Distributed Computing

Auston O Schendzielos

December 15, 2014

Contents

List of Figures

Abstract

For years, the computing industry has expected continued increases in CPU performance as predicted by Gordon Moore. Unfortunately, Moore's law is likely to end, but that does not mean the end for computing performance. Several alternative software options have presented themselves in the form of parallel computing. This paper focuses on one of those options: Distributed Computing/Systems. One such implementation of distributed systems is referred to as "MapReduce" and while it is gaining popularity, we will show that there are better, previously existing implementations that perform better.

1 Introduction

Distributed computing is defined as any computing system that involves multiple independent computers working together on one process. There's been a large push in the last 20 years for distributed computing for a number of reasons. Firstly, there's a strong need for parallelism, which will be covered in the following sections. Additionally, many programming models could use the additional work to speed up lengthy computations. This paper will cover the technical details of how programmers use software to exploit parallelism and achieve their goals.

2 The Problem: Moore's Law

Since 1965, the number of transistors integrated on a processor has nearly doubled every two years (approx. 1.995x). The trend has been sustained by designing architectures that could accommodate an increase in transistors, and subsequently, and increase in clock speeds, or switching speed of transistors. While the number of transistors on a chip has continued to increase, clock speed growth has stagnated at speeds generally in the 3 GHz range [6]. This stagnation, was observed around 2003 for Intel-developed processors, and would lead to utilization of multiple cores (referred to as parallelism) [6]. There are three factors that contribute to this decline: the Instruction-level parallelism wall, memory wall, and power wall.

One limitation, the instruction-level parallelism wall, is the result of parallel resources at the hardware level that have already been consumed. In the past, " . . . computer architects (and compiler writers) worked diligently to find ways to automatically extract parallelism . . . from their code" [6]. Some examples of instruction-level parallelism include: pipelining, and hyper threading. Unfortunately, these techniques no longer yield nearly as much parallelism, and " . . . most computer architects believe that these techniques have reached their limit" [6].

Another limitation is the memory wall. The memory wall results from off-chip memory not increasing as fast as processing speeds [6]. The reasons for this include: "slow and power hungry off-chip communication, and difficulty incorporating additional pins in the integration package" [6].

Lastly, the most significant barrier to the increase in computing power is the power wall. According to McCool, "The power wall results because power consumption (and heat generation) increases non-linearly as the clock rate increases, and typical heat dissipation methods (air cooling etc.) can no longer deal with the power generation". More specifically, the amount of power, when not dissipated efficiently, causes false bit-flips. "Whenever the thermal noise causes crossing of the value of the actual logic threshold voltage . . . a false bit flip occurs" [4]. In simpler terms, whenever there is too much heat traveling across a chip, it has the potential to cause the wrong transistors to switch their bit (binary) values. This leads to indeterminate, and incorrect processing. According to Kish, the only way to get around this effect would be either to give up increasing the integration density, that is itself Moores law, or to give up increasing the clock frequency. Clearly, neither of these two alternatives attract many followers, and the work-around is to make parallel microprocessors, and/or distributed systems.

3 Parallelism

The foundation of every distributed system is an implementation of parallelism, and the core principle of parallelism is using multiple compute-units to perform a task that could be done on one. This can be achieved through both software and hardware. Generally, increasing compute-units also correlates to an increase in compute time, but that is not always the case. There are several factors that prevent parallelism from being efficient. That will be discussed later, for now, it is important to understand the types of parallelism that can occur, speedup, and concurrency.

3.1 Types of Parallelism

There are two hardware mechanisms that are utilized by software to achieve parallelism.

The first of which is thread parallelism. Thread parallelism is an implementation that allows each worker (core or node) to have its own flow-of-control i.e. each worker operates its own operations on the data [6]. Facilitating thread parallelism requires thread parallelism at the lowest level, and software threads at a higher level. Software threads are simply virtual hardware threads. An op- erating system typically enables

many more software threads to exist than there are actual hardware threads by mapping software threads to hardware threads as necessary [6]. The ability to write multiple threads at once is crucial for speeding up the performance of par- allel programs.

Vector parallelism on the other hand is an implementation that uses the same flow-of-control for each worker by performing single operations replicated over collections of data [6]. The paradigm can be broken down into two parts: vector instructions, and vector registers. Vector registers holds a small array of elements where each vector instruction is performed on the registers one at a time [6].

A way to understand these two concepts is to imagine Bart Simpson when he is writing lines on the chalkboard during detention. There are two ways for him to write the lines: the first is to write one word at a time for each line. The second way is to write each line individually. These scenarios represent vector parallelism and thread parallelism respectively with the exception that in a parallel environment there would be multiple workers completing tasks at the same time.

3.2 Benefits of Parallelism

The most important metric used in parallel computing is speedup. Speedup is calculated by taking the time a program takes to run serially divided by the time it takes to run in parallel. A similar measure, efficiency, is measured by taking the speedup divided by the number of workers. There are two limits that determine the amount of parallelism that is possible given a program and a set of workers.

The first is Amdahl's law. Amdahl's law is that " . . . the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude" [6]. To put it simply, a computational program will always be limited by the serial work in the program. Serial work is work that cannot be parallelized. This usually comes in the form of dependency. To illustrate Amdahl's law, imagine if you had a program that was one percent serial, and 99 percent parallel and it took 100 seconds to run with one worker. If you were to add another worker, the program would take approximately 46 seconds (one second for the serial work, and 45 for the parallel). Each additional worker would continue to reduce the time it takes to complete the parallel work until the only overhead invovled in computing the program is one second, or the time it takes to compute the serial work. Amdahl's law leads to significantly less return for each additional worker which discourages increasing the number of workers in a system.

Fortunately, Gustafson-Barsis' law restores the incentive to increase the workers in a system because it takes problem size into account. They state that " . . . speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size" [6]. Essentially, each additional unit adds the same amount of parallel work as the first original worker in the same amount of time.

Both of these limits are important to the performance of a parallel system. Every parallel system aims to optimize these metrics, and without these benefits, the end of Moores Law would be closer to a death sentence to the computing industry.

4 Differences Between Parallel and Distributed Systems

Parallelism needs to be incorporated through software, and there are two software systems that incorporate it: Parallel systems, and distributed systems. Distributed systems introduce the idea of a "workstation" which means that each node can run processes on their own. This is in contrast to Parallel systems where each node dedicates itself to the current process. Because of this difference there are several defining characteristics about each system: resource management, functionality, and interprocess communication.

Acceptability determines how resources are acquired and used in either system. For the parallel systems, utilization drives the acquisition of resources [8]. When a program is running, a parallel system will not allow any idle resources to be used for other applications [8]. When a distributed system has idle resouces, it allows them to be used for other purposes [8]. Both systems want to . . . [maintain] the parallelism and the structure of the underlying system [to be] hidden from the users [8]. This is sometimes referred to as a serial illusion [6].

Functionally, distributed systems have " . . . functional overhead on nodes which have been dedicated to parallel applications" [8]. This is due to the fact that distributed nodes try to provide full functioning workstations while maintaining parallel resources. On the other hand, compute-nodes in a parallel system " . . . only need the functionality required to run parallel applications" [8].

Lastly, differences in interprocess communications gives parallel systems an advantage over distributed systems. This is due in large part to memory management differences in each system. "In the parallel system, each node has a shared memory system which allows for quicker, more efficient communication" [8]. Distributed systems have completely separate memory from each other and must rely on slower networked communications [8].

5 MapReduce

MapReduce is a very popular paradigm because of its simplicity. It has only two functions that the user needs to worry about: the Map, and Reduce functions. Both functions are completely customizable, and both functions are fundamental in providing parallelism to the distributed cluster. In the following sections, the details of how MapReduce utilizes a distributed system will be described in detail.

5.1 Programming Model

The overall computation of a MapReduce program involves a set of key/value pairs, and produces a set of output key/value pairs. From there, the computation is broken into several steps. The first part of the execu- tion is Map which takes the original input and produces a set of intermediate key/value pairs i.e. intermediate values that have the same key are grouped together [3]. The intermediate values are then passed to the reduce function which merges values to find a possible smaller set [3]. The reduce functions output is the final output of the computation.

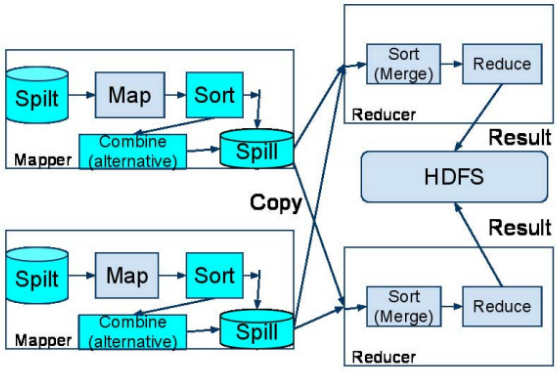

Figure 1: MapReduce execution. Input is broken into splits, and the user's map function is applied. The result, called intermediate data is then sorted and put through the user's reduce function. The final result is stored in the HDFS

Both the Map and Reduce functions are parallel patterns that invoke vector parallelism. This is because both map and reduce are vector functions that are given to the workers, giving each worker the same flow of control.

5.2 A Basic Program

One very simple program that is commonly used is a word count program. The program essentially takes a text document and counts the occurrence of each word. MapReduce accomplishes this task as such: the map function emits each word plus an associated count of occurrences. The reduce function sums together all counts emitted for a particular word [3]. The program is widely used as a benchmark to test different implementations of MapReduce (Mantha) and will be used to demonstrate a Hadoop implementation.

5.3 Execution

In general, all implementations of the MapReduce work similarly to the word-count program in that each part is broken down by the Map function and then further reduces the input into the same or smaller files. A more detailed execution algorithm goes like this:

1. The original key/value pairs are split into M pieces of user-controlled sizes between 16/64MB [3]. M does not have to correspond to the number of nodes in the system. Then several copies of the program are distributed onto a cluster of machines [3].

2. One copy is given the master designation. There are M tasks, and R reduction tasks that the master assigns these tasks to idle machines [3].

3. The workers that are assigned to the map tasks take the contents of the split and runs them through the user-defined function. The intermediate key/value pairs are buffered in memory [3].

4. The buffered pairs are then written to a local disk, partitioned into R regions, and then the location of the stored data is given to reduce workers through the master node [3].

5. Once a reduce worker knows a location of intermediate pairs, it will find that partition and extract all its data. It then sorts the values by keys and all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task [3].

6. The reduce worker takes the intermediate data it collected and iterates each pair through the users reduce function. The output is appended to the final output file [3].

7. When all tasks are completed, the master notifies the user [3].

5.4 Locality

Usually, each implementation of MapReduce comes with a Global File System (GFS). This compensates for the scarcity of network bandwidth in the distributed

environment. When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth [3].

5.5 Fault Tolerance

One of the large benefits to using the MapReduce framework is its ability to handle disk failures gracefully. In a MR system, if a unit of work fails, then the MR scheduler can automatically restart the task on an alternate node [7]. In the MapReduce framework, the master pings every worker periodically [3]. Once a worker is deemed corrupt, the master notifies the other workers, the master re-executes the work on that machine on another node, and the workers make sure not to accept output from that node [3].

5.6 Architecture Models

Another benefit of the MapReduce framework is its ability to utilize multiple resources from around the world. The question then becomes a matter of how to architect the MapReduce to optimize its performance.There are three architectures to run Mapreduce:

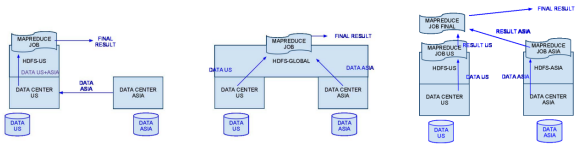

Figure 2: An illustration of the Architecture models described in section 5.6

Local MapReduce (LMR) All data is located in a centralized cluster and perform MapReduce on said cluster. This was how MapReduce was intended to be run [1]. "However, large-scale wide-area data transfer costs can be high, so it may not be ideal to move all the source data to one location . . ." [1].

Global MapReduce (GMR) In this architecture, the data is pushed to a literal global file system that is shared amongst all nodes–even if they are thousands of miles apart. Once all the data is in the file system, a MapReduce is run over all the nodes. This is generally inefficient due to intermediate data having to potentially transferred thousands of miles to another node [1].

Distributed MapReduce (DMR) The data in this architecture stays with it's original cluster. An initial MapReduce is run on the cluster's local information, and then a global MapReduce is performed on all the clusters in the system. This architecture solves the problems of both LMR and GMR by combining their strengths (i.e. LMR is good locally, GMR is needed for wide-spread computation) [1].

Choosing an architecture depends entirely on the locality of resources, and the amount of workload needed for the MapReduce program. If the resources are primarily in one cluster it would make sense to run in LMR regardless of the problem size. If the resources are global but the program is simple, GMR works best, and if the program is more complicated, a DMR is better suited [1].

5.7 Software Implementations of MapReduce

The most popular implementation of MapReduce in software is Apache's Hadoop. It runs a no-frills version of MapReduce that is insignificantly different than the generic form of MapReduce. The biggest difference is the Hadoop Distributed File System which takes the place of the GFS in section 6. It does come with its pitfalls: "The main limitation of Hadoop MapReduce is that it forces applications into a very rigid model" [5]. According to Mantha et. Al. this means that Hadoop is "limited in terms of extensibility". Mantha et. Al. propose a different MapReduce software referred to as Pilot-MapReduce.

6 Demonstration

For my demonstration, I built a Raspberry Pi cluster and ran Hadoop on it. The project did not commplete in time for the writing of this paper, but I will give an explanation of the process and materials needed to replicate my project.

6.1 Materials

In order to complete this project, you will need the following items:

Raspberry Pi For my project, I used four Raspberry Pi computers. Three are Model B+'s and the other is a Model B. Both models have 512 MB of SDRAM, and their processors compute at 700MHz. The Pi's use SanDisk cards for their hard drives, and the B+'s require a micro-SD card, but the B requires a regular sized SD-card. The B+'s also have several additional attachments, but none are relevant to this project. The SD cards must have 8 GB or more of storage.

Ethernet Switch You will need a powered ethernet switch that has at least 4 ports. Ideally, you would get a 6 port switch so you can use 4 for

the cluster itself, another port for your own personal computer, and the last port for an internet connection.

USB Hub The USB hub must have 4 or more ports. The Pi's are powered by a micro-USB port on the mother-board, and so a hub with 4 male-USB to male-micro-USB cords are necessary for powering the cluster. The hub MUST be powered, otherwise the machines will not draw enough wattage.

Peripheral Devices You will need a keyboard and monitor for configuring the Raspberry Pi's at least for the initial start-up. After the Pi's have been SSH enabled, you can ssh from a personal computer for a better interface.

6.2 Procedure

1. The first part of setting up the cluster is formatting the SD cards. Using your own personal machine, format the SD cards using SanDisk's formatter. If you use your machine's native formatter you will corupt the disk.

2. After the disk has been formatted, you will need to download and install an operating system onto the disk. You can go to raspberrypi.org for downloads. I chose to use the Raspbian operating system for my project.

3. Once the operating system is on the card, eject it from your personal computer and insert it into the card slot on the pi. Then power the machine and hook up a keyboard and monitor to the pi. The initial startup brings you to a configuration screen where you should enable SSH. Upon completion, you should remote into the machine using SSH from your own personal machine.

4. Now you will want to update/upgrade the system, and prepare the system with the software needed to run Hadoop. Hadoop runs exclusively on Java, so you will need to download the newest version of Java.

5. Create a static IP address so that the other nodes can communicate with the current node.

6. Install Hadoop by going to Apache's website. They will want you to download Hadoop from a mirror site, and personally, the mirror sites did not work for me, so I downloaded from their archive and used the SCP command to transfer the file from my machine to the pi.

7. After Hadoop has been installed, you will now want to run the pi in pseudo-distributed mode. Configure the environment variables so that this occurs using the instructions on Hadoop's website.

8. Now that the first node is setup, repeat steps 1 through 5 on another node, but this time you are going to configure the node to be a slave node.

9. Once that node has been completely configured, make an image of it's hard drive and install it onto the remaining SD cards.

10. After each node has Hadoop running on it's system, configure each node so it can communicate with the others in the system. You will want to setup automatic login for ssh so that you do not have to do this manually when you run jobs.

11. Now you are ready for execution. Give the namenode (first node you configured) a MapReduce job, and it will distribute it to the clusters. I used a wordcount program and downloaded books from Project Gutenberg.

7 Future Trends

In the 1970s, the database community faced a choice in regards to aprogramming model. The choice was between stating what you want and presenting an algorithm for data access. They would eventually decide on going the former route, and the paradigm referred to as relational programming would prevail for 30 years in the database community. The reasons the database research community chose relational programming models are similar to why many companies and researchers are moving to parallel-relational databases instead of Hadoops MapReduce.

7.1 Programming Model

MapReduce is analogous to the Codasyl of the time because it uses an algorithm to extract data. Both the Map and Reduce functions are user defined, which means every time a program is run, the user must specify these operations. At the time of the database communitys decision, Codasyl was considered the assembly language of DBMS access. Unsurprisingly, many MapReduce users share code fragments to do common tasks, such as joining data sets" [7]. Some MapReduce community projects attempt to implement high-level languages on top of the current interface to solve this issue. This begs the question; why not just use a parallel-relational database in the first place?

Additionally, a number of distributed computing community members, including it's creator, Google, have removed MapReduce from their software entirely. Cloudera's analytical database, Impala, is created using the Hadoop File System (HDFS) [9], and claims

to "leverage the flexibility and scalability strengths of Hadoop" [2]. Interestingly, the MapReduce layer is not included in this software which means Impala's creators do not believe that MapReduce is a strength of Hadoop [9]. That is discouraging for Hadoop, but should users be weary of MapReduce itself? "Google announced that mapReduce is yesterday's news and they have moved on . . . because they wanted an interactive storage system and MapReduce was batch-only." [9] Hence, the driving application behind MapReduce moved to a better platform a while ago. Now Google is reporting that they see little-to-no future need for MapReduce [9]. Needless to say, MapReduce may be on it's way out, but is Hadoop?

7.2 File Systems

HDFS remains the only usable part of Hadoop, but parallel DBMS's remain the better option. Because of Hadoop's prerogative to be simple, it left the user with a lot of work to do in order to efficiently manage the system. In a parallel DBMS, "B-tree indexes [are used] to accelerate access to data," but in MapReduce systems, "the programmer must implement any indexes that they may desire to speed up access to the data inside of their application" [7].Correspondingly, data distribution in a Hadoop system again must be defined by the user while the parallel DBMS's do this automatically. Even Hadoop's approach to data access excludes itself from becoming useful in the future. ". . . from the point of view of a parallel SQL DBMS, HDFS is a 'fate worse than death'. A DBMS always wants to send the query to the data and never the other way around" [9]. Since HDFS is essentially the other way around, it is unlikely that an SQL interface will be practical.

7.3 Performance

In multiple studies on the performance of the two systems, both had MapReduce and parallel DBMS's had unique advantages. The studies used a common grep task to compare performance on varying numbers of compute nodes. In general, Hadoop systems took less time to set up than did the parallel DBMS's. But the performance of the task at hand was significantly faster using the parallel DBMS. One of these studies gave a glimpse of hope to MapReduce users by saying: "[MapReduce]-style systems excel at complex analytics and [extract, transform, and load] tasks" [10]. This at least leaves room for Hadoop and MapReduce users to continue on with their path, but it severely limits the advertised utility of Hadoop.

8 Conclusion

Ultimately, distributed systems remains a great path for solving the end of Moore's law via parallelism. MapReduce may not be the best paradigm to use, but it is still useful for teaching purposes because of it's simplicity.

9 Appendix

Throughout the procession of this course and this assignment, I have utilized several skills and incorporated knowledge gained from other classes I've taken during my time here at Saint John's University. The bulk of my knowledge and skills came from Parallel Computing, Networks, and the internships I've had.

Parallel Computing equipped me with the knowledge I needed to do this research, and to do this project. I actually referrenced the textbook used for that class for this paper. I also used some of the command-line skills from that class for the project.

The Networks course that I took this semester helped me understand some of the networking principles involved in making a cluster. This actually worked both ways as I would learn several new skills in my project and be able to apply them to my Network assignments.

Lastly, the internships I had gave me skills I used to build the cluster. Most of my command-line skills came from my first internship.

References

[1] Michael Cardosa, Chenyu Wang, Anshuman Nangia, Abhishek Chandra, and Jon Weissman. Exploring mapreduce efficiency with highly-distributed data. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 27–34, New York, NY, USA, 2011. ACM.

[2] Cloudera.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[4] Laszlo B Kish. End of moore's law: thermal (noise)death of integration in micro and nano electronics. *Physics Letters A*, 305(34):144 – 149, 2002.

[5] Pradeep Kumar Mantha, Andre Luckow, and Shantenu Jha. Pilot-mapreduce: An extensible and flexible mapreduce implementation for distributed data. In *Proceedings of Third International Workshop on MapReduce and Its Applica-*

tions Date, MapReduce '12, pages 17–24, New York, NY, USA, 2012. ACM.

[6] James Reinders Micahel McCool, Arch D. Robison. *Structured Parallel Programming.*

[7] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[8] Arthur B. Maccabe Rolf Riesen, Ron Brightwell. Differences between distributed and parallel systems, 1998.

[9] Michael Stonebraker. Hadoop at a crossroads?

[10] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: Friends or foes? *Commun. ACM*, 53(1):64–71, January 2010.