View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Eastern Washington University: EWU Digital Commons

Eastern Washington University
EWU Digital Commons

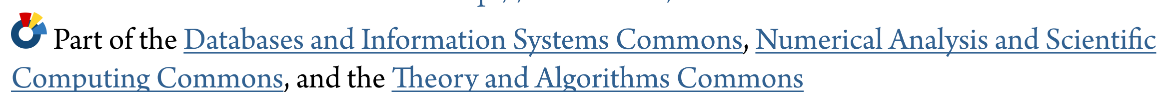
EWU Masters Thesis Collection | Student Research and Creative Works

Winter 2018

Evaluating a Cluster of Low-Power ARM64 Single-Board Computers with MapReduce

Daniel McDermott
Eastern Washington University

Follow this and additional works at: http://dc.ewu.edu/theses


Part of the Databases and Information Systems Commons, Numerical Analysis and Scientific Computing Commons, and the Theory and Algorithms Commons

Recommended Citation

McDermott, Daniel, "Evaluating a Cluster of Low-Power ARM64 Single-Board Computers with MapReduce" (2018). *EWU Masters Thesis Collection*. 474.
http://dc.ewu.edu/theses/474

This Thesis is brought to you for free and open access by the Student Research and Creative Works at EWU Digital Commons. It has been accepted for inclusion in EWU Masters Thesis Collection by an authorized administrator of EWU Digital Commons. For more information, please contact jotto@ewu.edu.

Evaluating a Cluster of Low-Power ARM64 Single-Board Computers with MapReduce

---

A Thesis

Presented To

Eastern Washington University

Cheney, Washington

In Partial Fulfillment of the Requirements

for the Degree

Master of Science in Computer Science

By

Daniel McDermott

Winter 2018

Thesis of Daniel McDermott Approved by

BOJIAN XU, GRADUATE STUDY COMMITTEE CHAIR DATE

STUART STEINER, GRADUATE STUDY COMMITTEE MEMBER DATE

SAQER ALHLOUL, GRADUATE STUDY COMMITTEE MEMBER DATE

Master's Thesis

In presenting this thesis in partial fulfillment of the requirements for a masters degree at Eastern Washington University, I agree that the JFK Library shall make copies freely available for inspection. I further agree that copying of this project in whole or in part is allowable only for scholarly purposes. It is understood, however, that any copying or publication of this thesis for commercial purposes, or for financial gain, shall not be allowed without my written permission.

______________________________ ______________

SIGNATURE DATE

Abstract

With the meteoric rise of enormous data collection in science, industry, and the cloud, methods for processing massive datasets have become more crucial than ever. MapReduce is a restricted programing model for expressing parallel computations as simple serial functions, and an execution framework for distributing those computations over large datasets residing on clusters of commodity hardware. MapReduce abstracts away the challenging low-level synchronization and scalability details which parallel and distributed computing often necessitate, reducing the concept burden on programmers and scientists who require data processing at-scale.

Typically, MapReduce clusters are implemented using inexpensive commodity hardware, emphasizing quantity over quality due to the fault-tolerant nature of the MapReduce execution framework. The nascent explosion of inexpensive single-board computers designed around multi-core 64bit ARM processors, such as the RasberryPi 3, Pine64, and Odroid C2, has opened new avenues for inexpensive and low-power cluster computing.

In this thesis, we implement a novel cluster around low-power ARM64 single-board computers and the Disco Python MapReduce execution framework. We use MapReduce to empirically evaluate our cluster by solving the Word Count and Inverted Link Index problems for the Wikipedia article dataset. We benchmark our MapReduce solutions against local solutions of the same algorithms for a conventional low-power x86 platform. We show our cluster out-performs the conventional platform for larger benchmarks, thus demonstrating low-power single-board computers as a viable avenue for data-intensive cluster computing.

Contents

1 Introduction 4

1.1 Motivation and Goals . . . 4

1.2 Thesis Structure . . . 6

2 Background 7

2.1 Data-Intensive Cluster Computing . . . 7

2.2 Low-Power Hardware . . . 8

2.3 MapReduce Programming Pattern . . . 9

2.3.1 Map . . . 10

2.3.2 Fold / Reduce . . . 11

2.4 MapReduce Execution Framework . . . 11

2.4.1 Mappers and Reducers . . . 12

2.4.2 Distributed File System . . . 14

2.5 Text Processing with MapReduce . . . 16

2.5.1 Word Count . . . 16

2.5.2 Inverted Index . . . 17

2.6 Related Work . . . 19

3 Implementation 20

3.1 Low-Power SBC Cluster . . . 20

3.1.1 SBC Nodes . . . 20

3.1.2 Cluster Specifications . . . 21

3.2 Disco MapReduce Framework . . . 22

3.3 Conventional x86 Platform . . . 24

3.4 Text Processing with Wikipedia and Python . . . 25

3.4.1 Wikipedia Article Dataset . . . 25

3.4.2 Word Count and Inverted Link Index . . . 26

3.4.3 Implementation Variants . . . 27

4 Evaluation and Analysis 29

4.1 Benchmark Setup . . . 29

4.2 Execution Time . . . 29

4.3 Memory Consumption . . . 31

4.4 Power Consumption . . . 32

4.5 Throughput Analysis . . . 33

5 Conclusions and Future Work **36**

5.1 Future Work . . . 36

6 Additional Materials **38**

6.1 Source Code . . . 38

6.2 Cluster Construction . . . 38

6.3 Slides . . . 38

7 Data **39**

List of Figures

1 Data Flow in MapReduce . . . 10

2 Map . . . 10

3 Fold / Reduce . . . 11

4 KeyValue Flow . . . 13

5 MapReduce Execution [8] . . . 13

6 Distributed File System [19] . . . 15

7 Word Count Execution [19] . . . 17

8 Inverted Index Execution [19] . . . 18

9 Cluster Hardware . . . 20

10 Cluster Network . . . 22

11 Word Count Execution Time . . . 30

12 Inverted Link Index Execution Time . . . 30

13 Per-Task Memory Usage . . . 32

14 Word Count Throughput . . . 34

15 Inverted Link Index Throughput . . . 34

1 Introduction

MapReduce is a programing pattern and execution framework for transparently distributing computation across large clusters of commodity computers. Conceived as an execution framework for massive data processing by Google in their seminal 2008 paper [8], MapReduce finds its roots in a common pattern of the functional programming paradigm often termed "map and fold" or "map and accumulate". Users write a *map* function which processes raw data to produce an intermediate list of key/value pairs, and a *reduce* function which performs some aggregation of all intermediate values by their associated key. Many real-world data processing problems can be expressed in terms of this simple pattern, and programs expressed this way have desirable properties which facilitate natural parallelism. As an execution framework, the programming pattern is built upon and extended with fault-tolerant distributed file systems and workload scheduling. When taken together, the pattern and framework enable programs to be seamlessly executed in parallel across large clusters of potentially unreliable machines.

In this thesis, we introduce a novel MapReduce cluster composed of small, inexpensive, low-power ARM64 single-board computers (SBCs). We use MapReduce, both as a local programming pattern and as a distributed execution framework, to implement data-intensive benchmarks for evaluating the performance of our low-power SBC cluster against a conventional low-power x86 platform. This thesis predominately explores the trade-offs between scale-up and scale-out architectures for data-intensive computing with low-power machines.

1.1 Motivation and Goals

MapReduce, a simple idea at its core, has become of fundamental importance for modern data processing. Many popular NoSQL databases and other distributed systems for Big Data management find their underlying implementation relying on MapReduce in some capacity [4] [21]. At its inception, MapReduce played a central role in Google's architecture to process a significant portion of the World Wide Web on a daily basis.

However, as data volumes become enormous, so do the energy and space requirements of clusters for processing such workloads. Reducing power consumption and increasing density have become a principal concerns in modern data center architecture, precipitating movements such as "Green Computing" [29]. While the x86 processor architecture has dominated the data center market for more than twenty years, the ARM processor architecture has a proven track record in power and space constrained applications. With the extension to a multi-core 64bit architecture, ARM has recently seen application beyond its traditional role in mobile and

embedded computing. Mainstream CPU manufacturer AMD has begun producing ARM64 chips for use in servers [1], and there has been an explosion of single-board computers designed around inexpensive ARM64 processors for the hobbyist market [31] [28] [24]. Some hobbyists and enterprises have recently begun exploring computing architectures composed of numerous small ARM64 computers as alternatives to larger x86-based server platforms [32] [27], and there has been past research into the efficiency of small-node cluster architectures for data-intensive computing [3] [18].

Our work uses MapReduce to evaluate such an architecture. We construct a MapReduce cluster around the Odroid-C2 [24] ARM64 low-power single-board computer (SBC) and the Disco [11] Python MapReduce framework. We evaluate our SBC cluster against a conventional low-power x86 platform by implementing several data intensive text processing benchmarks. While the x86 processor family generally delivers considerably more instructions per CPU cycle and greater memory bandwidth than the average ARM processor [26], data intensive text processing is an inherently I/O-bound activity [19]. We will show that a cluster of inexpensive nodes can outperform a single more costly node from a text processing throughput perspective, when memory and power consumption are considered.

Our text processing benchmarks are based on solving the Word Count and Inverted Link Index problems for the Wikipedia article dataset. We implement each solution for both our MapReduce cluster (distributed) and our conventional x86 platform (non-distributed). While benchmarks for our cluster are inherently phrased as MapReduce solutions, so as to execute on the Disco MapReduce framework, we also utilize MapReduce as a parallel programming pattern in some of our non-distributed x86 platform benchmarks.

As the MapReduce framework is inherently external in its execution (i.e. it writes most intermediate work to disk before processing another record), we implement external variants as well as in-memory variants of each solution for our x86 platform. When evaluating benchmarks, we will consider in-memory versus external solutions by factoring peak per-record memory consumption into the final throughput measurements. Peak per-record memory consumption serves as a good analogue for a solution's ability to scale-out versus scale-up. Algorithms and implementations which cannot perform in an external context or do not have a constant space complexity are unsuitable for scale-out computing architecture, and thus unsuitable for massive data processing generally.

Finally, as power consumption is becoming an increasing fraction of total cost of ownership of cluster computers, sometimes up to 50% over a three year lifespan [17], we will consider the power consumption of benchmarks in our final throughput calculations.

As a side-effect to the primary goals stated above, we will provide detailed instructions

on constructing a MapReduce cluster and example code for processing a dataset of theoretical interest. Exploring MapReduce from an empirical perspective can be inaccessible to many students and researchers due to the high costs associated with traditional cluster computing. Furthermore, mainstream MapReduce frameworks such as Apache Hadoop [37] come saddled with an enormous ecosystem of libraries, utilities, and side-projects which can be overwhelming to learn and cumbersome to orchestrate, especially to those seeking to study MapReduce for its elegant simplifications.

These costs and complexities motivate us to provide a more accessible model for pedagogy and research. Our MapReduce cluster is designed around the lightweight Disco framework and inexpensive SBCs intended for the hobbyist market. Our cluster consumes less power at idle load than a laptop computer and is so small it will fit into an office desk drawer. The low unit cost of hobbyist SBCs puts purchasing the multiple nodes required for cluster computing within the financial reach of most students. The Disco MapReduce framework, being driven by the Python programming language [30], is much simpler to use than its popular industrial counterpart, Hadoop. Therefore, aside from our practical results, our work provides an accessible experimental model for MapReduce research.

1.2 Thesis Structure

Following this introduction, Chapter 2 gives context to our work by presenting an overview of data-intensive cluster computing and emerging low-power hardware architecture, introducing the MapReduce programming pattern and execution framework, and describing the word count and inverted index text processing problems generally and the MapReduce approach to solving them. Chapter 3 describes our novel low-power ARM64 cluster and our solutions to the Word Count and Inverted Link Index problems for the Wikipedia dataset. Chapter 4 presents our analysis of benchmark results comparing our distributed and local implementations of the solutions. The concluding Chapter 5 gives a summary of the achieved work and outlines potential future work.

2 Background

In this chapter we give an introduction to topics related to our work. We cover data-intensive computing generally, followed by a survey of new evolutions in low-power computing hardware. We then describe the MapReduce programming pattern and execution framework, and the Word Count and Inverted Index text processing problems and the MapReduce approach to solving them. We end with a brief survey of prior works related to our own.

2.1 Data-Intensive Cluster Computing

Cluster computing is of fundamental importance to modern data processing. The size of modern data processing workloads has grown well beyond the capacity of any individual machine; numerous machines must be grouped together to into a "cluster" to solve most massive data processing problems. These numerous individual machines are often orchestrated together as a single logical computer, under the direction of a distributed-parallel software execution framework such as MapReduce. Some of these clusters become large enough to occupy an industrial warehouse, giving rise to the notion of a "warehouse scale computer" [5].

In recent years, there has been a trend of diminishing returns in improving the performance of individual processing nodes [6]. CPU power consumption grows super-linearly with per-core performance [3], as does the financial cost and complexity of implementation [6]. A similar relationship is true of increasing the memory capacity attached to an individual CPU [14]. Thus the cost of scaling-up performance of individual computers has dramatically outpaced the cost of scaling-out cluster computers by adding additional nodes. This trade-off between vertical "scale-up" and horizontal "scale-out" computing architecture is well understood in the context of massive data processing [19]. Data processing is an inherently I/O-bound activity, where the ability to move data through the system - that is, the bandwidth capacity or throughput - generally outweigh concerns for latency and processor performance [5]. Data intensive cluster architecture optimizes for throughput, not latency; increasing storage and network bandwidth are principal concerns when designing data centers and clusters for handling data-intensive workloads.

Even when distributed across many machines, data-intensive workloads often exceed the size of any individual machine's main-memory capacity, thus "external" memories such as disk or network storage must be swapped to during processing. External memory algorithms are of key importance to data-intensive computing [36]. Though not explicitly an "external-memory algorithm" per-se, the MapReduce framework is inherently external in execution, as it writes most intermediate state to disk between processing each data record. For external

memory algorithms, data throughput to the external memory is more important that CPU performance, and is usually the gating factor in overall execution time. It is for this reason we will evaluate external as well as in-memory variants of our text processing benchmarks for comparison with our low-power MapReduce cluster.

2.2 Low-Power Hardware

Power consumption is of critical concern for large cluster computers, often representing a significant portion of their total cost of ownership. Data centers and warehouse-scale clusters are being constructed nearby hydroelectric power plants to discount their enormous power costs [17]. These enormous costs as well as the physical footprints of such computers have precipitated the emergence of "Green Computing" and similar paradigms for reducing power and space consumption in the data center [29]. Much of the recent progress in conventional data center hardware architecture has focused not on increasing absolute performance of individual machines, but rather on reducing power and space consumption while maintaining performance.

To this end, the ARM processor architecture has emerged as a tenuous challenger to x86 in the data center market. Historically, ARM has been dominant in power and space constrained embedded and mobile computing spaces. As ARM has evolved to include advanced superscalar multi-core 64bit microarchitectures, the performance of some high-end ARM64 chips has approached that typically seen of x86 desktops or servers. It seems natural then, given the increased concerns for power and space consumption combined with ARM's proven track record in low-power and space constrained applications, that ARM would eventually come to be a viable competitor to x86 for conventional general purpose computing.

As the ARM64 processor architecture gains share beyond mobile and embedded computing, new avenues are being opened for low-power cluster computing. There has been a recent explosion of inexpensive single-board computers designed around multi-core ARM64 CPUs, such as the RasberryPi 3 [31], Pine64 [28], and Odroid C2 [24]. Mainstream CPU manufacturer AMD has begun producing ARM64 CPUs for use in servers with their "Opteron A-Series" processor line [1]. Even some cloud computing providers have begun offering small bare-metal ARM64 servers as alternatives to similarly priced x86 virtual server slices [32].

Our low-power SBC cluster is constructed around the Odroid C2 "credit-card sized" ARM64 SBC. There is much architectural interchange between cellphone/mobile computing architecture and low-power ARM64 SBCs, since they tend to possess similar CPUs and only differ in their physical format, peripheral IO, and supported power states. SBCs are typically intended for use in multimedia or embedded applications. In this work, we will explore their

application to a more traditional data center workload; text processing with MapReduce.

2.3 MapReduce Programming Pattern

MapReduce finds it's roots in a common pattern of the purely functional programming paradigm. Pure functional programming differs from more common procedural and object-oriented programming paradigms in it tends to enforce properties of functions such as freedom from side-effects and referential transparency. In this section, we explain these properties and their importance to MapReduce as a method for facilitating parallel programming.

Ahmdahl's law [2] implies the theoretical speed-up gained from parallelizing a program will be bounded by the portion of the program which must execute serially. Parallel programs must execute serially, or synchronize, in order to deterministically modify shared state between multiple threads of execution. If there is no shared state in a parallel solution, then the speed-up gained from adding additional threads of execution should theoretically scale linearly with the number of threads, unbounded. Of course, in practice this is nearly always limited by hardware and systems overhead, or the need to ultimately synchronize state to arrive at a final solution, but nevertheless reducing shared state is a predominant design constraint when implementing parallel and distributed programs.

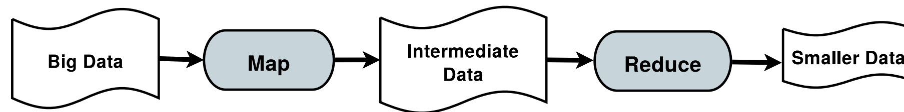
Modifying shared state from within a function is fundamentally a *side effect* of calling the function. Functions which are *pure* in the functional programming paradigm are *side effect free*, that is they depend only on their inputs and always create new outputs, no external or otherwise global shared state is modified by calling a *pure* function. Such functions are said to have *referential transparency*, that is they return the same output for a given input. As such functions do not depend on nor modify any shared or internal state, it is possible to make multiple calls to *pure* functions in parallel, one for each distinct input.

If an input dataset can be divided into many distinct records, and a data processing solution can be implemented in terms of *pure* functions, then the solution can be parallelized almost infinitely over the dataset, limited only by the need to synchronize or coalesce the results of these parallel function calls at the end of execution to produce a final result. This is the essence of MapReduce as a parallel programming pattern; many parallel calls to a *pure* function, each call operating over a distinct input record and producing an intermediate result, followed by a coalescing of all intermediate results into a final result. Operating a function over a set of distinct input records to produce an intermediate set of output records is often termed a *map*. The coalescing or accumulation of many records into a single, smaller, record is often termed a *fold* or *reduce*.

As a functional programming pattern, MapReduce is most frequently encountered under

the names “map and fold” or “map and accumulate”. The general data flow between these two concepts working in unison to process a large dataset is shown figure 1.

Figure 1: Data Flow in MapReduce



2.3.1 Map

Functional programs frequently operate over lists, more so than typical procedural or object-oriented programs. Another key concept of functional programming applied to MapReduce is that of *higher order* functions. Higher order functions are functions which take other functions as their arguments.

Map is a higher order function which takes as its input a function f and a list. *Map* then applies the input function f to every element of the list, producing a new list which is the result of that application. Conceptually, *map* produces a new output list which is f mapped onto the input list.

In a parallelized *map* implementation, each instance of f in figure 2 is executed in a distinct thread. Thus f must be a side-effect free *pure* function to be suitable for a parallel *map* implementation.

Function MAP(func, list)

Input: *list*: A list to iterate over. *func*: A function to apply to each element in the list, which returns a new element.

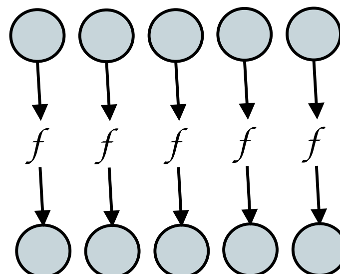
Output: A new list, representing *func* applied to every element of *list*.

newlist ⟵ **empty list**;

foreach *element in list* **do**

  newlist.append(*func*(*element*));

Figure 2: Map



2.3.2 Fold / Reduce

Reduce has many pseudonyms, and is commonly encountered under the names "fold" or "accumulate". Similar to *map*, *reduce* is a higher order function which takes a list and a function g as its inputs. In contrast to *map*, *reduce* requires an additional argument; an accumulator value which will be both passed as input to and returned as output from each successive call to the function g while iterating over the input list. This accumulator value is represented by the grey box in figure 3. Whereas *map* returns a new list as its output, *reduce* returns only the final value of the accumulator.

The accumulator value is a mechanism to preserve state between subsequent calls to the input function g while iteratively applying g to the input list. Thus reduce itself cannot be parallelized - each successive call to g depends on the result of a previous call to g. In the MapReduce programming pattern, *reduce* serves the purpose of coalescing or reducing the intermediate output of *map* into a smaller or otherwise useful final result.

Function FOLD(acc, func, list)

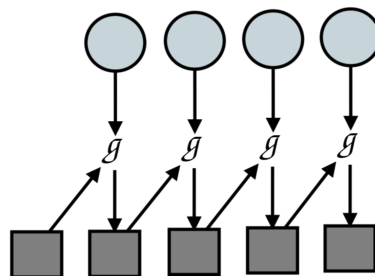
Input: *acc*: An accumulator value. *func*: A function which takes a list element and a value, then returns a new value. *list*: A list to iterate over.

Output: The final value of *acc*.

foreach *element in list* **do**

  $acc \longleftarrow func(acc, element)$;

Figure 3: Fold / Reduce



2.4 MapReduce Execution Framework

As a parallel cluster computing execution framework, the MapReduce programming model described in section 2.3 above is extended with fault-tolerant distributed workload scheduling and distributed filesystems. The core design which all MapReduce execution frameworks share was first described by Dean and Ghemawat of Google in 2008 [8] as a method for simplified data processing on large clusters.

As a distributed execution framework, the MapReduce runtime is generally split into two

phases; a Map phase and a Reduce phase, which conceptually mirror the *map* and *reduce* functions of the parallel functional programming pattern, extended into the distributed computing context.

2.4.1 Mappers and Reducers

Whereas the *map* and *reduce* of the functional programming pattern operate on lists and raw values, the Map and Reduce of the distributed execution framework operate over (*key*,*value*) tuples. This is in order to facilitate the movement of data through the distributed system by *key*, which can act as a conceptual tag for determining where to direct data in the system. Throughout the execution of the MapReduce runtime, *values* associated with a particular *key* are brought together on cluster nodes which have been assigned responsibility for processing that *key*.

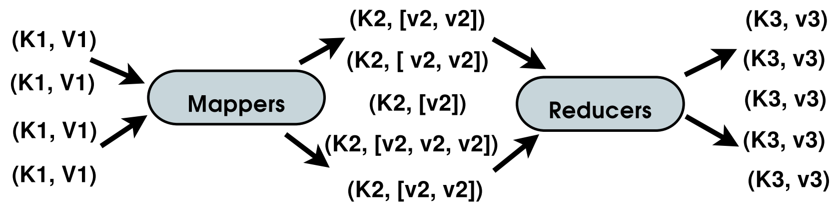
In many ways, the Map phase of execution serves to structure, filter, and tag an otherwise unstructured raw dataset. The structure taken on is that of *keys* (tags) and *values* (structured data), or in the broader computer science parlance, simply a dictionary. That is, Map is primarily a method for converting a flat list of unstructured input data into a dictionary. Often the size of the data output from the Map phase is on-order of the input size.

After the Map phase, the execution framework will bring together all *values* output with the same *key* to aggregate and sort them before passing to the Reduce phase. As all *values* for a given *key* represent a sorted list, each list and its *key* can be processed by a separate call to the Reduce function. In this way, the Reduce phase can be parallelized similar to Map. The output from the Reduce phase is a final list of (*key*, *value*) tuples. The conceptual flow of (*key*, *value*) tuples through the MapReduce execution framework is shown in figure 4, where:

- **(K1, V1)** is typically an input document identifier or filename (**K1**), and the document's raw contents (**V1**).
- **(K2, [v2, v2, ...])** is some intermediate key (**K2**) output by a Mapper, and all values associated with that key output by any Mappers, subsequently aggregated and sorted into a list (**[v2, v2, ...]**) by the execution framework.
- **(K3, v3)** is some final key (**K3**) output by a Reducer and the value (**v3**) resulting from some aggregation by the Reducer for that key.

In the cluster computing context, each potential hardware execution thread available in the cluster is a "Worker" for the MapReduce execution framework. Thus a cluster of 5 machines each having 4 CPU cores would supply 20 Workers to the execution framework. Map and

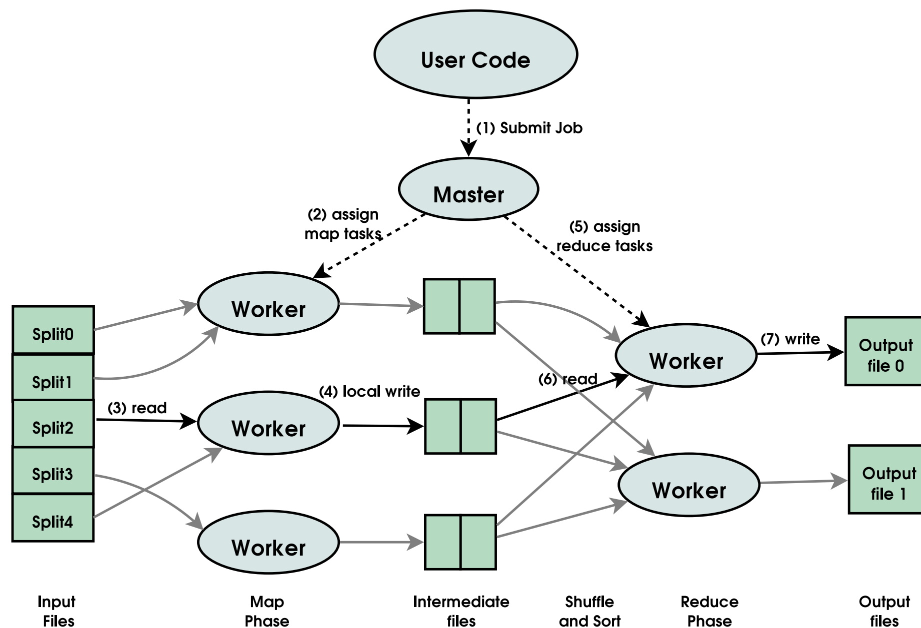
Figure 4: KeyValue Flow



Reduce functions are assigned to Workers as tasks during the execution of a MapReduce job. When a worker is executing a Map task, it is frequently referred to as a "Mapper", and when executing a Reduce task, it is a "Reducer" [19].

The overall flow of MapReduce execution as outlined by Dean and Ghemawat in the original MapReduce paper, and as implemented by the Disco [11] and Hadoop [37] frameworks, is shown in figure 5.

Figure 5: MapReduce Execution [8]



The execution steps as labeled in the figure [8]:

1. User code submits a MapReduce job to the cluster through a library or some other mechanism. The input dataset residing on the cluster is expected to have been split into M files by the user.

2. One node of the cluster is special - the master. The other nodes are workers and are assigned tasks by the master. There will be M map tasks and R reduce tasks to assign. The master assigns map tasks to workers which are preferably idle and have the input data stored locally.

3. A worker assigned a map task reads its input split and applies the user supplied function to be mapped over the dataset to each input.
4. As records from the input split are processed, they are written to the local disk of the map worker as intermediate results.
5. When enough intermediate results become available, the master notifies the reduce workers of the location of the data and they begin executing.
6. The reduce workers read data, possibly remotely over the network, from the map workers, and aggregate and sort the intermediate data by *key* before processing the data through the user defined reducer function.
7. As the reduce workers process the intermediate data, they write final results to their local disks. Once all final results are ready, the user program from step 1. is notified.

Altogether this would seem to be a complex framework, with much synchronization and orchestration required between moving parts. However, its important to note that users of the system need only understand the basics of the MapReduce functional programming pattern. All the challenging low-level details of parallel-distributed computing are hidden behind the execution framework. Users need only supply a dataset split, a mapper function, and a reducer function, to take full advantage of massive clusters using MapReduce.

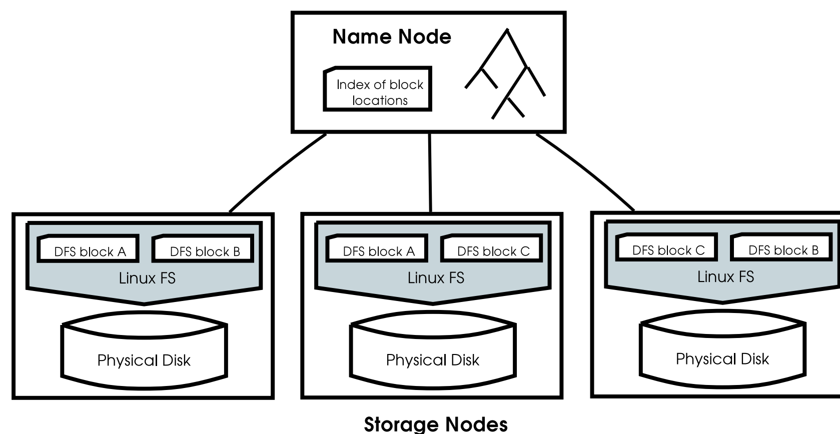
2.4.2 Distributed File System

The MapReduce framework was designed to handle data volumes which exceed the storage capacity of any single machine. Distributed file systems orchestrate data across multiple machines in a cluster, allowing data capacity to scale horizontally with the number of machines. Prime examples of such distributed filesystems are Google's Big Table [7] and the Google File System [13], as well as Hadoop's HDFS [37].

A key idea underlying the efficiency of the MapReduce execution framework is to move computation to data. Programs are small, but data is large - if programs can be co-located with the data they are dependent on, then data fetched remotely (over the network) can be minimized. In the context of MapReduce, a "program" is any individual mapper or reducer task. The MapReduce framework will attempt to schedule and assign a tasks to worker nodes such that tasks execute on nodes which host the data those tasks will consume, as much as is possible. Therefore, the MapReduce execution framework must have insight into the structure of the cluster's distributed file system, and frameworks and their file system implementations are typically tightly coupled. The framework used in our cluster, Disco, relies on DDFS, the Disco Distributed File System [11].

It is important to note, however, that MapReduce distributed file systems are not file systems in the traditional sense, in they do not manage underlying block devices. Rather, MapReduce file systems primarily serve as a mechanism to orchestrate data and know the location of many replicated data sources across a cluster, in order to assign tasks to nodes for optimal data-locality during execution. Underlying physical block device management is usually delegated to the operating system. MapReduce file systems function more like directed graphs for unstructured data blocks residing on top of many traditional Linux file systems distributed across the cluster. In MapReduce file systems, there is a "Name Node" which keeps an authoritative index, or representation of this graph, with tags (graph nodes) pointing to other tags or to blocks of data (leaf nodes) residing on one or more cluster nodes. It's worth mentioning that this Name Node can sometimes represent a single point of failure for the cluster.

Figure 6: Distributed File System [19]



Another key feature of distributed file systems which improves both fault-tolerance and data-locality is data replication. Figure 6 shows an example of a distributed file system with a replication factor of two. Notice that each data block resides on at least two nodes. Therefore, it is possible for exactly one node to fail, yet no data will be lost. More generally, a distributed file system with a replication factor of N can have at most $(N-1)$ nodes fail before experiencing data loss. This is a key property of distributed filesystems employed by MapReduce's fault tolerance, as MapReduce clusters are frequently implemented with low-cost and unreliable machines. Data replication can also aid the MapReduce execution framework in assigning tasks to nodes for maximum data locality. Higher replication factors give the execution framework more options in node assignment. Specifically, a task which depends on a particular data block can be assigned to any of N possible nodes in a cluster with a data replication factor of N.

2.5 Text Processing with MapReduce

Massive text processing is the quintessential data-intensive computing problem space. Everything from natural language processing, to web-crawling, to cloud-scale log parsing falls into the domain of massive text processing. For an excellent treatment of the subject of text processing in the context of MapReduce, see Lin and Dyer's work "Data-Intensive Text Processing with MapReduce" [19], to which our work owes much.

We have implemented several variants of the Word Count and Inverted Index solutions over the Wikipedia article text dataset as our data-intensive benchmarks covered in chapter 3. Here we describe the Word Count and Inverted Index text processing problems generally and the MapReduce approach to solving them.

2.5.1 Word Count

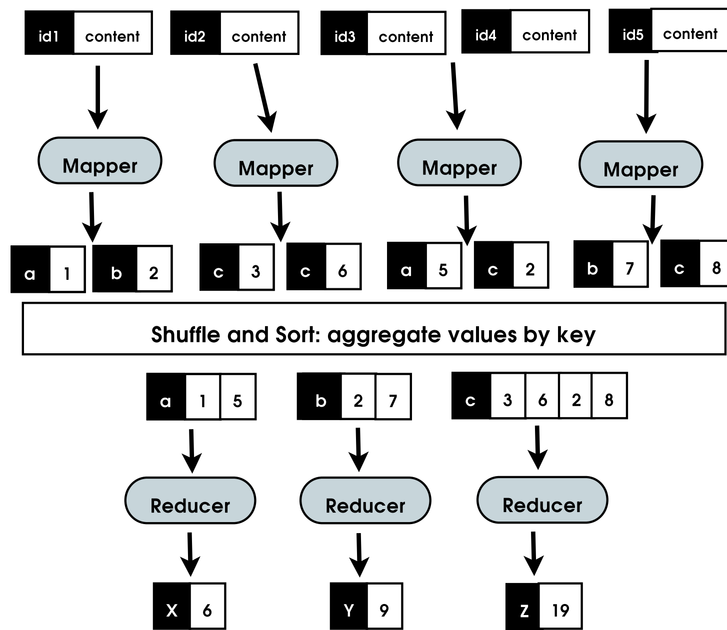
The simplicity of Word Count belies it's fundamental importance in practical computing. It appears everywhere, under term count, event count, and any other name where "counting" is somehow involved. Word Count is the proverbial "Hello Word" program of MapReduce, but is of great practical importance and utility in real world text processing.

```
WordCount
class Job:
    method Map(docid a, doc d):
        forall term t ∈ doc d do
            Emit(term t, count 1)
    method Reduce(term t, counts[c1, c2, ...]):
        forall count c ∈ counts[c1, c2, ...] do
            sum ⟵ sum + c
        Emit(term t, count sum)
```

MapReduce jobs are typically implemented as "classes" in object-oriented programming libraries designed to interface with a MapReduce execution framework, where the user-defined Map and Reduce functions are supplied as member methods of a MapReduce "Job" class. This is reflected in our pseudocode implementation of Word Count.

Word Count takes as input a dictionary mapping document identifiers (docid) to document contents (doc) and returns as output a dictionary mapping each term occurring in any document to the number of times that term occurs over all documents. Notice the document identifier is discarded. A simplified example of how Word Count might execute over a few documents containing only the terms **a**, **b**, and **c** is sown in figure 7.

Figure 7: Word Count Execution [19]



2.5.2 Inverted Index

Inverted Index is only slightly more complex than Word Count. Inverted Index takes as input a dictionary mapping document identifiers (docid) to document contents (doc) and returns as output a dictionary mapping each term to a list of documents in which that term appeared, as well as the number of time it appeared. This solution is of fundamental importance in web-crawling, and forms the basis of the PageRank query input when terms are replaced with web links [19] [25]. A visual example of simple Inverted Index execution is shown in figure 8.

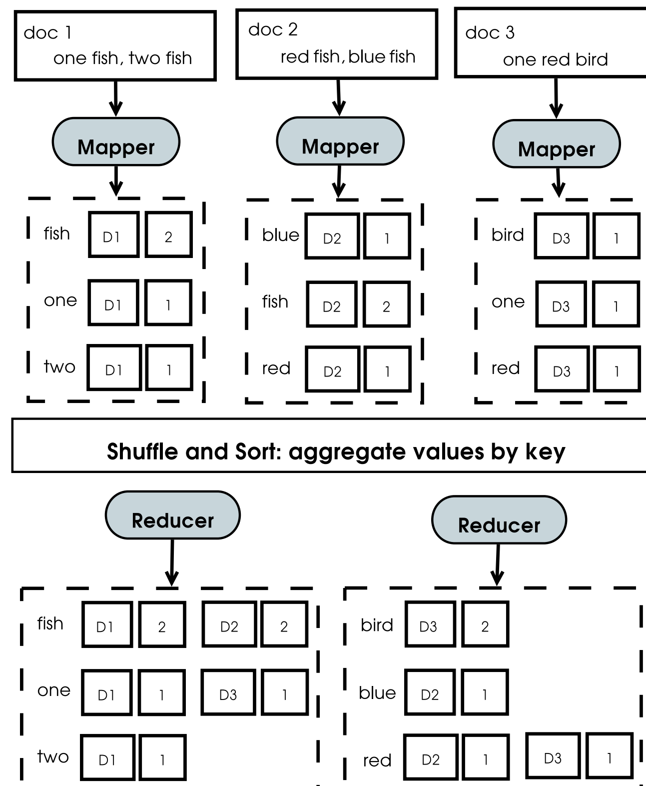
In our work, we focus on index construction, not access, compression, nor the data retrieval problem generally. Storing and compressing indexes for efficient access and retrieval of information is a deep area of research, with much activity. For an excellent treatment of these

related topics, see [20] and [22].

InvertedIndex

```
class Job:
    method Map(docid a, doc d):
        H ⟵ new AssociativeArray
        forall term t ∈ doc d do
            H{t} ⟵ H{t} + 1
        forall term t ∈ H do
            Emit(term t, posting (a, H{t}))
    method Reduce(term t, postings[(a1, h1), (a2, h2), ...]):
        P ⟵ new List
        forall posting(a, f) ∈ postings[(a1, h1), (a2, h2), ...] do
            P.Add((a, f))
        P.Sort()
        Emit(term t, postings P)
```

Figure 8: Inverted Index Execution [19]



2.6 Related Work

Our general distributed text processing work draws heavily on the work "Data Intensive Text Processing With MapReduce" [19] by Lin and Dyer.

"The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines" by Barroso, Clindaras, and Hölzle [5] serves as an excellent in-depth introduction to cluster computer architecture, and we owe much inspiration to their musings on the subject. Their notion of many clustered nodes orchestrated together as a single logical computer is particularly important to this work and works on cluster computing in-general.

FAWN, a "Fast Array of Wimpy Nodes" [3] by Andersen, et. al. is a closely related work to ours. FAWN presents a fast key-value lookup using a cluster of low-power nodes. FAWN's cluster architecture is similar to our own, being composed of "wimpy" nodes with fast flash memory. The FAWN work shows that such a cluster architecture can improve power consumption per key lookup by two orders of magnitude over a conventional implementation. FAWN has inspired further work, aside from our own, on the subject of "wimpy" versus "brawny" CPU architectures applied to throughput intensive problems [18].

At the time of this writing, we are only aware of two prior works related to evaluating ARM64 processor architecture in data-intensive applications. Neither of these prior works have treated low-power single-board computers specifically, but deal with emerging ARM64 hardware for the server market.

"ARM Wrestling with Big Data: A Study of ARM64 and x64 Servers for Data Intensive Workloads" by Kalyanasundaram and Simmhan [16] presents a comparison of the latest AMD ARM64 CPUs [1] with x86 CPUs for data intensive text processing.

"Comparing the Performance and Power Usage of GPU and ARM Clusters for Map-Reduce" by Delpace, et. al, [10] compares a highly integrated ARM64 server architecture against GPUs using MapReduce. Delpace is of note as the only usage of the Disco [11] MapReduce framework on an ARM64 cluster that we are aware of outside our own work.

3 Implementation

In this chapter we introduce our novel MapReduce cluster composed of low-power ARM64 single-board computers. We describe a conventional low-power x86 platform used for comparison. We outline benchmark implementations solving the Word Count and Inverted Link Index problems for the Wikipedia article dataset. We implement external versions and in-memory versions of each solution for our x86 platform, to compare with the intrinsically external execution of the MapReduce framework versions for our cluster. We also implement parallel versus serial versions for our x86 platform, with similar motivation.

3.1 Low-Power SBC Cluster

As a core part of our work, we construct a MapReduce cluster using the Odroid-C2 [24] ARM64 single-board computer, running the AArch64 port of Debian [9] Linux and the Disco [11] MapReduce execution framework.

Figure 9: Cluster Hardware

3.1.1 SBC Nodes

The Odroid-C2 is a low-power "credit-card-sized" computer manufactured by Hardkernel, Inc. Like many SBCs, the Odroid-C2 supports all peripheral I/O expected of a standard desktop computer, including USB, video, networking, and storage. We considered similar ARM64 SBCs for our cluster implementation, such as the Pine64 [28] and RasperryPi 3 [31]. While the RaspberryPi 3 is certainly the most popular single board computer on the market today [35], we found it had poor support for the AArch64 port of Linux at the time we began our

work. Similarly, the Pine64 may be a more cost-effective hardware selection than the Odroid-C2 today, but it had not yet been brought to market at the time. When we began our work, the Odroid-C2 had the highest hardware specifications relative to other card-sized ARM64 SBCs available on the market, when price was accounted for. We only considered SBCs with a unit cost of less than $50 US dollars.

The Odroid-C2 supports eMMC [38], a faster flash based storage mechanism not commonly supported in inexpensive SBCs. The support for eMMC contributed to our decision to use the Odroid-C2 for our cluster. Since we are evaluating our cluster in the space of data-intensive or I/O-bound computing, we elected to include the faster eMMC flash storage in every node. Purchasing eMMC has significant cost, increasing the price of each node by nearly 50%. However, eMMC has considerably better I/O throughput than MicroSD storage, increasing disk read performance by as much as 347% and write by up to 826% (based on benchmarks by Hardkernel). The results of FAWN [3] showing that fast flash storage can compensate for weak processor capability in I/O intensive tasks have inspired us to use faster flash storage in our cluster implementation.

We include both eMMC and MicroSD storage in each cluster node. We install the base operating system for each node to the slower MicroSD storage, reserving the eMMC storage for the MapReduce distributed filesystem and workload processing.

Node Specifications:

- **Operating System:** Debian Linux 8.0, AArch64
- **CPU:** Amlogic S905 quad-core ARMv8 @ 1.5GHz
- **Memory:** 2GB DDR3 SDRAM
- **Storage:** 16GB HS400 eMCC + 16GB UHS-1 MicroSD
- **Network:** Gigabit Ethernet
- **Power Consumption:** 1.65 Watts (idle)

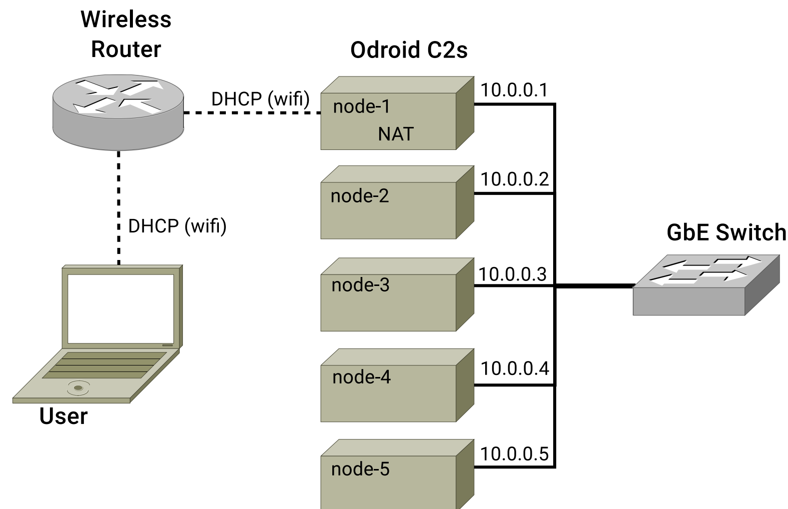
We measured an individual Odroid-C2's ambient idle power consumption to be 1.65 watts. While idle power consumption is no indication of power efficiency under load, it does establish a baseline. In our evaluation in section 4.4, we will consider power consumption during load (benchmark execution), and factor that into our throughput calculations.

3.1.2 Cluster Specifications

Our cluster is composed of five identical SBC nodes built around a simple stacking enclosure and 5-port GbE switch. The master node, which is also running the Disco MapReduce master process, acts as a gateway router between the cluster's internal GbE network and a household wireless network. Therefore, the master node is equipped with a Ralink RT5370 Wireless-N

USB adapter to serve as the external interface. The general arrangement is show in figure 10. Below, we list the specifications for our cluster as if it were a single machine. Obviously, this method of specification does not account for the communication costs incurred by distributing computation across multiple machines, but is simply a roll-up of available resources.

Figure 10: Cluster Network



Theoretical Aggregate Specifications:

- **CPU:** 20 cores (combined 30GHz ARMv8)
- **Memory:** 10GB DDR3
- **"Fast" Storage:** 80GB eMMC (combined 4Gbps read I/O)
- **"Slow" Storage:** 80GB MicroSD
- **Network:** 1000base-T (GbE) Switched
- **Power Consumption:** 8.25 Watts (idle)

Since our cluster is constructed from "credit-card-sized" SBCs, the final assembly is incredibly space-efficient for a cluster in the traditional sense, and can even fit into an office desk drawer. See figure 9.

The actual cluster construction and systems installation is an involved process, wherein we encountered many challenges. For a complete guide detailing how to replicate our cluster, refer to additional resources in appendix 6.2.

3.2 Disco MapReduce Framework

To take our cluster from being merely a network of single-board-computers to being a true "cluster computer" with which we can solve data-intensive text processing problems, we employ the Disco MapReduce execution framework.

Disco [11] is a lightweight MapReduce framework produced by Nokia, Inc. While Disco is primarily interfaced with via the Python [30] programming language, most underlying systems code of the framework is implemented using Erlang [12], a purely functional programming language designed for building fault-tolerant distributed systems. MapReduce, as a technology for fault-tolerant distributed computing rooted in the functional programming paradigm, finds an ideal match with the predominate themes of the Erlang programming language.

However, no knowledge of Erlang programming is required to use Disco. Disco MapReduce solutions are implemented using Python, making them relatively simple and minimal when compared to the more popular Hadoop [37] (which uses Java). Indeed, Disco's motto is "Massive Data, Minimal Code". We chose Disco for its simplicity, as well as the "batteries included" philosophy of the Python programming language.

In our cluster implementation, the Disco MapReduce master process is installed to *node-1* of Figure 10. While production MapReduce clusters typically do not use the master node as a worker node, our implementation allows this. Thus all five nodes, with four CPU cores apiece, are available to the Disco framework, giving us a total of 20 MapReduce workers.

Similar to our cluster hardware construction, the installation and configuration of the Disco framework (or any piece of suffieciently complex distributed systems software) is an involved and lengthy process. Refer to appendix 6.2 for an in-depth guide on how to install and configure Disco for our cluster.

Disco Distributed Filesystem As outlined in section 2.4.2, MapReduce relies on a distributed filesystem to orchestrate data across the cluster. The Disco framework includes the Disco Distributed File System (DDFS) to fulfill this requirement.

Like all MapReduce filesystems, DDFS provides data replication of blocks to multiple nodes in the cluster. In our cluster, we configure DDFS with a replication factor of two (2). This replication factor was chosen for practical reasons; our Wikipedia dataset is roughly 15GB in size, thus a replication factor of two results in our dataset consuming approximately 30GB of the available 80GB of storage across the cluster. This leaves ample storage for intermediate processing and result data, while also providing the execution framework with single-node fault tolerance and the ability to choose between two possible nodes when assigning tasks for data locality.

3.3 Conventional x86 Platform

With the goal in mind of evaluating scale-out versus scale-up architecture in low-power computing, our scale-out architecture is obviously our SBC MapReduce cluster. For a low-power conventional (i.e. non-distributed) scale-up computer, we will turn to a high-end x86 laptop. While the choice to use a laptop to fulfill this role may appear to be a convenience for the experimenter, there are many similar design constraints between laptops and SBCs. Card-sized ARM64 SBCs share much underlying hardware architecture with smartphones and embedded multimedia systems. Therefore, both card-sized SBC and laptop designs are based on architectures low power enough to sustain battery power for prolonged periods of time. Both tend to be optimized for some form of mobile computing. Both are heavily space constrained.

The laptop we employ as our conventional x86 platform is a Lenovo ThinkPad t460s, manufactured recently (in 2017) and possessing a latest generation Intel Skylake CPU. Our x86 platform has flash-based SSD storage, similar to our cluster, but with higher throughput capacity compared to a single SBC node.

Platform Specifications:

- **Operating System:** Debian Linux 9.0, i686
- **CPU:** Intel Skylake i5-6300U quad-core x86-64 @ 2.4GHz
- **Memory:** 12GB DDR4 SDRAM
- **Storage:** 128GB SATA3 SSD
- **Power Consumption:** 9.8 Watts (idle, lid-closed)

Being a conventional computer and not a cluster, our x86 platform will not run a MapReduce execution framework. While it is possible to run a MapReduce framework such as Disco on a stand-alone computer (typically for development purposes), MapReduce frameworks come with considerable runtime overhead. This overhead is only overcome by the increased throughput capacity of running in a clustered context. Furthermore, the MapReduce framework demands solutions be implemented within the restricted MapReduce programming pattern. To maximize the advantages of non-distributed scale-up computing, we do not subject our conventional platform benchmarks to these restrictions. We implement serial in-memory, serial external, and parallel external versions of each solution (c.f. 3.4.3) directly for the platform. The parallel external versions will take advantage of MapReduce as pattern for facilitating parallel programming, but the solutions are otherwise implemented to be as direct as possible.

3.4 Text Processing with Wikipedia and Python

We benchmark our systems by solving two common data-intensive text processing problems for the Wikipedia article dataset. We choose Wikipedia as both a dataset of practical interest, proverbially being "the sum knowledge of mankind", and one possessing convenient properties which support our research goals. The text processing problems we chose to solve for the Wikipedia dataset are Word Counting and Inverted Link Index. We implement our solutions using the Python programming language. The Python programming language was chosen for it's convenient built-in text-processing facilities, and because it is the predominate language of the Disco MapReduce framework and a popular language for data processing in-general [34].

3.4.1 Wikipedia Article Dataset

Wikipedia, aka "The Free Encyclopedia", is a free on-line encyclopedia supported by the Wikimedia Foundation [41]. At the time of this writing, the English language Wikipedia has 5,573,912 articles, or individual documents, in its database [33]. Due to its size and its structured text document format, the Wikipedia article database makes an ideal dataset for exploring data-intensive text processing.

Dumps of the Wikipedia database can be downloaded as compressed XML from the Wikimedia foundation website [39]. The entire content of Wikipedia can be downloaded, or selected subsets. In our benchmarks, we use the English language "multistream" article dump, which includes the text content of all articles, while omitting "talk", "user", or "about" pages, article revision history, and images.

Even with these omissions, the Wikipedia article dataset is quite large, nearly 56GB when uncompressed. The article content includes much extraneous formatting and meta-data. To strip this extraneous data and make articles generally simpler to parse, we use Wikiextractor, a tool written at the Multimedia Laboratory of the University of Pisa, Italy [40]. Wikiextractor allows us to strip all useless formatting and metadata, while preserving article titles and links (titles and links will be required for our Inverted Link Index solution).

Post-processing the Wikipedia article dump through Wikiextractor reduces the size to around 15GB. While 15GB may not seem large, "data-intensive" computing is more about I/O throughput-bound problems than necessarily "big-data" sizes. A mentioned in 3.2, 15GB fits nicely into our cluster's distributed filesystem when using a replication factor of 2. 15GB is relatively large in the context of the small, low-power, platforms we are evaluating.

The extracted Wikipedia article dump is provided as a single, massive, text file. Wikiextractor provides a mechanism to split the dump into evenly sized files while preserving the individual document structure of each article. Using this mechanism, we split our dataset into

3,771 4MB files. This 4MB "chunk" containing a stream of Wikipedia articles forms the unit of the numerous dataset sizes used in our benchmark evaluations in chapter 4.

Dataset Specifications:

- **Number of Articles:** 5,573,921
- **Number of Files:** 3371
- **File Size:** 4MB
- **Total Size:** 15GB

3.4.2 Word Count and Inverted Link Index

The text processing problems we chose to solve over the Wikipedia dataset are Word Count and Inverted Link Index. These problems are described more generally in sections 2.5.1 and 2.5.2. Here we cover a few specific implementation details worth noting in the context of processing the Wikipedia article dataset.

Word Count Word Count is fairly straightforward. Our benchmark solutions return a dictionary mapping any unique word contained in the input dataset with the number of times that word occurs across the entire dataset. Each dictionary key is a word (string), and the value is an integer.

We disregard all case, punctuation, and numbers in the dataset. All words are coerced to lower case. For reference, the Python method in our implementation responsible for this is `clean_words(s)` in `util.py` (c.f. 6.1). Additionally, words shorter than 3 characters in length are ignored. These choices are merely convenient simplifications. Our purpose here is to evaluate systems, not to rehash a semantically robust word count implementation.

Inverted Link Index Our Inverted Link Index benchmarks return a dictionary mapping any article title in the input dataset to a list of all articles which contain a link to that article title. This is a practical application of the more general Inverted Index problem described in section 2.5.2. Wheres general Inverted Index considers any word as key, our implementation only considers links. This was implemented to diversify our benchmarks. Word Count above already outputs a *key, value* for every term in the dataset. In contrast, the *key, value* output of Inverted Link Index is more sparse; a *key, value* is only output when a link is encountered. Furthermore, the value output of Inverted Link Index is a list of article titles (strings), whereas Word Count values are simple integers.

Inverted Link Index can provide a basis for future work in processing the link-graph structure of the article dataset (for instance, a PageRank [25] implementation for Wikipedia).

Similar to Word Count above, we disregard case and punctuation. Furthermore, Wikipedia has various meta-articles which serve as aliases, redirects, and indexes. We do not resolve those to their terminal articles. In a production environment, this sort of processing would likely be handled as a post-processing step, after bulk processing had reduced the data to a more manageable size.

It's worth noting that every page on the officially hosted Wikipedia provides a link on the side-bar which will return to users a list of all inbound links to that page's article (see `https://en.wikipedia.org/wiki/Help:What_links_here`). That is, each Wikipedia page hosts it's own Inverted Link Index. Naturally, this proves quite useful for spot-validation during the development and demonstration of our inverted link index solution.

3.4.3 Implementation Variants

We implement several variants of the Word Count and Inverted Link Index solutions described in section 3.4.2 above, with the goal of exploring the trade-offs between serial, local parallel, and distributed computing models, and benchmarking our low-power SBC cluster against our conventional x86 platform. The first three variants described are the non-distributed implementations for our conventional x86 platform. The last variant is the distributed / MapReduce implementation to be run on our cluster.

These implementation variants will form the basis for our benchmark comparisons and analysis in chapter 4.

In-Memory (conventional x86) Both the Word Count and Inverted Index solutions outlined in section 3.4.2 output a *key, value* dictionary as their final result. Since our conventional x86 platform has 12GB of system memory, we can maintain this dictionary in-memory while processing all data records. No expensive latency costs are incurred by writing intermediate state to disk. However, memory usage grows with the size of the input data. This is analogous to scale-up architecture.

The in-memory variants are single threaded.

Source files (6.1):

- `word_count_inmemory.py`
- `inverted_index_inmemory.py`

External (conventional x86) The MapReduce framework is inherently external in it's execution (i.e. it writes most intermediate work to disk before processing another record), thus we implement external variants in addition to the in-memory variants of each solution for

our x86 platform. When evaluating benchmarks, we will consider in-memory versus external solutions by factoring peak per-record memory consumption into the final throughput measurements (c.f. 4.3). Our external implementation variants make use of the Python `shelve` standard library module, which provides a buffered dictionary implementation which can be flushed to disk between processing input records. Thus, memory consumption is bounded by the individual record size, and is roughly constant over all input dataset sizes.

This variant is single-threaded, and establishes a baseline for the next variant.

Source files (6.1):

- `word_count_external.py`
- `inverted_index_external.py`

Parallel External (conventional x86) This implementation variant extends the previous with local MapReduce-style parallelism. The Python `multiprocessing` standard library module provides a `map` function which functions much like the mapper of a MapReduce execution framework, but in a machine-local context. We use multiple Python `shelve` objects, one per thread, and merge them after processing all records. This is in-essence a limited MapReduce implementation.

This variant is multi-threaded, and uses all 4 cores of our x86 platform.

Source files (6.1):

- `word_count_parallel.py`
- `inverted_index_parallel.py`

MapReduce (ARM64 SBC cluster) Our MapReduce implementations closely follow the generalized forms described in sections 2.5.1 and 2.5.2 for Word Count and Inverted Index, respectively.

The MapReduce variants are inherently parallel, external, and distributed. They may execute on up to 20 cores across our cluster.

Source files (6.1):

- `word_count_mapreduce.py`
- `inverted_index_mapreduce.py`

4 Evaluation and Analysis

In this chapter, we present our benchmark results. We first define the benchmarks and environment. We then measure execution time, memory consumption, and power consumption of our benchmarks. We wrap up by calculating the raw throughput for each benchmark, and throughput adjusted for power and memory consumption.

4.1 Benchmark Setup

Benchmarks are based around the Wikipedia text processing solution variants outlined in section 3.4.3. We refer to these solution variants throughout this chapter in graphs and analysis as *In Memory*, *External*, *Parallel Ext.*, and *MapReduce* variants or implementations. To recap them here:

- *In-Memory:* for x86 platform. Does all processing in-memory.
- *External:* for x86 platform. Saves intermediate data to disk between processing each record, minimizing main memory usage.
- *Parallel Ext.:* for x86 platform. Similar to above, but parallelized using the MapReduce programming pattern.
- *MapReduce:* for low-power SBC cluster. Executes in the parallel-distributed MapReduce framework.

We consider each of the above variants for both our Word Count (3.4.2) and Inverted Link Index (3.4.2) solutions. Thus there are eight total benchmark implementations.

We test each benchmark implementation with dataset sizes of 10, 20, 40, 80, 160, 320, 640, 1280, and 2560 files. Each file is a 4MB stream of Wikipedia articles, as detailed in section 3.4.1. Thus test datasets range in size from 40MB all the way up to 10GB. There are (8×9) 72 total benchmarks to be run. We automated much of the benchmark testing through scripts.

4.2 Execution Time

The first and foremost performance metric identified for any data processing implementation is typically execution time; the time elapsed before arriving at a solution. In our benchmarks, execution times are measured as total data processing execution duration in milliseconds, from the launch of data processing functions until result dictionaries are ready.

The execution time results for Word Count based benchmarks are show in figure 11. The execution time results for Inverted Link Index based benchmarks are show in figure 12. Refer to appendix 7 for full numerical data.

Figure 11: Word Count Execution Time

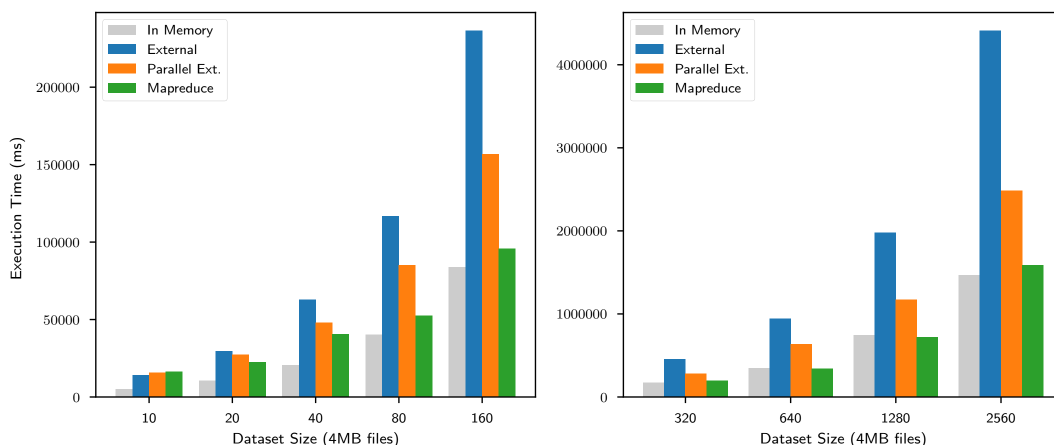
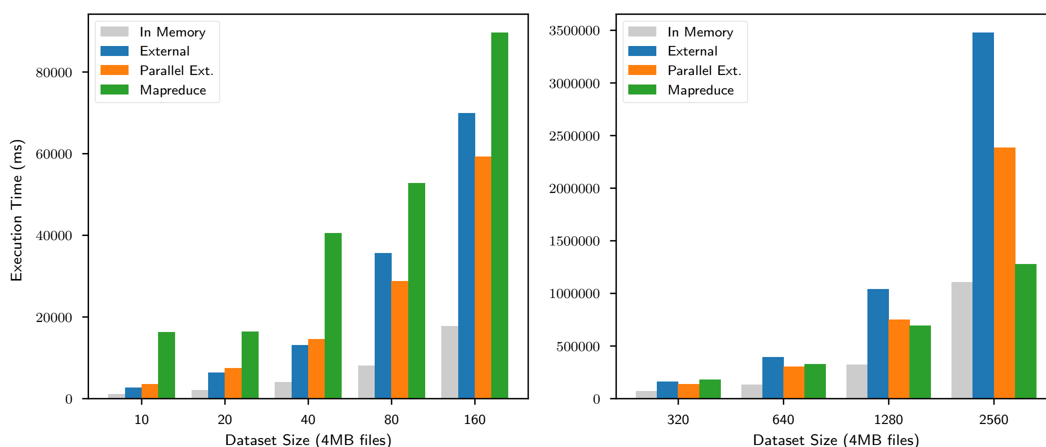


Figure 12: Inverted Link Index Execution Time



As we can see, the conventional x86 platform in-memory implementations are the fastest executing, especially for small dataset sizes. However, as dataset size grows, our SBC MapReduce cluster begins to catch-up. Both our scale-out implementation and best scale-up implementation perform almost identically for the largest dataset size (2560 files or 10GB) under either text processing solution. Under Word Count, our MapReduce cluster is competitive even for smaller datasets, down to 160 files or 640MB.

Inverted Link Index is much sparser in its (*key*, *value*) output for a given dataset input, implying a reduced constant of proportionality in its IO complexity. This becomes apparent when comparing the Inverted Link Index and Word Count execution times. For small inputs, Inverted Link Index is not as dominated by IO throughput as Word Count, and thus the In-Memory implementation of Inverted Link Index out-performs all other benchmarks before

the largest dataset size (2560). In contrast, the added throughput capability of the scale-out / MapReduce implementation performs competitively even for small datasets down to 160 files in the more IO-bound Word Count problem.

This illustrates the importance of IO throughput over CPU capability for data intensive tasks. There is a clear trend showing our scale-out / MapReduce cluster performing commensurately (or possibly better than) the best scale-up conventional solution (In-Memory) as dataset size grows asymptotically.

4.3 Memory Consumption

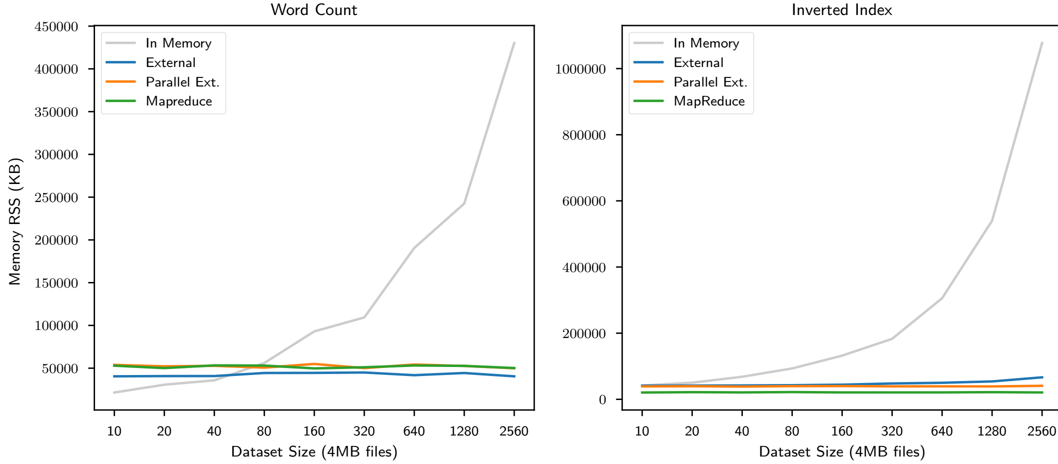
Execution time alone does not give a complete picture of the overall efficiency of an implementation. When allowed "unlimited" space, it is often possible to make trade-offs to gain better execution time. Thus, we measure not just a benchmark's execution time, but also it's memory consumption.

The conceptual "size" of an individual machine is often analogous to its CPU performance and main DRAM capacity. Individual machine costs grow super-linearly with CPU Performance and DRAM capacity. The ability to swap to disk, network, or other external device is a good measure of the decomposability of a solution to run on many "small" and thus less expensive machines. Intuitively, the more efficiently a data processing solution executes when it's peak per-record DRAM occupation is constrained by swapping to external memories, the more amenable to executing on a small machine that solution is and the more effectively it will execute in a distributed / scale-out setting.

The DRAM utilization of our benchmarks is measured as peak per sub-task resident segment size (RSS), as reported by the Linux Kernel in `/proc/self/statm`, or by the `htop` [15] process monitoring utility for MapReduce and Parallel External implementations. What we mean by "per sub-task" is the individual sub-processes which make-up the (potentially) parallelized execution of a data processing job, either in the Mapreduce framework or in our local serial and multi-processing variants of the solutions. That is, an individual map or reduce worker in a MapReduce variant, or a Python `multiprocessing` sub-process in our parallel external variant, or the entire task in our local in-memory and external variants. These tasks have processing functions which are called for each record. In our external implementation variants, where intermediate data is swapped to disk after each record, the peak per-task memory consumption is generally the same as the per-record memory consumption.

We do not measure external solid-state memory utilization, or storage, as the cost of this is fractional and the availability nearly unlimited when compared to the expense of main system memory (DRAM).

Figure 13: Per-Task Memory Usage



As expected, figure 13 confirms that external implementation variants (*External*, *Parallel Ext.*, *MapReduce*) have a constant DRAM consumption in their input size, whereas the In-Memory implementation's DRAM consumption grows linearly with the input size. To account for this large disparity, we will factor these measurements into our adjusted throughput calculations in section 4.5.

4.4 Power Consumption

We factor-in the power consumption of each solution in our analysis of final throughput numbers. In doing so, we can account for the overall efficiency of our x86 platform versus SBC cluter hardware architecture. The rough power consumption measurements for each variant of each benchmark are listed in table 1.

Table 1: Average Power Consumption

	In Memory	External	Parallel Ext.	MapReduce
Word Count	16.2W	16.0W	16.0W	17.25W
Inverted Link Index	17.2W	16.1W	16.2W	20.0W

ARM64 Cluster As our cluster is powered via a multi-port USB power supply, we obtained a USB power meter which will give a voltage (V) and amperage (I) readout when placed in-line between a cluster node and the power supply. From this, the power consumption in watts (P_W) of a single node can be calculated using the simple power formula $P_W = VI$. We periodically sample (by visual inspection of the meter) power draw of an individual node to obtain an average power draw during the execution of each MapReduce algorithm implementation. We

then multiply this average by the number of nodes in the cluster (5), to achieve a rough overall average for the combined cluster's power draw during the execution of each implementation.

x86 Platform Our x86 platform, being a laptop, has an internal battery from which power draw in watts can be read while not connected to AC power. We take advantage of this to measure power consumption of our local in-memory, external, and parallel external implementations, by periodically sampling the battery draw in watts while executing each implementation. From this, we derive a rough average power draw for each local implementation. The battery draw in watts can be read from Linux via the command:

```
cat /sys/class/power_supply/BAT0/power_now
```

Most modern x86 platforms have advanced power governors which scale CPU frequency and floating point performance to optimize for power savings when on battery or at idle-load. To get a fair measurement, this behavior must be disabled by statically assigning a high performance CPU frequency governor in the Linux Kernel via sysfs. This overrides any benefits the x86 platform would gain from battery power conservation mechanisms.The CPU frequency governor is set for high performance via the command(for every CPU in the system `cpu0 cpu1 cpu2 cpu3`):

```
echo performance > /sys/bus/cpu/devices/cpu0/cpufreq/scaling_governor
```

4.5 Throughput Analysis

For any solution to a data-intensive problem, the ultimate metric of performance is often data throughput. That is, how many bits of raw input data can be processed or otherwise rendered useful per unit time. Given our execution time measurements above (4.2) and our known dataset sizes (3.4.1), calculating the raw throughput achieved by each implementation is trivial. We will also present an *adjusted throughput* which factors both power consumption and memory utilization into the raw throughput metric, giving a more holistic hardware-software-scalability perspective to our final analysis.

Raw throughput in Megabits per Second at Dataset Size i is calculated as:

$$Mbps_i = \frac{(i) \times (4MB) \times (8bits)}{(Execution\ Time_i)} \times (1000ms)$$

To factor-in memory consumption, we adjust the above value by dividing in the per-task RSS memory usage at dataset size i (RSS_i), converted to Megabits. Similarly, to consider power consumption, we further divide that value by the average wattage consumed for the implementation, as measured in section 4.4 above.

Figure 14: Word Count Throughput

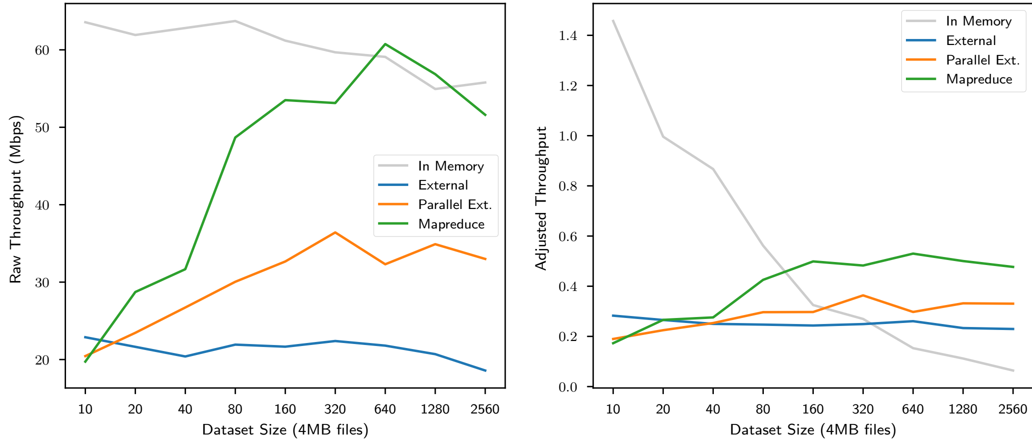
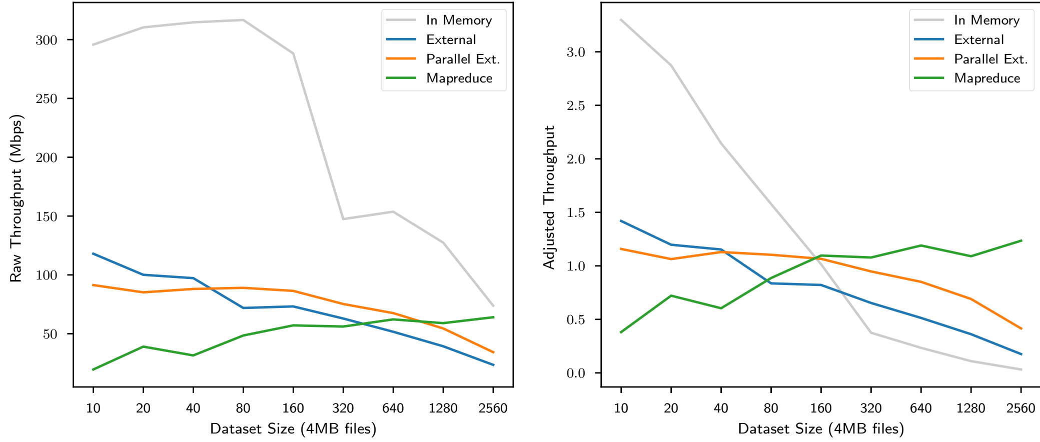


Figure 15: Inverted Link Index Throughput



$$Mbps_i adj = \frac{(Mbps_i)}{(RSS_i)} \times (1000KB) \times (8bits) \times \frac{1}{(Watts_{avg})}$$

Intuitively, this represents the amount of data which can be processed in Megabits for dataset size i, given 1 Megabit of per-task or per-record RSS consumption, 1 Watt of power, and 1 second of time.

The raw and adjusted throughputs for each solution are graphed on the left and right blocks, respectively, of figures 4.5 and 4.5.

Aside from painting a clearer picture for comparative purposes, raw throughput is relatively uninteresting as merely a different presentation of the execution time data in section 4.2 above. Of note again, due to its comparatively sparse (*key, value*) output, the Inverted Link Index solution is not as IO-bound as Word Count. This can be seen in the higher raw throughput achieved by Inverted Index when compared to Word Count.

Of greater interest is the throughput adjusted for power and memory consumption. From figures 4.5 and 4.5, we can see that for larger dataset sizes both our Word Count and Inverted Link Index MapReduce solutions out-perform the conventional x86 platform solutions. Further note this is in-spite of the marginally higher power consumption of our ARM64 cluster per table 1. This demonstrates a core result of our work; clusters of inexpensive low-power machines can out-perform conventional machines with stronger CPUs in data-intensive workloads, from a throughput perspective, when power consumption and scalability of the implementation are considered. Even omitting adjustment for power and memory consumption, our novel low-power MapReduce cluster performs competitively with the conventional x86 platform for the largest dataset sizes, at a fraction of the overall cost.

5 Conclusions and Future Work

We have demonstrated that, in the context of external solutions for data-intensive problems, CPU performance is of little importance compared to combined I/O throughput capacity of a system. A small cluster of low-power and inexpensive machines with comparatively "weak" CPUS can out-perform a relatively expensive single machine with a "strong" CPU, when power and per-record memory consumption are considered.

As a throughput-oriented parallel programming model and execution framework, MapReduce has served as an ideal vehicle for evaluating scale-up versus scale-up computing architectures in the data-intensive context. We have shown our novel MapReduce cluster constructed from low-power ARM64 single-board computers performs comparably to a conventional low-power x86 platform in the space of data-intensive text processing, thus demonstrating the viability of ARM64 single-board computers and similar low-power space-constrained machines for use in data-intensive computing.

In the process of demonstrating this result, we have implemented an accessible model for empirical distributed systems research with MapReduce. Our cluster is inexpensive, low-power, and space efficient enough to be convenient for students and researchers. We have provided practical examples of text processing with MapReduce through our straightforward solutions to the Word Count and Inverted Index problems for the Wikipedia dataset. We hope that our examples and cluster construction inspire further research into computing architectures composed of numerous small and low-power nodes, as well as make MapReduce and general distributed systems research more accessible to those with limited resources.

5.1 Future Work

Having established two solid baseline text processing solutions for the Wikipedia dataset, we would like to develop further solutions exploring more analytical problems. Given the structured document graph format of Wikipedia, it serves as an ideal scaled-down model for exploring the structured document graph of the World Wide Web. For example, our Inverted Link Index solution would make a good first step towards a PageRank [25] implementation for Wikipedia. Word Count and Inverted Link Index are particularly straightforward, both from a complexity analysis standpoint and in their practical implementation. Exploring more complex / non-linear graph processing solutions could provide additional insights to the implications of distributed systems hardware architectures on data-intensive workload, beyond those of IO throughput versus CPU capability, or scale-up versus scale-out approaches.

As low-power computer hardware architecture is constantly evolving, keeping up-to-date

with the latest innovations and their implications for distributed systems is an area of boundless research. We look forward to new evolutions in low-power hardware, such as the embedding of many-core GPUs into ever smaller and more power-efficient packages [23], as well as the ongoing emergence of ARM64 as a mainstream processor architecture for use beyond the embedded and mobile computing space. We believe these innovations will continue to drive movement towards numerous small yet densely distributed nodes in data centers and cluster computing. Studying the application of low-power embedded hardware architecture to traditional data center workloads can give an early look towards the ultra-dense data centers of the future.

6 Additional Materials

6.1 Source Code

The Python source code for our Inverted Index and Word Count solutions for the Wikipedia dataset, and associated scripts and utilities used in all experiments, as well as possible code for future work, is available on the author's personal website at:

`http://dmcdm.org/repo/wikidata`

6.2 Cluster Construction

In-depth instructions on how to replicate our Odroid-C2 cluster running Disco MapReduce are available as a two-part setup guide on the author's personal website.

- **Part 1** covering initial cluster hardware setup and systems software installation:

  `http://dmcdm.org/posts/arm64-cluster.html`

- **Part 2** covering installation of the Disco MapReduce framework to the cluster, and preparation and deployment of the Wikipedia dataset:

  `http://dmcdm.org/posts/arm64-mapreduce.html`

6.3 Slides

Presentation materials which accompany this work are available on the author's personal website at:

`http://dmcdm.org/slides/defense.pdf`

Related to this work, the authors lectured on the topic of MapReduce during the Summer School for Massive Data at TÜBİTAK in Gebze, Turkey. The slides accompanying that lecture are also available:

`http://dmcdm.org/slides/mapreduce.pdf`

7 Data

Here we collate the full data from our benchmarks. The ”Dataset” column of each table is the number of input files, and thus total number processing function calls in our local benchmarks, or the total map jobs to be executed in our MapReduce benchmarks. Each Wikipedia dataset input file is 4MB, and so to obtain total input data size for each Dataset row, multiply the value therein by 4MB. Execution times are reported as total processing job execution time milliseconds, from the launch of dataprocessing tasks until results are ready. Per-Task Memory Usage is measured by a single representative sample of Resident Segment Size (RSS) memory occupation taken during execution of one sub-process in the Parellel-External case, one Map job in the MapReduce case, or the whole process peak memory utilization in the the local In-Memory and External implementations’ columns, as reported by the Linux Kernel for that process in `/proc/self/statm` or by the `htop` process monitoring utility.

Table 2: Execution Time (ms) - Word Count

Dataset	In Memory	External	Parallel Ext.	MapReduce
10	5,034	13,985	15,651	16,218
20	10,336	29,573	27,291	22,282
40	20,376	62,755	47,900	40,418
80	40,172	116,754	85,210	52,586
160	83,674	236,370	156,745	95,689
320	171,577	457,303	281,182	192,731
640	346,589	939,571	633,939	337,125
1,280	745,528	1,978,995	1,173,625	720,337
2,560	1,468,463	4,408,490	2,482,904	1,587,387

Table 3: Per-Task RSS Memory Usage (KB) - Word Count

Dataset	In Memory	External	Parallel Ext.	MapReduce
10	21,544	40,472	53,856	52,920
20	30,700	40,768	52,176	50,096
40	35,780	40,804	52,744	53,272
80	56,032	44,384	50,652	53,064
160	92,996	44,504	54,952	49,744
320	109,324	44,952	50,088	51,068
640	190,572	41,812	54,304	53,160
1,280	242,504	44,372	52,584	52,709
2,560	430,248	40,472	49,920	50,182

Table 4: Execution Time (ms) - Inverted Index

Dataset	In Memory	External	Parallel Ext.	MapReduce
10	1,082	2,712	3,503	16,322
20	2,062	6,395	7,514	16,396
40	4,068	13,162	14,538	40,562
80	8,084	35,607	28,768	52,797
160	17,764	69,950	59,245	89,684
320	69,457	162,748	136,050	182,525
640	133,227	396,375	303,268	329,442
1,280	321,407	1,041,561	751,684	694,177
2,560	1,109,818	3,479,578	2,387,131	1,279,975

Table 5: Per-Task RSS Memory Usage (KB) - Inverted Index

Dataset	In Memory	External	Parallel Ext.	MapReduce
10	41,716	41,312	38,952	20,562
20	50,244	41,532	39,556	21,642
40	68,248	41,916	38,512	20,900
80	93,336	42,712	39,810	21,880
160	132,100	44,276	40,042	20,824
320	182,864	47,856	39,212	20,816
640	305,636	49,996	39,204	20,892
1,280	539,968	54,068	38,972	21,656
2,560	1,077,320	66,488	40,816	20,728

References

[1] Amd opteron a-series processors. https://www.amd.com/en-us/products/server/opteron-a-series. [Online; accessed 30-November-2017].

[2] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference* (1967), ACM, pp. 483–485.

[3] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. Fawn: A fast array of wimpy nodes. *Commun. ACM 54*, 7 (July 2011), 101–109.

[4] Apache cassandra. http://cassandra.apache.org/. [Online; accessed 10-September-2017].

[5] Barroso, L. A., Clidaras, J., and Hölzle, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[6] Borkar, S., and Chien, A. A. The future of microprocessors. *Commun. ACM 54*, 5 (May 2011), 67–77.

[7] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[8] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[9] Debian linux arm64 port. https://wiki.debian.org/Arm64Port. [Online; accessed 14-January-2018].

[10] Delplace, V., Manneback, P., Pinel, F., Varrette, S., and Bouvry, P. Comparing the performance and power usage of gpu and arm clusters for map-reduce. In *2013 International Conference on Cloud and Green Computing* (2013), pp. 199–200.

[11] Disco mapreduce. http://discoproject.org/. [Online; accessed 8-Aug-2017].

[12] Erlang programming language. https://www.erlang.org/. [Online; accessed 10-September-2017].

[13] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. *SIGOPS Oper. Syst. Rev. 37*, 5 (Oct. 2003), 29–43.

[14] Hennessy, J. L., and Patterson, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[15] Htop - an interactive process view for unix. https://hisham.hm/htop/. [Online; accessed 25-February-2018].

[16] Kalyanasundaram, J., and Simmhan, Y. ARM wrestling with big data: A study of ARM64 and x64 servers for data intensive workloads. *CoRR abs/1701.05996* (2017).

[17] Katz, R. H. Tech titans building boom. *IEEE Spectr. 46*, 2 (Feb. 2009), 40–54.

[18] Liang, X., Nguyen, M., and Che, H. Wimpy or brawny cores: A throughput perspective. *J. Parallel Distrib. Comput. 73*, 10 (Oct. 2013), 1351–1361.

[19] Lin, J., and Dyer, C. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[20] Manning, C. D., Raghavan, P., and Schütze, H. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[21] Mongodb. https://www.mongodb.com/. [Online; accessed 10-September-2017].

[22] Navarro, G., and Mäkinen, V. Compressed full-text indexes. *ACM Comput. Surv. 39*, 1 (Apr. 2007).

[23] Nvidia jetson. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules. [Online; accessed 28-Feb-2018].

[24] Odroid c2, hardkernel. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G145457216438&tab_idx=2. [Online; accessed 8-Aug-2017].

[25] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web, 1998.

[26] Patterson, D. A., and Hennessy, J. L. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[27] Picocluster micro desktop datacenter. https://www.picocluster.com/. [Online; accessed 30-November-2017].

[28] Pine64 single board computer. https://www.pine64.org/. [Online; accessed 8-Aug-2017].

[29] Priya, B., Pilli, E. S., and Joshi, R. C. A survey on energy and power consumption models for greener cloud. In *IEEE International Advance Computing Conference (IACC)* (2013), pp. 76–82.

[30] Python programming language. https://www.python.org/. [Online; accessed 10-September-2017].

[31] Raspberry pi. https://www.raspberrypi.org/. [Online; accessed 8-Aug-2017].

[32] Scaleway baremetal cloud servers. https://www.scaleway.com/baremetal-cloud-servers/. [Online; accessed 10-September-2017].

[33] Size of wikipedia. https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia. [Online; accessed 12-September-2017].

[34] The most popular languages for data science. https://dzone.com/articles/which-are-the-popular-languages-for-data-science. [Online; accessed 20-January-2018].

[35] Top 10 single board computers. https://www.eetimes.com/document.asp?doc_id=1332763. [Online; accessed 12-January-2018].

[36] Vitter, J. S. External memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surv. 33*, 2 (June 2001), 209–271.

[37] Welcome to apache hadoop. http://hadoop.apache.org/. [Online; accessed 10-September-2017].

[38] What is emmc. https://www.datalight.com/solutions/technologies/emmc/what-is-emmc. [Online; accessed 14-January-2018].

[39] Wikimedia dumps. https://dumps.wikimedia.org/. [Online; accessed 12-September-2017].

[40] Wikipedia extractor. http://medialab.di.unipi.it/wiki/Wikipedia_Extractor. [Online; accessed 12-September-2017].

[41] Wikipedia:about. https://en.wikipedia.org/wiki/Wikipedia:About. [Online; accessed 12-September-2017].

Curriculum Vitæ

Personal

Daniel J. McDermott

Graduate Student (Computer Science)

Eastern Washington University

Computing and Engineering Building

Cheney, WA, 99004-2493, USA

Web:	`http://dmcdm.org/`
Email:	`dmcdmott@gmail.com`
Tel:	+1 (509) 434 - 6853

Education

Masters of Science, Computer Science, March 2018 (in progress), GPA: 3.88

Eastern Washington University, Cheney, WA 99004

Thesis: Evaluating a Cluster of Low-Power ARM64 Singe-Board Computers with MapReduce

Supervised by: Dr. Bojian Xu, Dr. Stuart Steiner

Bachelor of Science, Computer Science, December 2011, GPA: 3.74, Magna cum Laude

Eastern Washington University, Cheney, WA 99004

Honors and Awards

2011–2013	Graduate Service Appointment in Computer Science
2011	Graudated Magna cum Laude
2009–2011	Deans List of Distinguished Students (all quarters)

Experience

2013 – present	Software Engineer	F5 Networks Liberty Lake, WA

Worked on numerous projects from small-scale embedded ARM SoCs to extremely large, highly integrated, x86 platforms, doing firmware development of UEFI/BIOS, embedded linux, and baseboard management controllers. Currently involved in F5's core datapath writing highly optimized close-to-the-metal code for accelerated network drivers. Promoted twice in 4 years. In 2016, won prestigious "5Star" trip awarded to top 2% of non-sales performers annually.

2011 – 2013	Graduate Assistant	Eastern Washington University Cheney, WA

Involved in a wide variety of critical systems administration activities for the Computer Science Department, including managing and designing compute clusters, scripting, service deployment, desktop support, configuring web, mail, and database servers, hardware maintenance, virtualization, and networking. Responsible for over 160 machines and dozens of servers across seven Computer Science labs. Responsible for training first year assistant and providing guidance to undergraduate interns.

Talks and Presentations

2013 "MapReduce Framework for Distributed Computation". Summer School on Massive Data Management, TÜBİTAK.

2013 "Introduction to the MapReduce Model for Distributed Computation", Graduate Colloquium talk, Eastern Washington University.

2013 "Algorithms for Establishing Trust in the Cloud", guest lecture to CSCD539 Probabilistic Methods in Computer Science, Eastern Washington University.

2012 "Optimization and Instrumentation", presented at Eastern Washington University to visiting computer science graduate students from Iwate Prefectural University, Japan.

Service

2013, 2017	Regional Site Technical Director, ACM PacNW Regional Programming Contest
2013–present	Volunteer Judge, Local ACM Programming Competitions
2013	Lecturer, Summer School on Massive Data Management at TÜBİTAK, Gebze, Turkey
2013	Collegiate Cyber Defense Competition Coach
2012–2013	President, EWU ACM Linux Users Special Interest Group
2012, 2013	Head Referee, First Lego Robotics League, Regional Championship
2011–2013	IT Staff, ACM PacNW Regional Programming Contest
2012	Mentor, Rogers High School Cyberpatriot Program
2012	Mentor and Judge, Washington National Guard Cyberwarfare Exercises
2012	Tutor for Visiting Graduate CSE Students from Iwate Prefectural University, Japan
2011	CS Student Representative for Graduate School Accreditation Review