a setup are clear: a mapper that finds itself blocking due to a full queue can spend its time reducing/clearing its queues instead, and a reducer that finds all its queues empty can switch back to mapping instead of idly waiting for tuples to arrive for reduction.

The challenge under this new scheme now becomes balancing how much time threads spend in either mode of execution. As was the case with static partitioning, biasing in favor of one vs the other will either lead to higher cases of blocking while mapping or time wasted on queues that aren't full while reducing.

To deal with this ZipThru uses watermarking. During Map threads have a high watermark for how many elements should be present in a queue before they consider switching to Reduce, and during Reduce threads have a low watermark for how many elements should be leftover in their queues before they can safely switch back to Map. The complete decision tree for ZipThru's dynamic load balancing is shown in Figure 3.4

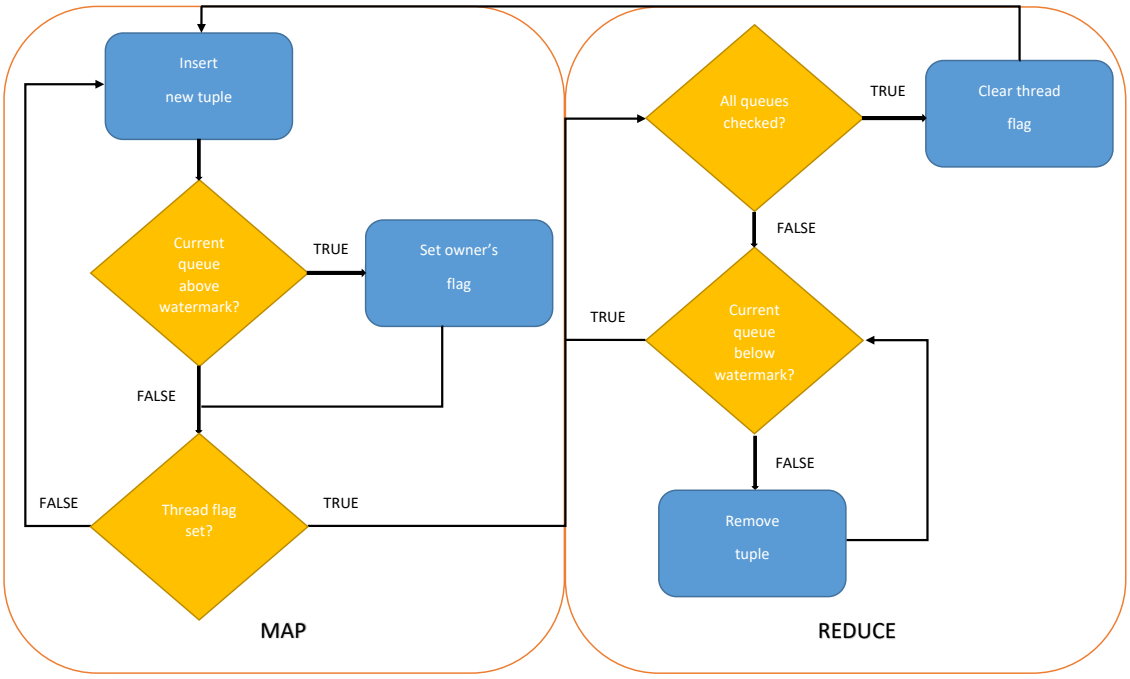


Figure 3.4. Decision tree for threads under ZipThru's dynamic load balancing

To further increase thread efficiency and reduce needless mode-switching ZipThru uses explicit signals between the threads to determine whether or not to switch. Such communication is made necessary by one key observation: the fact that the particular queue that a thread is attempting to insert into is full in no way implies that the inserting thread's reducer queues are full or are in danger of being filled. Switching to Reduce simply because a given thread is blocking on Map, while beneficial for all the other threads that are inserting into its queue, does nothing to alleviate its blocking. In fact it could easily lead to a situation where the inserting thread needlessly switches back and forth instead of simply waiting for the blocked queue to open up.

As a result ZipThru has threads communicate the presence of full queues to each other with a simple flag, also shown in Figure 3.4. At the start of execution ZipThru creates flags to indicate the status of each thread's queues and sets them to false. If while mapping a

thread finds the queue it wants to insert into is above the high watermark it sets the flag for the owner of the queue. Whether or not it succeeds in inserting its tuple it checks the status of its own flag. If its flag has been set it switches to reducing/clearing its queues. If its flag has not been set it simply continues trying to insert into the full queue or, in the event that its insert was successful, moves on to processing more input data.

The introduction of this new avenue of communication once again raises questions about correctness and synchronization. Much like the lock-free queues however ZipThru exploits the relationship between the threads to obviate the need for locks.

First, as mentioned before, every thread gets a single flag. This flag can be set by any other thread, but can only be cleared by the owner. The inevitable races for setting/clearing the flag are thus not a problem from the perspective of correctness. If two threads attempt to set the flag for a third, the end result is still that the flag gets set. If a thread clears its flag after draining all its queues and another thread sets it right after, that is still a valid signal that some queue is blocked or about to be, and so that thread is well within its rights to switch back to Reduce. Lastly, if there is a race between clearing and setting the flag, the thread attempting to set it will still eventually discover that the queue it wants to insert into is above the watermark and so it will try setting the flag again. By clearly demarcating which threads can do what to the flag ZipThru once again allows for fast communication without the overhead of locks or synchronization.

On the Reduce end, a thread stays in Reduce until all of its queues have fallen below the low watermark. Queues that are already below the watermark are ignored, since they pose no threat of being full any time soon, and queues that are above the watermark are drained continuously until they fall.

This method is in direct contrast to the manner of drainage under static partitioning, where each queue only gets one tuple removed at a time. Because switching to Reduce is an indication that at least one thread has set the flag for a given thread's queues, priority should be given to *every* single queue that is above the low watermark. Continually draining offending queues until they fall below the watermark guarantees that a thread will only be forced to switch to Reduce when absolutely necessary and that it would be a while before any thread that was previously blocking on its queues encounters that situation again.

The only tuning required under dynamic load balancing is setting the high and low watermarks. While this could also be exposed to the user it would lead to a similar situation as static partitioning, where users are compelled to experiment with different watermarks to gain optimal performance.

At the moment ZipThru's watermarks are set to 100% and 12.5% for high and low respectively. Setting the high watermark to 100% essentially tells ZipThru to prioritize mapping; threads will only set the flags for other threads if those queues are full. In my testing this was revealed to have the best performance across all surveyed datasets. At a high level this makes sense: Mapping is what produces the data for reduction and provides a bound on how long execution will take. It doesn't matter how many resources are dedicated to Reduce if the input takes too long to process, and forcing threads to switch to Reduce even though they are perfectly capable of continuing to Map will only serve to slow down input processing.

Setting the low watermark to 12.5% (1/8th the size of the queues) also required some experimentation.. With the low watermark the challenge is ensuring the threads don't have to frequently switch back to Reduce because they're not clearing their queues enough each time. In my tests it seems 1/8th the queue length is sufficient to achieve this goal without spending too much time in Reduce at the expense of Map.

To summarize, ZipThru employs four key mechanisms to boost performance while running MapReduce on datasets with Zipfian/Power Law skew:

- It schedules Map and Reduce tasks concurrently, allowing reducers to consume/reuse the intermediate state generated by mappers as it is created;
- It caches only the small reduced state of the minority of popular keys on-chip, boosting locality due to frequent accesses and reducing cache pressure due to its relatively small size;
- It employs a lookup table built from offline pre-processing of a subset/sample of the input data to distinguish between popular and unpopular keys; and

- It offers various mechanisms to load balance concurrently executing Map and Reduce tasks, allowing them to make efficient use of CPU and on-chip resources.

In the next chapter I cover the experimental methodology I use to test the effects ZipThru's contributions had on performance, and then I move on to the results of my tests.

4. EXPERIMENTAL METHODOLOGY

To test ZipThru I use Phoenix[11], a shared-memory MapReduce implementation. I implement ZipThru on top of existing Phoenix code and compare runtimes for the same workload/dataset running on both baseline Phoenix and ZipThru.

The default Phoenix implementation includes an extra step of sorting the final reduced data which is not a part of the traditional MapReduce workflow and is also not the focus of ZipThru's optimizations. As a result this step is omitted in the baseline Phoenix experiments as well as in ZipThru.

Phoenix also employs a relatively lazy/opportunistic memory allocation scheme for Map threads when saving intermediate state for the Reduce phase. This can lead to an inordinately high amount of calls to malloc by each thread when working on really large datasets, which often has deleterious effects on performance that have nothing to do with the workload or the MapReduce model.

Due to the automatic memory reduction provided by processing tuples on the fly ZipThru's implementation does not suffer from this problem. Thus to ensure fairer comparison the baseline also includes a more greedy memory allocation scheme that reduces the number of individual thread requests for more memory.

4.1 Datasets

I evaluate ZipThru on 5 different datasets, each with varying degrees of skew, in order to provide a broad view of the performance benefits that come from optimizing for popular keys. The datasets and the coverage of their top 10% of keys are shown in Table 4.1. Twitter was obtained from the Laboratory for Web Algorithmics (LAW) [16][13][14], while Friendster, Stackoverflow, Orkut, and Amazon Books were obtained from the Stanford Network Analysis Platform (SNAP) [15]. In order to make the datasets suitable for MapReduce some of them, such as Amazon Books, had to undergo transformation/reorganization.

The workload is a simple ranking algorithm, akin to a single iteration of Google's PageRank, which determines the relative importance of the different keys in the datasets based on reviews (for products) and edges (for social networks). The broad applicability of ranking

algorithms is easy to see. For social networks and web traffic ranking allows companies to know which pages, people, and topics are popular/trending on any given day or for any given slice of time. For web stores and online entertainment ranking tells companies which of their products, in which categories, are being bought or viewed the most by their users.

Dataset	Keys	Tuples	10% coverage
Twitter	35.6 M	1.47 B	76% of tuples
Friendster	64.9 M	1.81 B	59% of tuples
Stackoverflow	2.3 M	63.4 M	72% of tuples
Orkut	3.0 M	117 M	39% of tuples
Amazon Books	2.3 M	22.5 M	68% of tuples

Table 4.1. Datasets used to evaluate ZipThru

I test each dataset multiple times, running each under baseline Phoenix and then under ZipThru with static partitioning and dynamic load balancing. The smaller datasets (Stackoverflow, Orkut, and Amazon Books) have runtimes in seconds, so to verify stability of the results I run those datasets 10 times each. For the larger datasets, whose runtimes stretch into minutes, 2 runs proved sufficient. I compute the average runtime and memory access count for each dataset and use those to compute speedup and memory access savings.

4.2 ZipThru configuaration

ZipThru's runs use a bootstrap list obtained from running a traditional MapReduction on a 5% sample of the full datasets. As mentioned in Chapter 3.3 the cost of this preprocessing step is a fraction of the overall execution time and would be amortized over hundreds/thousands runs before the list needs to be updated.

ZipThru allows the user to specify how many keys to treat as popular from the bootstrap list provided at the start of execution. Selecting this number involves balancing tuple coverage with on-chip capacity. As a result the smaller datasets cache 512 K keys while the larger ones, due to the number of keys and tuples they have, cache 3 million. The percentage of tuples covered in each dataset is shown in Table 4.2

Dataset	No. of cached keys	% of tuples covered
Twitter	3 M	73%
Friendster	3 M	38%
Stackoverflow	512 K	85%
Orkut	512 K	52%
Amazon Books	512 K	81%

Table 4.2. Coverage of keys cached by ZipThru

To properly evaluate ZipThru's load balancing schemes, I test static partitioning with a variety of thread splits and dynamic load balancing with different high and low watermarks.

The results in Chapter 5, however, only show the best of these runs. As mentioned in Chapter 3.4, experimentation revealed that for most datasets an even split of the hardware's threads is enough to provide a good performance boost over the baseline.

For dynamic load balancing I use a high watermark of 100% and a low watermark of 12.5%, implicitly prioritizing mapping for threads in ZipThru. Among the watermark values surveyed these provided the best performance for the test datasets.

4.3 Hardware

To test performance I run ZipThru and the baseline configuration on two different machines, shown in Table 4.3. Both machines were selected due to the difference in available resources. While their clock speeds are fairly identical, the AMD machine has double the cores of the Intel Xeon and 3.2x the LLC capacity.

	Intel Xeon E5-2623 v4	AMD Opteron 6320
Clock speed	2.6 GHz	2.8 GHz
Cores	8 cores/16 threads/2 sockets	16 cores/32 threads/4 sockets
LLC capacity	20 MB	64 MB
DRAM capacity	125 GB	256 GB

Table 4.3. Hardware used to evaluate ZipThru

Both machines also have significant DRAM capacity, nullifying any worries of paging due to the sizes of the datasets evaluated.

To measure runtimes I use the built-in system timers on both machines. For memory accesses I use the CPU's hardware performance counters. I count the number of DRAM column requests made by both ZipThru and the baseline during execution. Unfortunately AMD offers poor support for its off-chip memory performance counters in Linux, and so no memory access results are shown for the AMD machine.

I present and discuss the results in the next chapter.

5. RESULTS AND ANALYSIS

5.1 Memory Accesses

First I show the effects that ZipThru's concurrent execution and immediate reduction of only popular keys have on off-chip misses. As mentioned in the previous chapter, AMD's poor support for off-chip memory performance counters in Linux made it impossible to collect those results for the Opteron machine.

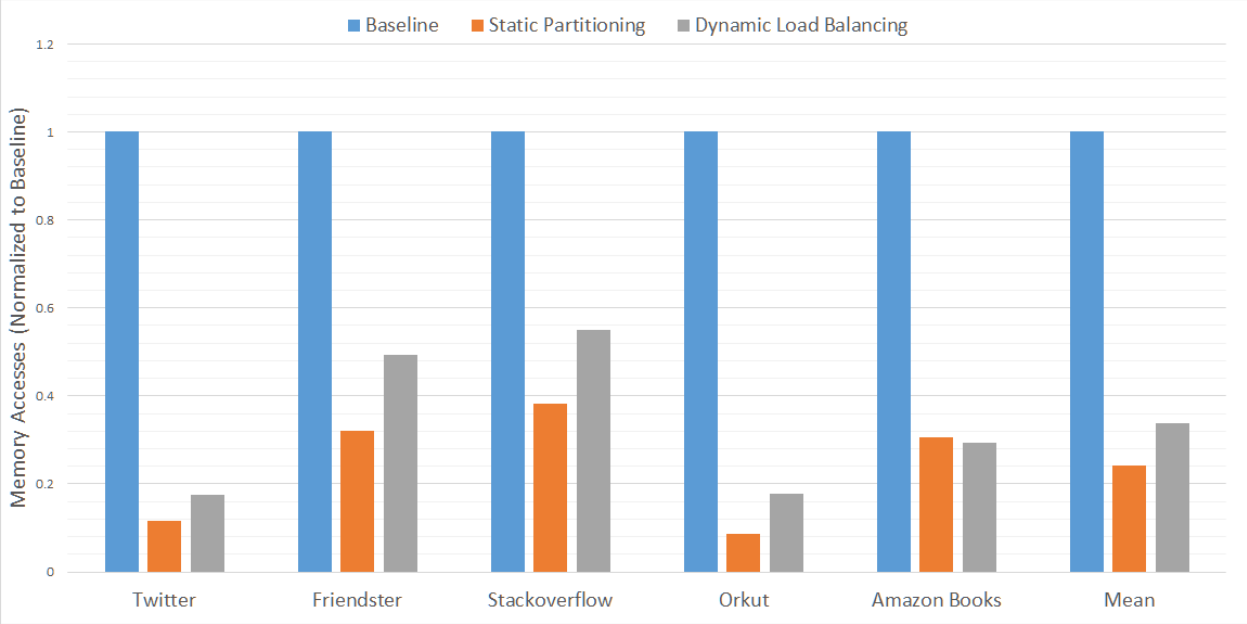


Figure 5.1. Memory access counts on Intel Xeon

Memory access counts for the Intel Xeon machine are shown in Figure 5.1. The Y-axis plots the access counts for each dataset, normalized to 1 for the baseline. The X-axis plots the results from 3 different configs: Baseline, ZipThru's Static Partitioning, and ZipThru's Dynamic Load Balancing. The average memory access savings across all datasets is shown on the far right.

Figure 5.1 shows that by consuming/reducing the popular keys' intermediate state as it is generated ZipThru saves considerably on memory accesses, with an average of 76% under static partitioning and 67% under dynamic load balancing. This, as mentioned in Chapter 3.2, is both because of the small size of the reduced state of popular minority and the much

smaller size of the secondary intermediate state of the unpopular keys in comparison to the global intermediate state created in the baseline.

With the exception of Amazon Books (the smallest dataset), static partitioning consistently provides better cache savings than dynamic load balancing. This is due to more efficient use of the private on-chip caches guaranteed by threads in static partitioning. Under static partitioning mappers need only hold the state for the input they're processing in their caches and reducers need only hold the reduced/intermediate state for their assigned keys. As a result there is higher locality for these accesses in the private caches of both task types and therefore less pressure on the lower level caches and DRAM.

In contrast dynamic load balancing forces threads to switch between mapping and reducing, and therefore hold different types of data with different kinds of access patterns in their private caches. Having the input data compete with the reduced and intermediate states as threads switch between tasks inevitably leads to more conflict and capacity misses for the data, and as a result more accesses to memory. Despite this ZipThru still provides significant memory savings over the baseline.

Next I show the effect ZipThru's memory access savings has on performance on both machines.

5.2 Performance

Figure 5.2 shows the speedups of the test workload across all datasets on the Intel Xeon machine. The Y-axis plots the performance for each dataset, normalized to 1 for the baseline. Unlike Figure 5.1, Figure 5.2 plots speedup i.e., the performance improvement of ZipThru over the baseline. The average speedup across all datasets is plotted on the far right.

The largest datasets Twitter and Friendster, with 10's of millions of keys and more than a billion tuples, show the best improvements over the baseline. Twitter reports a 3.87x/3.81x improvement while Friendster reports 5.52x/6.07x for static partitioning/dynamic load balancing respectively. Such a huge performance boost is consistent with sizes of the datasets and the capabilities of the test machine. The Intel Xeon machine has only 20 MB of on-chip cache. For especially large datasets the poorly optimized sequential MapReduce model

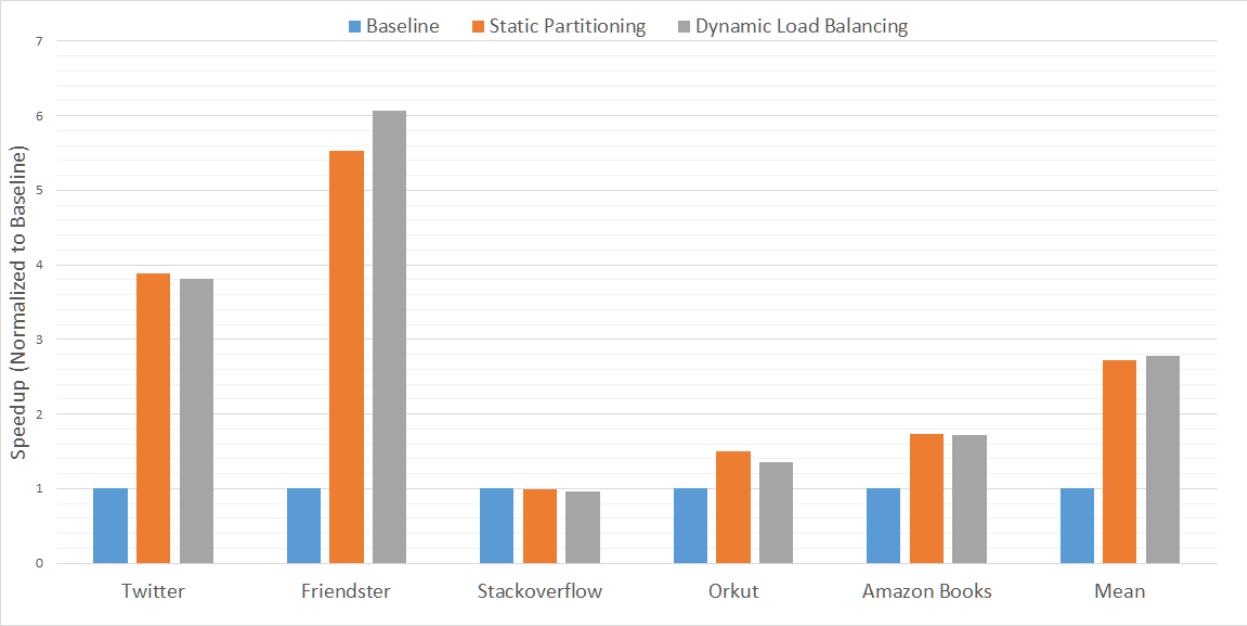


Figure 5.2. Speedup on Intel Xeon

would put a lot of pressure on this limited cache capacity, leading to very costly misses and significant slowdowns.

The other datasets show more modest but still impressive speedups. Orkut and Amazon Books both show 1.49x/1.34x and 1.73x/1.72x speedups for static and dynamic load balancing respectively. Stackoverflow is the only dataset that fails to show any speedup over the baseline, with 0.99x for static partitioning and 0.95x for dynamic load balancing. Incidentally Stackoverflow also shows the smallest savings in memory accesses over the baseline despite its 2.3 million key count. In fact it has savings comparable to Friendster, a dataset with 64 million keys and two orders of magnitude more tuples.

This indicates that the memory access patterns for the baseline in Stackoverflow are not quite as damaging to its performance as the other datasets', leading to less opportunity for ZipThru's savings to boost performance.

ZipThru's architecture also comes with its own set of unique overheads: mapper blocking and reducer idling due to full and empty queues at various points in execution (Chapter 3.4). For datasets with less costly access patterns this overhead further limits how much

performance improvement ZipThru can offer and, as seen from Stackoverflow's dynamic results, could even lead to slowdowns.

Overall there is little difference between the performance benefits offered by static partitioning vs dynamic load balancing across all datasets. The simplicity of dynamic load balancing from the user's perspective makes it the more attractive option, however, since there is no need to experiment with various thread splits in order to discover the optimal balance for mappers and reducers.

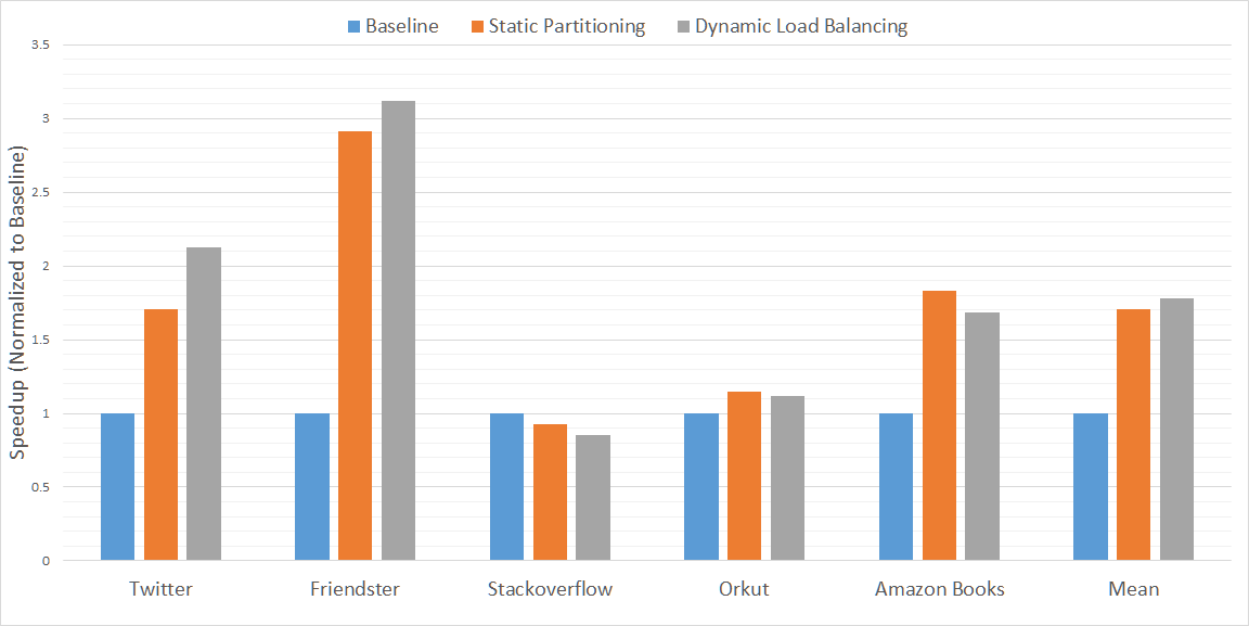


Figure 5.3. Speedup on AMD Opteron

Figure 5.3 shows the speedup of all datasets on the AMD Opteron machine. Like Figure 5.2 the Y-axis is normalized to 1 for the baseline, and the average speedup is displayed on the far right.

As expected, the more powerful AMD machine with its larger on-chip cache shows a much smaller performance improvement for ZipThru than the Intel machine. Twitter and Friendster, while still the best performing datasets, now have static/dynamic speedups of 1.70x/2.12x and 2.91x/3.12x respectively. The AMD machine's twice as many cores and 64 MB cache provided benefits to both the baseline and ZipThru, but because ZipThru's optimizations specifically target memory accesses its delta with the baseline shrunk.

Stackoverflow, already barely breaking on the weaker Xeon machine, shows slowdowns of 0.92x/0.85x for static partitioning/dynamic load balancing on the Opteron. Here, with even less pressure applied to the baseline's caches due to increased capacity, ZipThru's overheads slow it down considerably.

Orkut, much like Twitter and Friendster, also shows more modest speedups of 1.15x and 1.11x for the static and dynamic schemes. Amazon Books, however, with its small key and tuple count, does even better on ZipThru with the more powerful machine for static partitioning (1.83x on AMD vs 1.73x on Intel) and slightly worse for dynamic load balancing (1.68x on AMD vs 1.72x on Intel).

Overall the trends seen on the Intel Xeon machine hold on the AMD Opteron, with Twitter and Friendster performing best, and Stackoverflow performing worst. As was the case with the Xeon, the results for static partitioning and dynamic load balancing are quite similar when averaged across all datasets.

These results validate ZipThru's central thesis of boosting performance by saving on memory accesses and improving reuse of intermediate state. For large datasets running under the MapReduce model, ZipThru's concurrent execution and exploitation of Zipfian skew provides significant benefits over the traditional data-agnostic sequential MapReduce paradigm. ZipThru shows average memory access savings of 72% on the 16-core Intel Xeon machine and average speedups of 2.75x and 1.73x on the Intel system and the 32-core AMD Opteron respectively.

6. RELATED WORKS

Most work on speeding up MapReduce workloads has mainly targeted the integration of FPGAs into the MapReduce workflow, with a key focus on improving the compute performance, efficiency, and programmability of these new heterogeneous architectures. Works like [6] and [7], for example, focus on integrating execution between multiple different and disparate architectures. In [6] authors design a heterogeneous compute cluster, with each node consisting of a combination of FPGAs, CPUs and GPUs which are selected based on the specific workload being performed on the node. [7]'s single chip design uses only CPUs and GPUs, dividing Map and Reduce tasks between them and using a novel pipeline design to facilitate task splitting and coordination between both kinds of devices. [8] designs specialized FPGA-based hardware for Map tasks and a shared, application-based configurable accelerator for Reduce tasks. [9] focuses on integration of a reconfigurable fabric of FPGAs connected to servers via PCIe, with the goal of dynamically scaling up and down the number of accelerators used for analysis based on the current workloads on the servers. Due to difficulty in programming and designing FPGAs works like [17]–[19], seek to abstract low level FPGA intricacies away from developers, creating custom MapReduce frameworks specifically designed for use on FPGAs.

Instead of attempting to speedup execution by creating a new kind of heterogeneous architecture, ZipThru transparently extends traditional MapReduce and runs on off-the-shelf hardware, relying on its novel software techniques to boost reuse of popular keys' state and by extension application performance.

Works like [20], [21], and [22] attempt to optimize the performance of graph analytic frameworks such as Ligra[23], GraphLab[24], and Pregel[25], while also targeting graphs that possess an inherent Power Law skew. [20] is hardware oriented, focusing on augmenting the replacement policy of last-level caches in order to favor "hot" or popular vertices and prevent their eviction, while [22] provides software techniques that modify the graph's representation in memory in order cut down random accesses to DRAM and make all off-chip accesses sequential.

In contrast ZipThru is completely hardware-agnostic and targets MapReduce, an analytic framework that can be used for many different kinds of datasets and workloads, not just graph-specific analysis. Both [20] and [22] also require significant pre-processing/re-ordering of the target dataset in order to make its features identifiable to their schemes. ZipThru on the other hand only requires a quick, easily amortized analysis on a subset of the data to identify the popular keys to cache.

The idea of prioritizing content based on its popularity is quite old, specifically in the context of the Internet and its related technologies [26]–[28]. Due to the explosion of high resolution content on the Internet more recent work has focused specifically on information/content centric networks. Works like [29]–[31], for example, discuss analyzing content popularity on the fly, adjusting the likelihood of caching packets or objects based on requests/downloads and their predicted future popularity.

ZipThru does not include any predictive schemes and does not attempt to analyze the popularity of the keys as it is performing the MapReduction. As mentioned earlier, ZipThru is also focused specifically on Big Data analysis and MapReduce, and as such does not target content delivery and traffic routing based on packet popularity.

7. CONCLUSION

MapReduce is a popular framework used for performing Big Data analysis, favored for its easy programmability due to built-in support for fault tolerance and task and data management. While originally designed for distributed computing on networked clusters the MapReduce model is also applicable to datasets that can be analyzed efficiently on a single server. In this single-server context the challenges presented to MapReduce revolve around efficient use of local CPU resources, particularly the on-chip caches used to hold the data during execution.

Despite the well known fact that many big datasets possess a Zipfian/Power Law skew, traditional MapReduce frameworks are largely data-agnostic. Their Map-then-Reduce sequential mode of execution thus leads to distant reuse of the intermediate state of popular keys, resulting in poor locality and numerous off-chip memory accesses during execution. This in turn has a degrading effect on performance.

To address this I propose ZipThru, a novel MapReduce software architecture that exploits Zipfian skew in big datasets to process the state for popular keys on the fly, improving reuse, preserving locality, and boosting performance. ZipThru does this via four key mechanisms: 1) Concurrent execution of Map and Reduce phases; 2) Holding only the small, reduced state of the popular keys on-chip during execution; 3) Using a lookup table built from preprocessing a subset of the input to distinguish between popular and unpopular keys; and 4) Load balancing the concurrently executing Map and Reduce tasks based on the demands of the dataset being analyzed.

Comparisons using Phoenix, a shared-memory MapReduce implementation, reveal that for summarization workloads ZipThru provides an average memory savings of 72% on surveyed datasets, with average speedups of 2.75x and 1.73x on 16- and 32-core servers respectively.

By concurrently executing the Map and Reduce phases of execution and boosting reuse of popular keys' state ZipThru significantly improves the performance of single-server MapReduce workloads, and due to the broad applicability of the MapReduce model, offers performance improvements to local execution on multi-cluster workloads as well.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2007 Eurosys Conference*, Association for Computing Machinery, Inc., Mar. 2007. [Online]. Available: https://www.microsoft.com/en-us/research/publication/dryad-distributed-data-parallel-programs-from-sequential-building-blocks/.

[3] *Apache hadoop*, https://hadoop.apache.org/, 2018. [Online]. Available: https://hadoop.apache.org/.

[4] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990, ISSN: 0001-0782. DOI: 10.1145/79173.79181. [Online]. Available: http://doi.acm.org/10.1145/79173.79181.

[5] *Apache spark*, https://spark.apache.org/, 2018. [Online]. Available: https://spark.apache.org/.

[6] K. H. Tsoi and W. Luk, "Axel: A heterogeneous cluster with fpgas and gpus," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10, Monterey, California, USA: ACM, 2010, pp. 115–124, ISBN: 978-1-60558-911-4. DOI: 10.1145/1723112.1723134. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723134.

[7] L. Chen, X. Huo, and G. Agrawal, "Accelerating mapreduce on a coupled cpu-gpu architecture," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Salt Lake City, Utah: IEEE Computer Society Press, 2012, 25:1–25:11, ISBN: 978-1-4673-0804-5. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389030.

[8] C. Kachris, D. Diamantopoulos, G. C. Sirakoulis, and D. Soudris, "An fpga-based integrated mapreduce accelerator platform," *Journal of Signal Processing Systems*, vol. 87, no. 3, pp. 357–369, Jun. 2017, ISSN: 1939-8115. DOI: 10.1007/s11265-016-1108-7. [Online]. Available: https://doi.org/10.1007/s11265-016-1108-7.

[9] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter

services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 13–24, ISBN: 978-1-4799-4394-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665678.

[10] S. Fagan and R. Gencay, "An introduction to textual econometrics," in *Handbook of Empirical Economics and Finance*. CRC Press, 2010, pp. 133–153.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Feb. 2007, pp. 13–24. DOI: 10.1109/HPCA.2007.346181.

[12] Tien-Fu Chen and Jean-Loup Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[13] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, Manhattan, USA: ACM Press, 2004, pp. 595–601.

[14] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds., ACM Press, 2011, pp. 587–596.

[15] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, http://snap.stanford.edu/data, Jun. 2014.

[16] *Laboratory for web algorithmics*, http://law.di.unimi.it/datasets.php, 2018. [Online]. Available: http://law.di.unimi.it/datasets.php.

[17] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10, Monterey, California, USA: ACM, 2010, pp. 93–102, ISBN: 978-1-60558-911-4. DOI: 10.1145/1723112.1723129. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723129.

[18] D. Diamantopoulos and C. Kachris, "High-level synthesizable dataflow mapreduce accelerator for fpga-coupled data centers," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul. 2015, pp. 26–33. DOI: 10.1109/SAMOS.2015.7363656.

[19] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze fpga accelerator deployment at datacenter scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16, Santa Clara, CA, USA: ACM, 2016, pp. 456–469, ISBN: 978-1-4503-4525-5. DOI: 10.1145/2987550.2987569. [Online]. Available: http://doi.acm.org/10.1145/2987550.2987569.

[20] P. Faldu, J. Diamond, and B. Grot, *Domain-specialized cache management for graph analytics*, 2020. arXiv: 2001.09783 [cs.DC].

[21] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," Oct. 2012, pp. 17–30.

[22] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 293–302.

[23] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, no. 8, pp. 135–146, Feb. 2013, ISSN: 0362-1340. DOI: 10.1145/2517327.2442530. [Online]. Available: https://doi.org/10.1145/2517327.2442530.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, *Graphlab: A new framework for parallel machine learning*, 2010. arXiv: 1006.4990 [cs.LG].

[25] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," Jan. 2009, p. 48. DOI: 10.1145/1582716.1582723.

[26] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet," *Glottometrics*, vol. 3, pp. 143–150, 2002.

[27] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM*, 1999.

[28] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '99, Cambridge, Massachusetts, USA: Association for Computing Machinery, 1999, pp. 251–262, ISBN: 1581131356. DOI: 10.1145/316188.316229. [Online]. Available: https://doi.org/10.1145/316188.316229.

[29] K. Suksomboon, S. Tarnoi, Y. Ji, M. Koibuchi, K. Fukuda, S. Abe, N. Motonori, M. Aoki, S. Urushidani, and S. Yamada, "Popcache: Cache more or less based on content

popularity for information-centric networking," in *38th Annual IEEE Conference on Local Computer Networks*, Oct. 2013, pp. 236–243. DOI: 10.1109/LCN.2013.6761239.

[30] S. Li, J. Xu, M. van der Schaar, and W. Li, "Popularity-driven content caching," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, Apr. 2016, pp. 1–9. DOI: 10.1109/INFOCOM.2016.7524381.

[31] C. Bernardini, T. Silverston, and O. Festor, "Mpc: Popularity-based caching strategy for content centric networks," in *2013 IEEE International Conference on Communications (ICC)*, Jun. 2013, pp. 3619–3623. DOI: 10.1109/ICC.2013.6655114.

VITA

Ejebagom John Ojogbo was born in Ibadan, Oyo State, Nigeria, and grew up in Abuja, Nigeria. He has a Bachelor of Science degree in Computer Science from Fisk University (2012), and a Bachelor of Engineering degree in Electrical Engineering from Vanderbilt University (2013). He began pursuing his PhD at Purdue University in 2013, and over the course of his studies has worked both as a Research Assistant for the School of Electrical and Computer Engineering and as a developer for the Purdue University Cyber Center. He has interned for different companies, including ARM Inc, where he worked with the Performance Modeling team on improving the models and tools used in designing their cache-coherent interconnects. As a graduate student his work has focused on Computer Architecture, Security, and Big Data Analysis.

His interests include reading science fiction and fantasy novels, as well as following current affairs in technology and new advances in the computing industry.