HYBRID JOB SCHEDULING FOR IMPROVED SHARED CLUSTER UTILIZATION

A Thesis

by

Uğur Koçak

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
January 2014

Copyright © 2014 by Uğur Koçak

HYBRID JOB SCHEDULING FOR IMPROVED SHARED CLUSTER UTILIZATION

Approved by:

Assistant Professor Ismail Ari, Advisor
Department of Computer Science
Özyeğin University

Assistant Professor Barış Aktemur
Department of Computer Science
Özyeğin University

Associate Professor Güray Erkol
Department of Natural and Mathematical Sciences
Özyeğin University

Date Approved: 10 January 2014

To My Father

ABSTRACT

In this thesis, We investigate the models and issues as well as performance benefits of hybrid job scheduling over shared physical clusters. Clustering technologies that are compared include MPI, Hadoop-MapReduce and NoSQL systems. Our proposed scheduling model is above the operating system and cluster-middleware level job schedulers and operating system level schedulers and it is complementary to them. First, we demonstrate that we can schedule MPI, Hadoop and NoSQL cluster-level jobs together in a controlled-fashion over the same physical cluster. Second, we find that it is better to schedule cluster jobs with different job characteristics together (CPU vs. I/O intensive) rather than two or more CPU intensive jobs. Third, we describe the design of a greedy sort-merge scheduler that uses the learning outcome of this principle. Up to 37% savings in total job completion times are demonstrated for I/O and CPU-intensive pairs of jobs, but up to 50% savings (or 2x speedup) is theoretically possible. These savings would also be proportional to the cluster utilization improvements, if there are jobs waiting in the queue. At the end of the thesis, we also discuss potential power-energy savings from hybrid job scheduling.

ÖZETÇE

Bu tezde, bilgisayar küme yapıları üzerinde hibrid iş yüklerinin birlikte işlenmesiyle ilgili modeller, sorunlar ve performans kazanımları incelenmektedir. Desteklenen kümeleme teknolojileri arasında MPI, Hadoop-MapReduce ve NoSQL sistemleri bulunmaktadır. Önerilen programlayıcı modeli işletim sistemi seviyesindeki arakatman yazılımların üzerinde ve onları destekleyici niteliktedir. Tezde ilk olarak, MPI, Hadoop ve NoSQL işlerini bir arada programlayabildiğimizi göstermekteyiz. İkinci olarak, farklı özelliklere sahip (CPU vs. Girdi/Çıktı yoğunluklu) işlerin, aynı özelliklere sahip işlere göre (2 adet veya daha fazla CPU yoğunluklu) beraber daha iyi programlanabildiği bulgusu paylaşılmaktadır. Son olarak, bu bulgunun ışığında yeni bir greedy sort-merge programlayıcısı tasarımı anlatılmaktadır. İş tamamlama sürelerinde %37 zamansal kazanım gösterilmektedir, ancak %50 kazanımlar da (2x hızlanma) teorik olarak mümkündür. Bu zamansal kazanımlar kuyrukta yeterince yük olduğu takdirde kümenin kullanım kapasitesini de arttırıcı nitelikte olacaktır. Tezin sonunda, hibrid iş programlama ile sağlanabilecek potansiyel güç-enerji kazanımları da tartışılmaktadır.

ACKNOWLEDGEMENTS

First, I would like to express my gratitude to my advisor Dr. Ismail Ari for working with me in Cloud Computing Research Group in at Ozyegin University.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Tankut Barış Aktemur and Dr. Güray Erkol for their encouragement, insightful comments and questions. Next, I would like to thank to all of my instructors in Ozyegin University for their lectures and special attentions during masters.

I would also like to thank my friends in Cloud Computing Research Group, and all of graduate students in Ozyegin University. Especially, I would like to share special thanks with Erdi Ölmezoğulları, Nitel Muhtaroğlu and Melih Koca. I also would like to thank great people that I have good memories with Volkan Yazıcı, Yaprak Ayazoğlu, Buse Yılmaz and Athar Khodabakhsh.

Last of all to you my precious family, who always support and encourage me to reach the best I can. I dedicated this thesis to my father, who passed away before, with all loving memories.

I would also like to send special thanks to our sponsors. My masters research has been partially sponsored by European Union FP7 Marie Curie Program BI4MASSES Grant, Avea Labs, TUBITAK 3501 Project 109E194 and IBM Shared University Research program.

TABLE OF CONTENTS

DEDICATION . . . . **iii**

ABSTRACT . . . . **iv**

ÖZETÇE . . . . **v**

ACKNOWLEDGEMENTS . . . . **vi**

LIST OF TABLES . . . . **ix**

LIST OF FIGURES . . . . **x**

I INTRODUCTION **1**

1.1 Motivation 1

1.2 Definition and Benefits of Hybrid Scheduling 2

II BACKGROUND AND RELATED WORK **4**

2.1 Background 4

2.1.1 Hadoop 5

2.1.2 Message Passing Interface (MPI) 7

2.1.3 NoSQL 7

2.2 Related Work 8

III A DISCUSSION ON VIRTUALIZED CLUSTERS **11**

IV PERFORMANCE ANALYSIS AND RESULTS **14**

4.1 Cluster Environment and Profiling Utilities 14

4.2 Workloads 14

4.3 CPU-intensive Jobs and Cluster-Level Job Prioritization 16

4.4 Hybrid Scheduling of CPU and IO-intensive Jobs 18

4.5 Generalized Hybrid Scheduler 18

V HYBRID JOB SCHEDULING FOR ENERGY EFFICIENCY . **22**

5.1 Measuring Energy Consumption 22

5.2 Workloads 23

VI CONCLUSION **27**

REFERENCES **28**

VITA **30**

LIST OF TABLES

1 Resource usage profiles of benchmark cluster workloads. 15

2 Results of co-scheduling MPI and Cassandra jobs. 17

3 Results of co-scheduling of MPI and Hadoop jobs. 18

4 Power consumption and thermal output by running jobs on 2 nodes. . 25

LIST OF FIGURES

1 A hybrid job scheduler for shared physical clusters collects and uses job profiles or statistics from prior runs. 3

2 (Left) Hadoop architecture overview and (right) diagram showing colocation of Hadoop, MPI and NoSQL systems in one machine. 6

3 With YARN, applications run natively in Hadoop framework. [7] . . 6

4 Our proposed hybrid cluster job scheduling model vs. alternative models # 1 adapters and # 2 & # 3 virtualization 12

5 Cassandra and MPI jobs running together on 8 cores. 16

6 Greedy sort-merge scheduler. 20

7 A screenshot from IBM power graph screen. 23

8 Power consumption by node count. 24

9 Thermal output by node count. 24

CHAPTER I

INTRODUCTION

In many universities and research centers scientists depend on shared computing clusters for their computational needs. These scientists are usually from various disciplines such as Physics, Mechanical Engineering, Computer Science and Bioinformatics. Some of their computational jobs are CPU-intensive and some are data-intensive. They also use different distributed processing engines and associated middleware such as Message Passing Interface [1] [2], Hadoop-MapReduce [3], and any type of NoSQL engines to execute their jobs. As a worst case scenario (which is in-fact common in smaller scientific communities), people email each other to decide who runs a job first, which causes poor cluster utilization, or they all submit their jobs simultaneously leading to clashes and poor performance. Today, there is no easy-to-use, common scheduler to reorder and reprioritize these hybrid jobs to make better execution plans among different clustering technologies. In this thesis, we describe the design of a new MPI, Hadoop-MapReduce, and NoSQL hybrid job scheduler that uses jobs' execution profiles from prior runs and makes global scheduling decisions to improve overall system utilization and job throughput performance.

1.1 Motivation

Inspired by advances of technology and to answer particular application needs, scientists have constructed various kinds of computing platforms to perform High Performance Computing (HPC). Clusters, multi-clusters, grids, desktop or lightweight grids, peer-to-peer systems and lately HPC-clouds that propose different ways for executing workloads and applications. Hence integrating into or interfacing to different HPC and High Throughput Computing (HTC) environments had to be taken into

account.

1.2 Definition and Benefits of Hybrid Scheduling

In this context, the word hybrid can have multiple meanings:

- First, the interfaces or APIs of distributed processing technologies can be different (e.g. MPI vs. MapReduce) and these different middleware will be executed together on one physical platform.

- Second, the clusters physical resources can be heterogeneous consisting of relatively powerful and weaker machines in terms of CPU, memory and storage resources.

- Third, the new overlay hybrid scheduler should know that there are also other underlying cluster-specific job schedulers and operating system (OS)- level process schedulers, which creates a multi-level scheduling scenario.

The ideal hybrid scheduler should have the ability to make a combined scheduling decision considering all of the conditions above. Therefore, it would be fair to state that hybrid job scheduling is hard. Yet, we also observe the fact that in scientific communities similar jobs are run repeatedly over the same shared physical clusters as shown in Figure 1. Therefore, there is an opportunity to profile these different jobs in terms of their CPU, memory, disk bandwidth, and network usage and feed this information to a global hybrid scheduler for optimization.

The benefits of hybrid scheduling would include: *(1) convenience on the users side, (2) increased cluster utilization and (3) improved job isolation.* This thesis aims to investigate and highlight some of these potential advantages. User convenience refers to users not knowing (or needing to know) what OS, server capacity, virtualization, etc. technologies or issues exist. Cluster utilization refers to simultaneous use and effective utilization of all computing resources including CPU, memory, disk, and

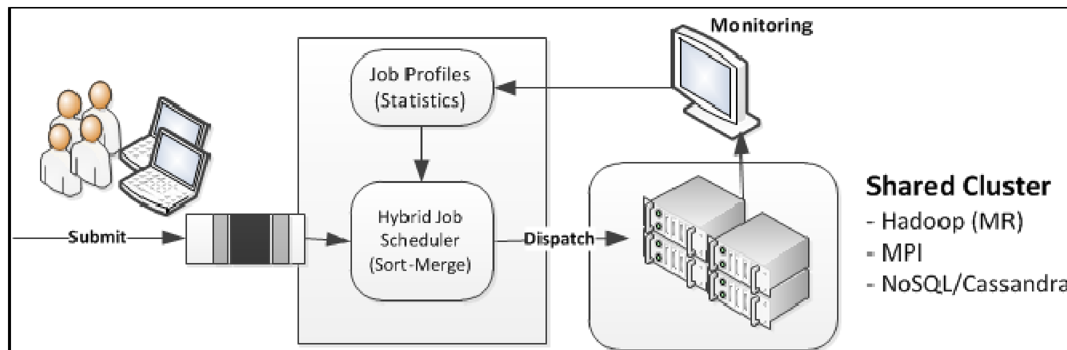

Figure 1: A hybrid job scheduler for shared physical clusters collects and uses job profiles or statistics from prior runs.

network (and not just CPU sharing). Finally, for job isolation we only seek an improvement over todays practice cross-platform and mixed (Hadoop, MPI, etc.) scheduling of many jobs, where there is practically “no isolation”. Otherwise, we do co-locate these jobs and there will be performance interactions among them.

In one aspect, the hybrid job scheduler virtualizes the underlying clustering technologies, creating a unified scientific cloud. The underlying cluster can be physical or virtual or implemented as a private cloud (OpenStack [8], etc.) or public cloud service (Amazon AWS, etc.). The end users do not have to know these details. In addition, they also would not have to manage the cluster middleware, i.e. know whether MPI, Hadoop, NoSQL (HBase [6], Cassandra [9], MongoDB [10]) are installed properly and which versions are installed. These decisions are all left to the service providers. This setting allows overlapping the research cycles of different departments and disciplines. One scientist running HPC jobs only, would not fully utilize all the cluster resources (e.g. 100% CPU, but not both CPU and disk), but when all tasks are put together better cluster utilization is achievable.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Background

The need for clusters have been driven primarily by HPC applications in the past. Today, there is an emergence of High Throughput Computing (HTC) applications that contend for the same cluster resources. The latter is thought to be more I/O intensive. Many organizations including universities, telecommunication operators, media & social networks, financial institutions and governments generate terabytes of data each day and attempt to insert this high "volume" data into databases that already contain petabytes. Data come from different sources such as sensors, various web or system logs, and mobile phone call detail records (CDR). The mixing of these different data types results in a wide "variety" of data to be processed and stored (e.g. unstructured text, semi-structured XML-JSON, structured CSV, or binary audio or video data). The term "Big Data" is used to refer to challenging data management problems that arise due to the high Volume, Velocity, Variety, and Veracity (4Vs) of the data [11]. Veracity refers to the brokenness of the data. Relational database management systems cannot cope with 4Vs of data very well, therefore Hadoop and NoSQL systems have quickly penetrated into the scientific and enterprise data infrastructures. Consequently, a variety of the cluster technologies including Hadoop, MPI and NoSQL started to get used over the same, shared physical clusters.

2.1.1 Hadoop

Apache Hadoop is an open-source framework mostly written in Java designed for supporting data-intensive distributed applications. Hadoop framework mainly consists of two components called MapReduce (MR) and Hadoop Distributed File System (HDFS). The framework also has other supporting projects that are shown in Figure 2. These projects provide and include database (HBase), data warehousing (Hive/Pig) and other management (Oozie, Chukwa, Zookeeper) functionality and applications. The data to be processed is stored in large fixed-sized chunks (64MB default) by HDFS for both performance and reliability purposes. Figure 2 also shows the Hadoop architecture. HDFS consists of a NameNode (master) and several number of DataNodes (slaves). MapReduce (MR) is a distributed job execution system consisting of a JobTracker (master) and TaskTrackers (slaves). Jobs will be split into many Map and Reduce tasks and sent as JAR files to many machines containing the data in HDFS. JobTracker distributes these client- submitted jobs and TaskTrackers track the progress of local Mappers and Reducers. The Map phase handles data transformations in each worker node and is followed by the sort-merge phase. The Reduce phase aggregates sorted data and outputs the results. Some jobs will only transform the input log files by extracting desired fields and writing them out in the new format, therefore requiring no Reduce phase. Chen, et al. [12] give a detailed analysis of MR workloads from two large-scale production Hadoop systems (from Yahoo and Facebook) and find great differences between MR use cases. They conclude that “no single benchmark can capture such diverse behavior” [12].

We used the Hadoop version 1.2.0 in our experiments. The previous Hadoop versions 0.22.x and the new beta version 2.x are also being used extensively by the industry. In Hadoop 2.x, HDFS has a namespace federation feature, which provides scalability beyond 10,000s of nodes, since the NameNode is also distributed.

In its 2nd version, MapReduce (or MRv2) came with a major change at the

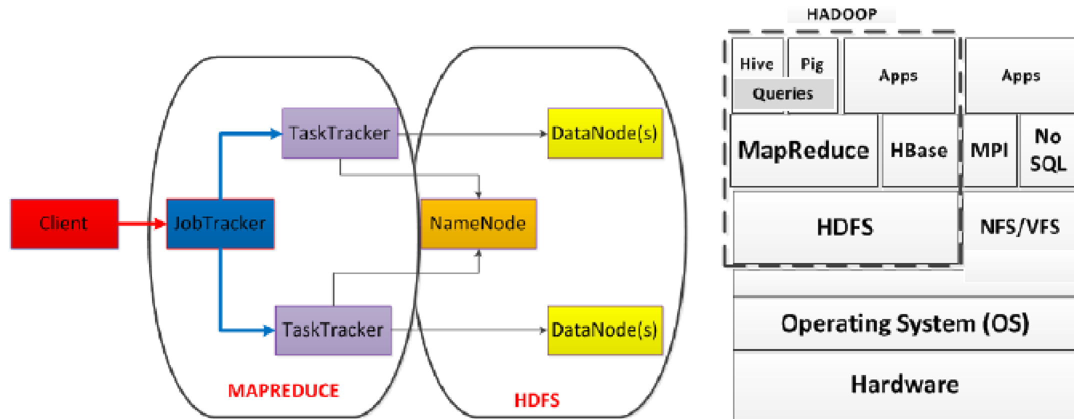

Figure 2: (Left) Hadoop architecture overview and (right) diagram showing colocation of Hadoop, MPI and NoSQL systems in one machine.

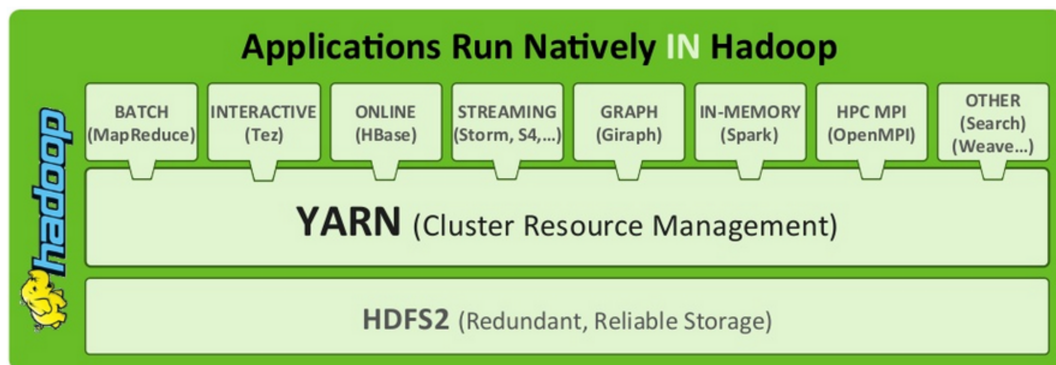

Figure 3: With YARN, applications run natively in Hadoop framework. [7]

JobTracker side. The new idea is one ApplicationMaster to schedule and monitor jobs and one ResourceManager to manage resources as shown in Figure 3. This resource manager in MRv2 is also known as YARN, which stands for "Yet Another Resource Negotiator", and it provides flexible programming, improved cluster utilization and fast introduction of new services over Hadoop.

YARN scheduler allocates resources for a job and it does not perform any monitoring or tracking process. On the other hand, it is extensible with plugins. So, it can be extended with more intelligent scheduling algorithms.

2.1.2 Message Passing Interface (MPI)

MPI is an HPC middleware used for distributed inter-process communication and high performance on both Symmetric Multi-Processor (SMP) machines and on workstation clusters. It is well-standardized and portable. We use the MPICH2 [1] implementation of the standard, but other implementations such as the Open MPI [2] also exist. In our previous HPC-focused work [15], we implemented a cloud computing platform service for Finite Element Analysis (FEA). FEA is a generally applicable numerical method to approximately solve partial differential equations and requires HPC setups. Application areas of FEA include mechanical structural analysis, heat transfer, fluid dynamics, acoustics, and electromagnetic modeling. We found that if the FEA job can be processed effectively on a single multi-core server (i.e. with enough CPU cores and RAM), we should use the SMP-style multi-processing. If not, the job will be distributed over multiple servers/processes and MPI will be used for inter-process communication. The scheduler has to make these decisions optimally for hundreds of jobs, simultaneously, which was the core contribution of our previous work [15].

2.1.3 NoSQL

NoSQL stands for Not Only SQL and NoSQL systems were created for needs in different aspects. Some of these include need for flexible data structures, horizontal scaling, high performance and distributed computing. The most commonly used NoSQL systems include HBase [6], Cassandra [9], CouchDB, Redis, Neo4j and MongoDB [10].

We worked with HBase, Cassandra and MongoDB, but there were several reasons for choosing Cassandra for testing. We did not select HBase since it also works on Hadoop (and we needed a variety of technologies to mix). MongoDB has a great data structure which is based on Javascript Object Notation (JSON). But, we omitted MongoDB from this work, since its cluster setup cost (especially sharding phase) was higher for us than Cassandra's setup at this time. We selected Cassandra and

continued with it as our NoSQL system.

Apache Cassandra [9] is a Java-based NoSQL database initially developed by Facebook. One of its differentiators is that, it has no central servers and uses a pure peer-to-peer approach for scalability. Cassandra does not enforce a schema, thus it is also a $< key, value >$ store like Amazons Dynamo [16] NoSQL system. But it is optional to make Cassandra behave as it has a defined schema. It stores the data in a HexByteArray format. The "table" concept in DBMS is represented by a "column family" in Cassandra. Cassandra tries to hold all the data to be processed in a memory-resident "memtable" and holds the extra data in a disk-resident "sstable". It has two types of initial partitioners ; the random partitioner evenly distributes data and the workloads, whereas the order-preserving partitioner performs better with range queries. We used the random partitioner and no replication in our experiments. Cassandra uses one thread for each worker. However, it also uses additional CPU resources for other background jobs, like indexing and flushing, which shows as random spikes in its CPU profile, an example of which is shown in Figure 5.

2.2 Related Work

While most state-of-the-art cluster job scheduling tools declare heterogeneous system support, they primarily refer to different hardware capabilities and not middleware interfaces. For HPC jobs, the famous MPI scheduler called LSF (Load Sharing Facility) [13] is primarily used for submitting MPI jobs to MPI clusters. The scheduler adjusts execution order of submitted MPI tasks based on their node-core ("-np") requirements and their priorities. In our previous work [15], we showed that additional savings are possible above the MPI-level with task-aware cluster job scheduling especially for FEA. In this thesis, we demonstrate potential savings for hybrid set of jobs, and not just HPC. Previous research on Grid scheduling and scientific workflows focuses on data or task dependencies [20], [21]. Since our jobs belong to completely

different cluster middleware we assume there will be no such dependencies and issues among our hybrid jobs and we do not need to build task-graphs. There are numerous past publications on Grid scheduling, but since none of these prior works support Hadoop, MPI and NoSQL we skip a detailed comparison here for brevity. For HTC jobs, there exists two Hadoop schedulers called FIFO and Fair Scheduler [17]. Using these schedulers and careful job pool settings, prioritized co-scheduling of MapReduce jobs is possible. However, these Hadoop schedulers do not accept MPI jobs today.

We are not aware of any system that integrates MPI, Hadoop, and NoSQL systems as we do in this paper. The closest systems to ours include the SLURM Scheduler plugins and HTCondor's ClassAds. "SLURM is an open-source resource manager designed for Linux clusters". It provides a framework for allocating, starting, executing, and monitoring work (typically a parallel job) on a set of allocated nodes for a specified duration. SLURMs modular design accepts plugins inluding scheduler plugins that implement the SLURM scheduler API. To name a few, builtin scheduler plugin implements first-in-first-out, backfill raises the priority of jobs to start them earlier without any delay, wiki plugin uses the Maui Scheduler, and wiki2 uses the Moab Cluster Suite. "HTCondor's ClassAds are a flexible mechanism for representing the characteristics and constraints of machines and jobs in the HTCondor system". HTCondor currently has no direct MapReduce support. Developers of the vgreen system investigated energy-efficient scheduling [26] of Virtual Machines (VM). They profiled the power and performance characteristics of several VMs and designed a power-oriented scheduler. Yet, they reached to a conlusion similar to ours. They found that co-scheduling jobs with similar characteristics (e.g. two CPU-intensive tasks) is not energy-efficient, whereas placing heterogeneous jobs together can save up to 20% energy. The reason is that under-utilized resources still consume energy and better utilization of all resources concurrently leads to lower energy use/jobs completed. In our case, we achieved up to 37% improvement in throughput performance and the

impacts on energy efficiency constitute interesting future work.

Nowadays, there are two Apache projects related with this topic. Apache Helix [5] provides resource management and Apache Mesos [4] provides cluster management and resource sharing frameworks. Apache Mesos can run MPI and Hadoop, but it has seperated schedulers for both of them.

Our previous work [14] includes a part of this research. We investigated to run jobs together with different parameters and showed preliminary results.

CHAPTER III

A DISCUSSION ON VIRTUALIZED CLUSTERS

One could simply use native or hosted machine virtualization to co-locate different clustering middleware (or platforms) on shared physical cluster infrastructure as shown in Figure 4. In comparison, our proposed hybrid job scheduling model has one of the simplest forms. In our proposed model, different clusters are placed under the common scheduler, which profiles all the submitted jobs and decides to co-locate (i.e. dispatch together) those who use different, heterogeneous types of resources. In this regard, the hybrid job scheduler virtualizes the underlying clustering middleware, creating a unified scientific cloud. Alternative # 1 denotes two cases called Hadoop over MPI or MPI over Hadoop, where the former refers to converting MapReduce source or binary code using an adapter to MPI code to run over an MPI cluster [24] and the latter refers to the new YARN approach mentioned in previous sections. Alternatives # 2 and # 3 denote a native and a hosted virtualized model, respectively, for mixed Hadoop-MPI virtual clusters. These two virtualization alternatives are quite similar with respect to the higher-layer job scheduling software. However, the natively virtualized clusters (# 2) provide additional advantages over hosted Virtual Machines (VM) including dynamic resizing and migration of VMs, and scale easily beyond a single physical machine. As tradeoffs, virtualized clusters have both performance overheads and additional VM management issues. Another concept that improves co-location of VMs is called VM ensembles. Since the switches and routers are expensive resources in large-scale data centers, systems such as Net-Cohort [18] discover VM ensembles and co-locate them to reduce inter-VM traffic. Testing is idea with Hadoop-MPI-NoSQL virtual clusters and comparing their performances with

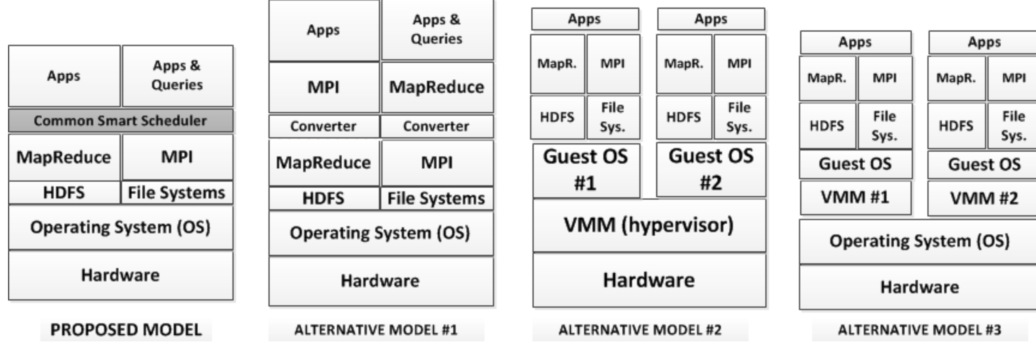

Figure 4: Our proposed hybrid cluster job scheduling model vs. alternative models # 1 adapters and # 2 & # 3 virtualization

our findings constitutes interesting future work.

Recent findings [23] also suggests that virtualization may not always be the answer in the HPC world for multiple reasons. First, HPC systems want to squeeze the last bit of performance from the systems whereas virtualization can mean up to 10% of performance hit, since the hypervisor adds yet another layer between the physical resources and the applications. The second reason is HPC systems are already designed for being shared among a community of people in unified scientific communities (i.e. with similar HPC jobs) the utilizations can be relatively high [23]. Therefore, the motivations that drive the use of virtualization (i.e. increase utilization, save power and same money) are not always there for HPC. Similarly for Hadoop clusters, experts neither recommend virtualization nor RAID, but to save the money and buy more nodes and disk spindles. While the fastest 15K rpm disks are common in HPC, they are not recommended in Hadoop and other HTC clusters, as one can get more parallel I/O bandwidth by buying more 7200 rpm disks. A third reason for why HPC systems may not prefer virtualization is that the hypervisors can simply put limits on the number of cores (e.g. 8) that can be taken by a virtual machine. In that case, a Symmetric Multi-Processor (SMP) system with many (e.g. 512) cores will have to be divided into hundreds of VMs requiring excessive use of messaging among these machines. This limit would practically turn a supercomputer into a group of

mini-clusters. In addition, Graphical Processing Unit (GPU) usage is quite common in HPC workloads today, whereas it is not possible to connect more that one VM to GPUs. Some of these limitations are being solved over time, but the world of virtualization seems to be always a step behind in trying to virtualize the next I/O layer (e.g. high-availability SAN storage or Infiniband) or the emerging system technology. In summary, virtualization is a good choice in dynamic test and development environments where reconfiguration speeds and elasticity are critical. However, HPC and HTC environments run similar and predictable types of workloads and require deterministic performance.

CHAPTER IV

PERFORMANCE ANALYSIS AND RESULTS

4.1 Cluster Environment and Profiling Utilities

The experiments for this work were conducted on a small, but powerful, private cluster of two IBM HS22 Blade Servers in a high-performance blade chassis with 1Gbps connectivity. Each server has two 2.40GHz Intel Xeon Quad-Core E5620 CPUs, 24GB Memory, and two 72 GB 15000rpm disks configured as RAID1. In Hadoop and Cassandra experiments, we also used a partition from the Storage Area Network (SAN) to store and process the benchmark data. We installed RedHat Linux 5 on these servers. For profiling we used the *sysstat* package [19] and its *pidstat* command to track CPU, memory and disk read-write bandwidth usage of the processes. "The sysstat utilities are a collection of performance monitoring tools for Linux. These include sar, sadf, mpstat, iostat, nfsiostat, cifsiostat, pidstat and sa tools" [19]. All experiments were run 3 times and the results were averaged.

4.2 Workloads

As Hadoop workload we used the classical *terasort* benchmark, which is more IO-intensive than our MPI task for our cluster. For this experiment, we first created a large (15 GB) file filled with random, unsorted characters using the teragen program. Next, we used terasort to read & sort the unsorted data and write back the sorted data. As MPI workload we used the dbworld benchmark, which solves a system of linear equations. The MPI command was executed as follows: $< mpirun\ -np\ 8\ dbworld\ mpi\ -n\ 1536\ -m\ 1536 >$, where np denotes node-core counts and $n \times m$ are the matrix row-column sizes. We used the MPICH2 implementation [1] of MPI.

During Hadoop comparison, the matrix size was asjusted to 1348×1348, so that one task does not finish much sooner than the other. This allowed us to simulate a scenario where the job pipeline is always full. For Cassandra benchmarking, we wrote a Java program that reads the XML-formatted CDR logs of a real mobile telecom company, indexes and inserts these transformed logs into Cassandra using the Astyanax client adapter by Netflix. For data distribution we used the random sharding technique and no data replication. Table 1 shows the average CPU-memory usage percentage, and read-write bandwidth usage in MB/second for these 3 workloads obtained using the pidstat tool. Basically, the MPI and Cassandra benchmark jobs were CPU-intensive (with 99% CPU utilization on average) and the Hadoop terasort job was more IO-intensive than others (10MB/sec reads, 40 MB/sec writes) and had 25% average CPU usage.

Table 1: Resource usage profiles of benchmark cluster workloads.

	CPU	**Memory**	**R-W Bandwidth**
MPI	99%	0.5%	0-0 MB/sec
Cassandra	99%	15%	0-4.3 MB/sec
Hadoop	25%	2.2%	9.8-40.2 MB/sec

Our focus in this thesis was to mix and run different types of jobs together for different middlewares, therefore we only analyzed interactions among jobs for multiple core count settings and did not vary the system-specific parameters internally. If we vary the size of either one of the benchmarks (MPI problem size, terasort data size, or Cassandra data size), their executions times will be longer, but the learning outcomes will be the same. Figure 5 shows a sample CPU usage profile of two concurrent jobs, MPI and Cassandra when running on the 8 core server with 4 worker threads dedicated to each. MPI is relatively stable and completes around 16 minutes, whereas Cassandra has a more jittery picture (due to its background jobs) and completes

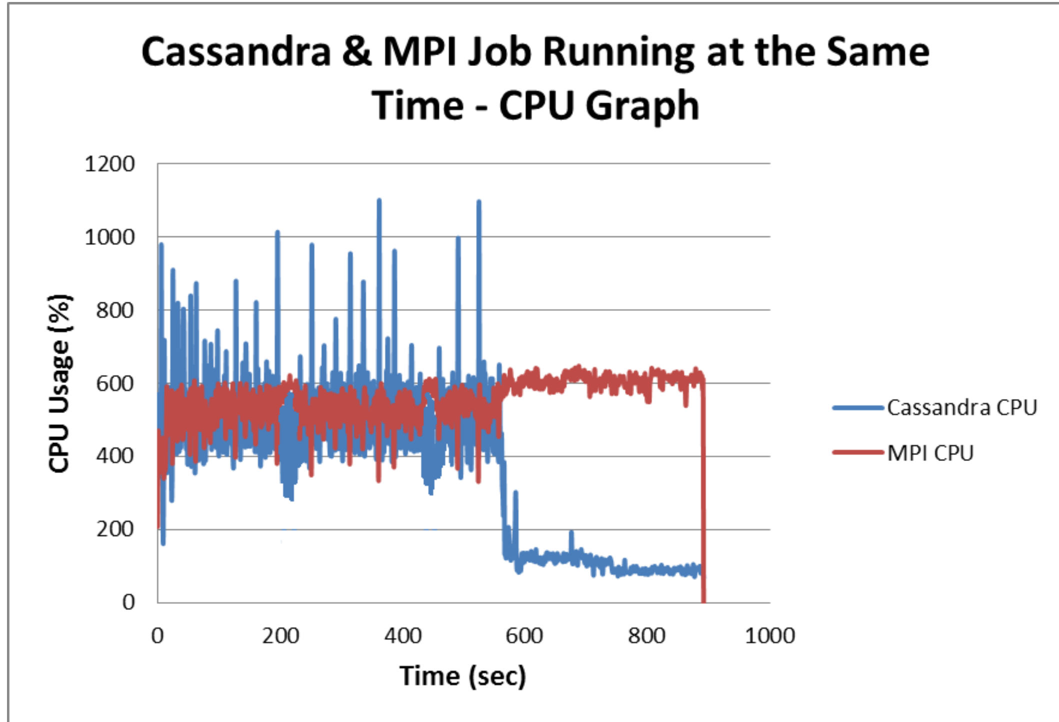

Figure 5: Cassandra and MPI jobs running together on 8 cores.

around 10 minutes.

4.3 CPU-intensive Jobs and Cluster-Level Job Prioritization

In this section, we investigate whether we can preempt, reorder, reprioritize (i.e. take control of) different types of cluster jobs when they are executing simultaneously over the same cluster. We increase job priorities to let them use more share of the CPU resources and vice versa. Table 2 shows the results of co-scheduling of two CPU-intensive tasks MPI and Cassandra on a single 8-core (physical) server for different worker (4,8,16) settings. For MPI, the term "worker" refers to the number of cores used (i.e. -np) and for Cassandra it refers to the number of loader client threads. "Standalone" refers to non-overlapping, i.e. serial and mutually exclusive, execution of these two tasks. With 4 workers the total time to finish both tasks is 20.8 minutes (11.8+9.0). With 8 workers the total time goes down to 15.4 minutes (i.e. 26%

improvement), but does not improve anymore for 16 workers since this server has only 8 physical cores and the hyperthreading does not contribute much for ~100% CPU-intensive tasks. The Cassandra task benefits more from increased core count. When the two tasks are started simultaneously with the same priorities and they execute in parallel, the OS-level process scheduler shows positive effect. Although both tasks execute longer than their serial execution counterparts (e.g. 16.2 mins for MPI and 10.1 mins for Cassandra 4) the system throughput increases from ~6 jobs/hour to ~8 jobs/hour: both tasks have completed at 16.2 minutes for 4 workers, 15.0 minutes for 8, and 13.8 minutes for 16 workers. This shows that 10-20% savings are possible even for CPU-intensive tasks

Table 2: Results of co-scheduling MPI and Cassandra jobs.

Worker # per-job	**Execution**	**MPI (min)**	**Cassandra (min)**	**Total (min)**
4	standalone	11.8	9	20.8
	normal - combined	16.2	10.1	16.2
	nice -n -20 for M	14.9	18.4	18.4
	nice -n +19 for M	18.4	10.5	18.4
	nice -n -20 for C	19.9	10.5	19.9
	nice -n +19 for C	13.4	17.7	17.7
8	standalone	9.4	6	15.4
	normal - combined	15	8.6	15
	nice -n -20 for M	11.8	16.3	16.3
	nice -n +19 for M	17.6	7.5	17.6
	nice -n -20 for C	16.8	7.8	16.8
	nice -n +19 for C	10	16.1	16.1
16	standalone	9.5	5.8	15.3
	normal - combined	13.8	10.7	13.8
	nice -n -20 for M	11.3	16	16
	nice -n +19 for M	16.5	6.6	16.5
	nice -n -20 for C	15.6	6.8	15.6
	nice -n +19 for C	10.6	15.4	15.4

Next, we change the priorities of the tasks using the Linux nice command to see if we can control the CPU-shares at the cluster level at run-time. In Cassandra, we

change the priority of the server, whereas in MPI the priority of the job is directly changed. The priority levels can range between [-20,+19], where -20 is the highest and +19 is the lowest priority. The results are affirmative in terms of job control. The job with higher priority can perform comparable to its standalone performance, whereas the niced job has to wait (or run slow) until the high-priority job completes. This demonstrates that we can order, prioritize or re-prioritize jobs belonging to different cluster technologies with respect to each other.

4.4 Hybrid Scheduling of CPU and IO-intensive Jobs

Table 3 shows the serial and hybrid execution of MPI and Hadoop jobs. We used the 15 GB data for the Hadoop terasort benchmark job. The serial execution completed in 1272 seconds total, whereas the hybrid execution completed around 800 seconds. This results in savings of e 37%. The savings are significantly higher compared to co-scheduling of two CPU-intensive jobs, which resulted in 10-20% savings. Note that the execution time of the CPU-intensive MPI job is ~20% higher (800 sec.) in hybrid form (i.e. MPI-Hadoop co-scheduled) compared to its serial execution time (671 sec). The Hadoop job also took e 25% longer, but they both finished at 800.3 seconds. These job latency increases can be attributed to the 20-25% on average CPU usage of the Hadoop job.

Table 3: Results of co-scheduling of MPI and Hadoop jobs.

	MPI(sec)	**Hadoop(sec)**	**Total Time(sec)**
Serial Execution	671.7	601.0	1272.7
Hybrid Scheduler	800.3	769.0	800.3

4.5 Generalized Hybrid Scheduler

We briefly describe a greedy sort-merge hybrid job scheduler. The jobs are profiled based on their usage of CPU, memory and disk bandwidth resources. Job $i(j_i)$ is

represented by a triplet $j_i =< c_i, m_i, d_i >$ where c_i, m_i, and d_i represents the average CPU, memory and read+write disk I/O usage of that job, respectively. In our current simplified model the reads and the writes are not differentiated in terms of performance as their sequential and sustanied throughput over different storage technologies is usually comparable. The simplest model assumes that each resource is orthogonal the other resources and can accept jobs as much as its capacity. Three seperate lists are kept for each resource (CPU, disk, memory) in the descreasing order of usage. Jobs are picked from the three lists in a round-robin fashion and the corresponding capacities are subtracted from the resources, respectively. This way, a CPU-intensive job will first be matched with a disk-intensive job and so on until all jobs are complete. Our current scheduling system works in a master-slave fashion. In the future, we plan to implement a fully-decentralized version using group communication (e.g. jGroups library). The cluster job commands for MPI, Hadoop, NoSQL are submitted to a master node, which holds the master job queue. Clients use our "*hsub*" command, which stands for hybrid job submit and is similar to the qsub-bsub (LSF) commands. Sample hsub commands are given below:

```
hsub -t mpi -c "mpirun -np 16 ./dbworld params";

hsub -t cassandra -c "java -jar import.jar params";

hsub -t hadoop -c "hadoop jar examples.jar terasort /in /out";
```

where -t denotes the job type (mpi,hadoop,cassandra) and -c parameter gives the command string (parameters are omitted).

Now, we briefly describe a preliminary greedy sort-merge hybrid job scheduler. The jobs are profiled based on their usage of CPU, memory and disk bandwidth resources. Job $i(j_i)$ is represented by a triplet $j_i =< c_i, m_i, d_i >$ where c_i, m_i, and d_i represents the average CPU, memory and read+write disk I/O usage of that job, respectively. In our current simplified model the reads and the writes are not differentiated in terms of performance as their sequential and sustanied throughput over

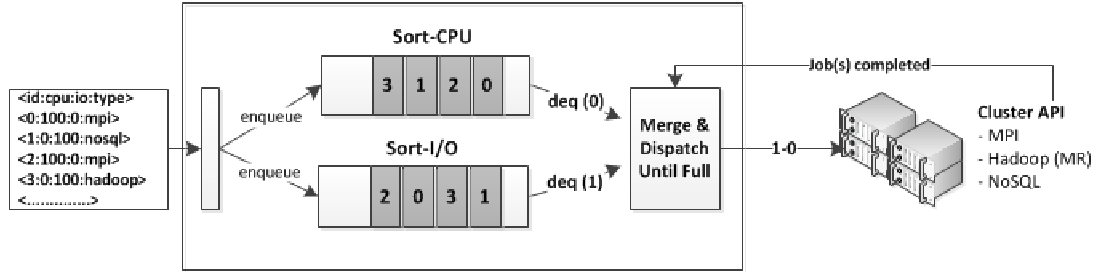

Figure 6: Greedy sort-merge scheduler.

different storage technologies is usually comparable. The IOPS-Latency curve for the storage system is assumed to be "L" (mirrored-L) shaped: Latency stays constant until 100% utilization and then goes to infinity. The simplest model basically assumes that each resource is orthogonal to (non-dependent on) the other resources and can accept jobs as much as their capacities. The jobs are assumed to be embarrassingly parallel, thus the messaging (network) overheads are also ignored by this scheduler. We also assume jobs have equal priorities and their execution times are also the same (1 epoch   10 minutes in the previous benchmarks). If a job has portions that display different profiling characteristics, those portions will be treated as different jobs by the scheduler after profiling.

Figure 6 illustrates the operation of this scheduler with a simple example. Jobs belonging to different clusters are submitted to the scheduler by various clients. In this example, a CPU-intensive job will be followed with an IO-intensive job. The scheduler keeps separate lists for each resource (CPU, disk, memory) in the decreasing order of their usage of that particular resource. Jobs are enqueued to all of the lists and then dequeued from the lists in a round-robin fashion. The corresponding capacities are subtracted from the resources, respectively. This way, a CPU-intensive job will first be matched with a disk-intensive job and so on until all jobs are complete. We generated two workloads two simulate these scenarios: first, an HPC-HTC mix (i.e. CPU-intensive and IO-intensive mix) and second completely random mix on all dimensions (cpu, memory, disk). We compared different job sorting strategies using

1000 simulated jobs. We have written about 500 lines of Java code for the hybrid scheduler simulation. We briefly report the results here. Since the HPC-HTC mix was generated for an ideal case of 100% CPU-intensive ($< 100 : 0 : 0 >$) and 100% IO-intensive ($< 0 : 0 : 100 >$) mix, the greedy scheduler was able to finish 1000 such jobs in 500 epochs. That is 2x performance improvement (50% savings in total time) over a single CPU-based or a single IO-based sorted scheduler using the same logic, but for a single queue. For the random mix, the two schedulers show similar performance around ~900 epochs each.

CHAPTER V

HYBRID JOB SCHEDULING FOR ENERGY EFFICIENCY

We showed that with a hybrid job scheduler we can reduce system resource usage for a batch of jobs. In this section, we investigate the impact of hybrid job scheduling on cluster power consumption. Dheeman et al. state that "Power consumption is a critical design parameter in modern data center and enterprise environments, since it directly impacts both the deployment (peak power, delivery capacity) and operational costs (power supply, cooling)." [26]. Greedy or brute-force FIFO job scheduling for clusters are inefficient. Physical infrastructures loaded by high performance jobs produce more heat. That means we need even more power for cooling the system to run it properly.

5.1 Measuring Energy Consumption

IBM Blade chassis we used for testing has a dashboard for monitoring power consumption. It serves information hierarchically organized to provide an overall summary of different power domains. It has 3 power domains; maximum power limit, remaining power and allocated power. The dashboard gives information about status, power modules, power management policy, maximum power limit and power in use. Another important point is total thermal output of chassis. It is calculated based on the total power in use (1 Watt = 3.412 BTU/hour). The data for power consumption is taken from "chassis power consumption" panel. We can set the data sampling interval.

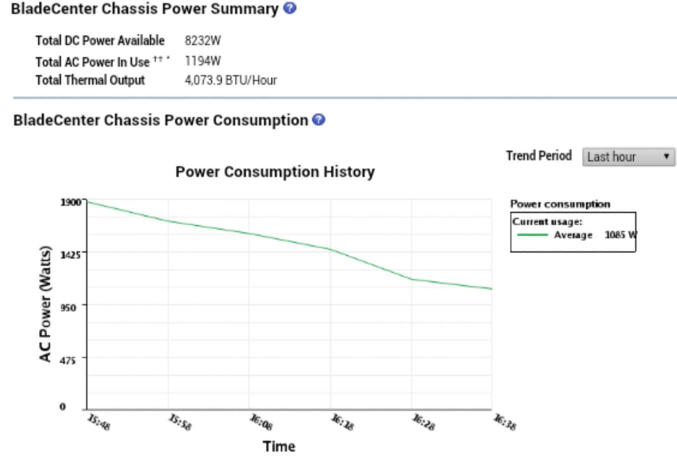

Figure 7: A screenshot from IBM power graph screen.

We repeated every experiment 3 times and averaged the result. After an experiment finishes we stopped all jobs and waited for nodes to return to normal state.

5.2 Workloads

An optimal hybrid scheduler can give us the ability to use the computing resources more effectively. First we measured power consumption and thermal output by changing node count. Before the experiments we set the data sampling interval to 10 minutes and we took the values in total AC power in use.

To obtain Figure 8 and 9, we started with 12 blade servers running in the chassis and a total power consumption of 2040 Watts and a total thermal output of 6950 BTU/hour. Next, we powered down servers one by one until 4 servers were only left operational. The power usage also decreased linearly.

According to a quadratic regression the formulas we obtained are:

$$power(x) = 7.2x^2 + 4.2x + 965 \tag{1}$$

$$thermalOutput(x) = 25.5x^2 + 4.7x + 3315 \tag{2}$$

According to a linear regression the formulas we obtained are:

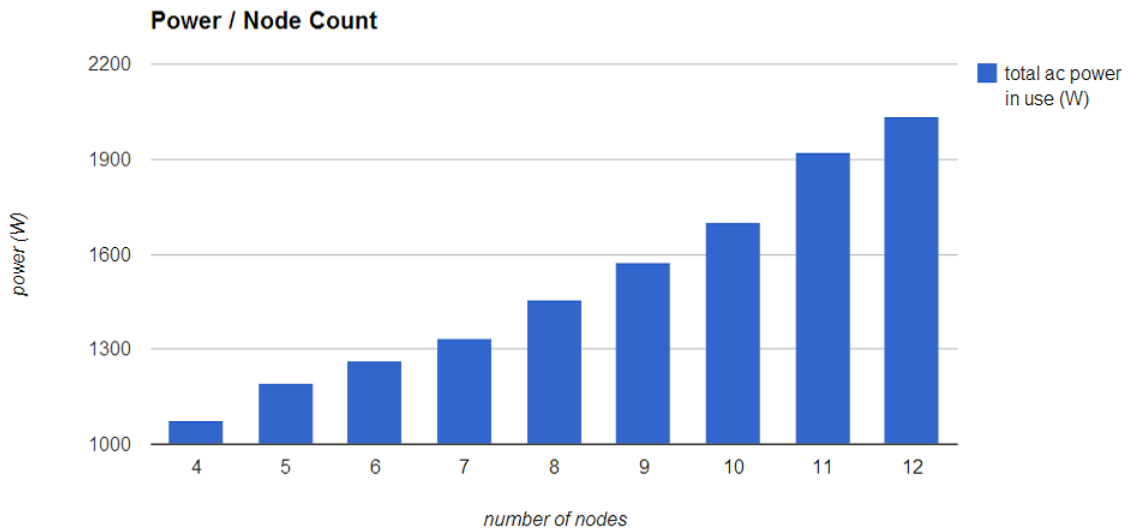

Figure 8: Power consumption by node count.

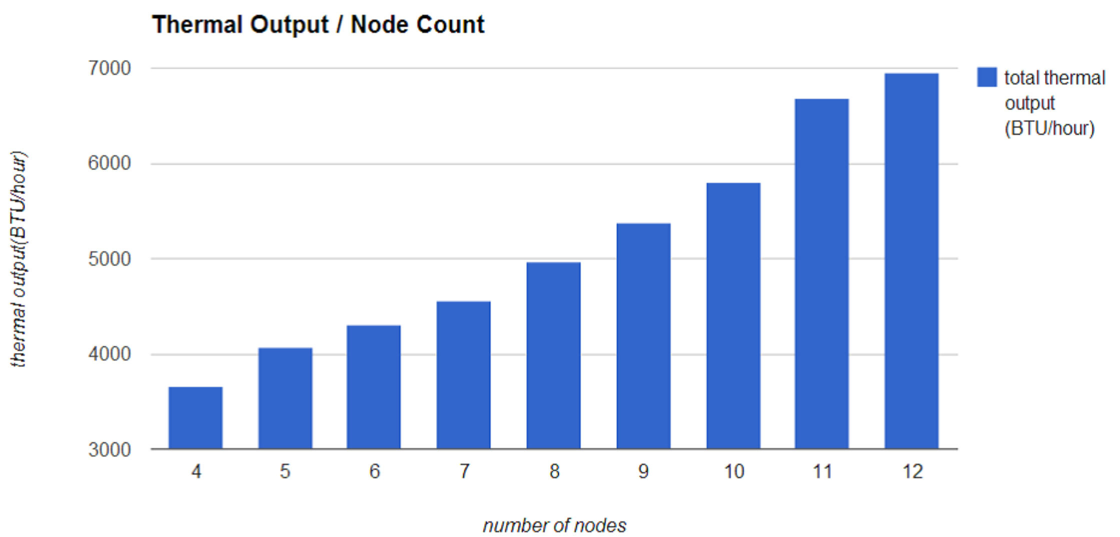

Figure 9: Thermal output by node count.

$$power(x) = 116x + 572 \quad (3)$$

$$thermalOutput(x) = 396x + 1953 \quad (4)$$

Subjected to x, node count; equations 1 - 3 are for power consumption with the coefficient of determination value (R^2) 0.99 and 2 - 4 are for thermal output with the coefficient of determination value 0.97. The change of thermal output and power consumption shows that every node effects whole chassis' at appreciable level.

Next, we ran our cluster jobs again, but this time monitored the power usage instead of performance. For this experiment we used MPI and Cassandra jobs.

Table 4: Power consumption and thermal output by running jobs on 2 nodes.

	Power (W)					**Thermal Out.(BTU/hour)**				
	# 1	# 2	# 3	avg.	std. d.	# 1	# 2	# 3	avg.	std. d.
Idle	2038	2012	2059	2036	23.5	6942	6902	7006	6950	52.5
MPI	2122	2110	2086	2106	18.3	7204	7176	7125	7168	40.1
Cassandra	2076	2099	2090	2088	11.6	7113	7132	7127	7124	9.9
MPI & Cas.	2122	2114	2116	2117	17.3	7249	7216	7226	7230	16.9

Based on our experiment results listed on Table 4 for MPI job, 2 nodes use 70 W more power and gives 218 BTU/hour more thermal output. Cassandra job uses 52 W more power and gives 174 BTU/hour more thermal output. When we combine jobs together they give 81 W more power and 280 BTU/hour more thermal output.

When we add node energy consumption according to equation 3, one single node needs 116 W. For the power domain, power usages changes to 302 W for standalone MPI, 284 W for standalone Cassandra and 313 W for concurrent MPI-Cassandra jobs. Considering time spent on these jobs:

serial execution $= 9.4min \times 302W + 6.0min \times 284W = 4542Wmin$

parallel execution = $15mins \times 313W = 4694Wmin$

According to calculations without considering chassis power we get worse energy consumption of 3.3% . When we consider energy consumption of chassis and cooling devices we have to add 572 W more power usage.

serial execution = $9.4min \times 302W + 6.0min \times 284W + 15.4min \times 572W = 13350Wmin$

parallel execution = $15mins \times 313W + 15mins \times 572W = 13275Wmin$

These calculations show that we get 0.5% better together.

If we consider the power usage of all 12 servers on the chassis which were operational, but idle, during the experiment then we obtain the power results in Table 4. According to the values in this table, the power savings or loss with respect to total power usage (including servers running the workloads + other servers and the chassis) can be calculated as:

serial execution = $9.4min \times 2106W + 6.0min \times 2088W = 32324Wmin$

parallel execution = $15mins \times 2117W = 31755Wmin$

So, we can save 1.8% energy when we run both simultaneously.

These results shows that blade chassis's power consumption is at remarkable level. Considering time saving of multiple CPU intensive jobs' execution, saving ratios are close. We can run jobs on all nodes for increasing total throughput and reducing overhead of chassis. As an inference, we can get more saving linearly with running CPU and I/O intensive jobs together. Another approach can be setting power saving options of chassis if it is available, ventilation and network bandwidth according to working node count.

CHAPTER VI

CONCLUSION

In this thesis we discussed feasibility and efficiency of running different type of jobs together. We analyzed different models for hybrid job scheduling over shared physical clusters. As a result of experiments we showed that it is possible to improve cluster utilization and overall throughput even jobs are both CPU intensive. On the other hand it is better to schedule I/O and CPU-intensive jobs together first. We get 37 % improvement for this case besides 2-20 % for CPU intensive jobs. Scheduling jobs that belong to different clustering technologies (Hadoop, NoSQL, MPI) together effectively is possible. At that point it is good to get characteristic of a middleware. We offer a dynamic characterization based on statistic data collection cycle.

We can control the execution speeds of hybrid jobs by adjusting their priorities at runtime. This gives to ability of switching job finishing time earlier without killing other concurrently running job.

We can save some energy and thermal output. This can be remarkable for large clusters for reducing energy costs.

Bibliography

[1] MPICH project, http://www.mpich.org

[2] Open MPI project, http://www.open-mpi.org

[3] Apache Hadoop Project, http://hadoop.apache.org/

[4] Apache Mesos, http://mesos.apache.org/

[5] Apache Helix, http://helix.incubator.apache.org/

[6] Apache Hbase project, http://hbase.apache.org

[7] Hortonworks, Modern data and hadoop, http://hortonworks.com/hadoop-modern-data-architecture

[8] Openstack http://www.openstack.org/

[9] Apache Cassandra, http://cassandra.apache.org/

[10] MongoDB by 10gen, http://www.mongodb.org/

[11] What is Big Data? http://www-01.ibm.com/software/data/bigdata/

[12] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating MapReduce performance using workload suites. 2011 IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 390-399, 2011.

[13] IBM Platform Computing, LSF, http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/

[14] I. Ari, U. Kocak, Hybrid Job Scheduling for Improved Cluster Utilization, Middleware for HPC and Big Data Systems, Aug, 2013

[15] I. Ari, N. Muhtaroglu, Design and implementation of a finite element service in the cloud, Advances in Engineering Software, Nov, 2013

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In SOSP, vol. 7, pp. 205-220. 2007.

[17] Hadoop Fair Scheduler, http://hadoop.apache.org/docs/stable/fair scheduler.html

[18] L. Hu, K. Schwan, A. Gulati, J. Zhang, and C. Wang. Net-cohort: Detecting and managing vm ensembles in virtualized data centers.In Proceedings of the 9th International Conference on Autonomic Computing, pp. 3-12. ACM, 2012.

[19] Sysstat Utils, sebastien.godard.pagesperso-orange.fr/

[20] Ewa Deelman, Grids and Clouds: Making workflow applications work in heterogeneous distributed environments, International Journal of High Performance Computing Applications, Vol (24), No. 3, pp. 284-298, Fall 2010

[21] Gideon Juve, Ewa Deelman, Resource provisioning options for large-scale scientific workflows, In Proc. of SWBES Workshop with IEEE E-Science 2008

[22] J. Zhan, L. Zhang, N. Sun, L. Wang, Z. Jia, and C. Luo, High Volume Throughput Computing: Identifying and Characterizing Throughput Oriented Workloads in Data Centers, 2012 IEEE 26th International Parallel and Distributed Processing Symposium.

[23] K. Hwang, G. C. Fox, J. J. Dongarra, Distributed and Cloud Computing, From parallel processing to Internet of things, Morgan Kauffman, 2012 Elsevier

[24] S. J. Plimpton, Karen D. Devine, MapReduce in MPI for large-scale graph algorithms, Parallel Computing Journal, Volume 37, pp. 610-632, 2011.

[25] HPC Virtualization, Virtualization in HPC may not always be the answer as Gillian Law discovers, Scientific Computing World, Oct/Nov 2011, pp. 21-24.

[26] Dhiman, G., Marchetti, G., and Rosing, T., vGreen: a system for energy efficient computing in virtualized environments. In Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design (pp. 243-248). ACM.

[27] Novotn M., Job scheduling with the SLURM resource manager, Bachelor Thesis, Masarykova Univerzita.

VITA

Uğur Koçak received his B.Sc. degree in Mechanical Engineering from Istanbul University in 2009. He started to M. Sc. in Computational Science and Engineering department at Istanbul Technical University. He continued his Master of Science in Computer Engineering program at Ozyegin University under under supervision of Dr. Ismail Ari as a research assistant. His research interests include distributed systems, cloud computing, High Performance Computing (HPC) and High Throughput Computing (HTC). Ugur is a proficient Javascript programmer. He is currently working at Hexagon Ortho as a senior software development engineer.