ABSTRACT

KANDULA, SANDEEP REDDY. Phadoop: Power Balancing Cloud-Based Workloads. (Under the direction of Frank Mueller.)

Over the past years, we have seen an increased adoption of cloud computing across enterprises and this trend is predicted to continue. This has lead to a greater concentration of hardware in massive, power-hungry datacenters that form the backbone of a successful cloud service. It is essential for these datacenters to be energy efficient not only to cut down on electricity and hardware maintenance costs but also to be in compliance with environmental regulations. Although there have been tools to control the frequency of a processor to balance the power consumption and performance of a system, there is a dearth of tools to achieve the same by limiting the power consumption directly. The introduction of Intel's Running Average Power Limit (RAPL) has created a new avenue where software agents can work in conjugation with the underlying hardware to dynamically enforce power bounds. Hadoop is a popular map-reduce system that is used by cloud vendors to provide large scale data-intensive and compute-intensive platforms. The Hadoop framework is built around an acyclic data flow model, which makes it unsuitable for iterative applications.

In this work, we investigate whether it is beneficial to use RAPL interfaces to conserve the energy consumption of a CPU in a cloud-based workload without significant loss of performance. We have enhanced the Hadoop framework to support iterative scientific applications. We have designed and implemented *Phadoop*, an enhanced version of Hadoop YARN framework that utilizes RAPL's power capping feature to mitigate computational imbalances in an application and to reduce CPU power consumption. This state-of-the-art framework makes use of a process pool to schedule map/reduce tasks and caches the input data to reduce I/O overhead. Our experimental evaluations show that RAPL interfaces can be used to cap the power consumption of a cloud-based workload. Using a set of synthetic benchmarks, we show that *Phadoop* conserves energy and performs better that Hadoop for iterative applications. To the best of our knowledge, such a power capping framework for cloud-based workloads is unprecedented.

Phadoop: Power Balancing Cloud-Based Workloads

by
Sandeep Reddy Kandula

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2014

APPROVED BY:

William Enck	Steffen Heber

Frank Mueller
Chair of Advisory Committee

DEDICATION

To my wonderful family and friends.

BIOGRAPHY

Sandeep Reddy Kandula was born and raised in Hyderabad, also known as the pearl city of India. He completed his schooling in Hyderabad. He did his undergraduation in Computer Science at BITS-Pilani (Birla Institute of Technology and Science), Rajasthan from 2005-09. Thereafter, he worked at Oracle India Pvt Ltd for three years. To specialize in the field of Computer Science, he joined Masters program at North Carolina State University, Raleigh in August, 2012. During the course of his Masters, he did his internship in Global Support Group at NetApp, Inc. He has been part of the Dr.Frank Mueller's research group since spring 2013 and focuses on job power awareness.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Frank Mueller for his professional guidance and invaluable advice throughout my Masters. His constant flow of ideas and out of box thinking inspired me to deal with many challenging and interesting problems in my research. I would also like to thank Dr. William Enck and Dr. Steffen Heber for agreeing to serve on my thesis committee.

Whatever I am today is only because of my loving parents, caring brother, supportive cousin and my lovely fiancee. None of this would have been possible with out their constant support and encouragement. I would like to thank all my friends who supported me at all times. Especially, my flatmates Mahesh and Kasyap, the ever energetic Bharat Babu Eruvuru, the wanderer Krishna, the sincere Harsha, the newly-wed nano-physicist Sarath, the pharmacist Shailendra, the construction specialist Vidya Sagar, the realtor Pavani and the trader Naveen, who were constantly on my side to give me the energy to pursue this not so long but arduous journey. Last but not least, I would like to thank all my labmates who made my research experience a fun filled journey.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Cloud computing [17] has gained wider traction over the past decade as it saves money and time for customers and also offers flexibility, fault tolerance and security. Salesforce, Amazon, Microsoft, Google and Rackspace are some of the big vendors in today's cloud market that offer software, a platform and infrastructure as a service [31]. Making use of such services removes the burden of maintaining private systems from the users. It proves to be an economical option as the service is shared by multiple users. Also, updates to the hardware are handled by the vendor distributing the overall cost among all the users of the service instead of a single user in case of an in-house system. It is also time-efficient as the vendors can easily adjust the scale of a service on-the-fly in accordance with the requirements of the customer. In order to enable the vendors to provide such services, all the hardware needs to be concentrated at centralized locations called datacenters.

Datacenters are the backbone of a cloud service and efficient datacenter management is the key to provide economical and time-efficient services to users. Although centralized hardware locations enable speedy updates and recovery, they pose the challenge of power supply and cooling requirements to the operators. Power supplied to a datacenter is used not only for the direct operation of compute machines, but a considerable chunk of it is also for running the cooling systems that keeps the thermal conditions at an acceptable operational level. Addressing thermal constraints is important because high temperatures are well known to introduce silent data corruption, which can cause faults in program execution or produce incorrect output without the user's knowledge. Also, operation at higher temperatures shortens the life of hardware components leading to frequent replacements or early retirement, which in turn upsurges the maintenance budget.

Studies by various IT analysts indicate that the power consumption of datacenters across the globe has been rising and this trend is likely to continue due to an increase in demand for compute resources. DatacenterDynamics [9] says that during 2013, datacenters in Middle East

and Africa, Latin America and North America have witnessed a growth of 17.5%, 15.1% and 6.8%, respectively, in the amount of power consumed. European and other Asian datacenter economies are said to have experienced similar growth patterns. This growth propounds the challenge of concentrating more hardware without having to build a new datacenter, a multi-million dollar expenditure. In addition, there has been a steady increase in electricity costs over the years. This increment together with the elevated demand for compute resources has made the electricity bill a significant expense for a datacenter. Due to the above reasons along with heightened environmental scrutiny and regulatory pressure from the local governments, energy efficiency has become one of the key challenges in the successful operation of a datacenter.

Cloud service vendors are continuously on the lookout for reducing the energy consumption of compute resources and for low-power equipment for their datacenters. Energy costs can be reduced by installing energy efficient hardware, equipment with customizable power interfaces or those which can work in conjunction with power management tools. In order to cater to such demand, hardware vendors are investing in developing energy efficient components and mechanisms, and software vendors are backing power efficient coding practices. One such effort is the Running Average Power Limit (RAPL) developed by Intel [28]. RAPL is a hardware implemented mechanism for measuring and capping the power consumed by CPU and DRAM starting with the Intel Sandy-bridge processor family. As CPU and DRAM typically utilize a significant portion of the power consumed by a compute node, these components are responsible for a significant portion of the power consumption in a data center that typically scales to thousands of such compute nodes. As shown in Figure 1.1, power consumption of CPUs and DRAMs accounts for 33% and 30%, respectively, of the peak power usage by one of Google's datacenters. A decrement in the energy utilization of these components shall improve the overall energy efficiency of the datacenter.

Our work analyzes the benefits of using RAPL interfaces for reducing the CPU power consumption while executing a heterogeneous cloud-based workload within permissible time limits. We have considered hadoop YARN [42] for our study as it is a widely used, state-of-the-art framework for executing map-reduce jobs on large clusters.Its widespread adoption over the past years is due to the simplicity of the map-reduce programming model that enables users to implement distributed applications easily.

1.1 Contributions

We have modified the YARN framework to reduce the I/O overhead and the energy consumption associated with running an application without affecting its functionality. From here on, the modified Hadoop YARN framework will be referred to as *Phadoop*. The Phadoop framework utilizes a process pool for executing the map and reduce tasks. This framework supports

caching of input data fetched from files during an initial task and reusing it all along subsequent tasks. This modification mainly benefits iterative map-reduce jobs, which typically spawn new map/reduce tasks and redundantly read the same input data from the underlying filesystem in subsequent iteration. The Phadoop framework also consists of a runtime system called the RAPL service that is capable of capping the power consumption of nodes running map-reduce tasks dynamically. This service determines the power limit for a node based on the data gathered from map-reduce tasks. It utilizes the RAPL interfaces for setting the power caps dynamically. In experiments, we observe up to 34% reduction in the energy consumption of blocked matrix multiplication, sparse matrix multiplication and k-means clustering workloads on Phadoop compared to the original Hadoop. And we observe a performance improvement in the map-reduce tasks in Phadoop compared to Hadoop by more than 50% for the same workloads.

1.2 Organization

The rest of the thesis is structured as follows: Chapter 2 provides an overview of the RAPL interfaces, the map-reduce programming model and the Hadoop YARN architecture. Chapter 3 and Chapter 4 present the design and implementation details of the *Phadoop* framework, respectively. Chapter 5 describes the applications used as workloads and Chapter 6 provides a detailed evaluation of our framework. We review the related work in Chapter 7 and conclude with Chapter 8.

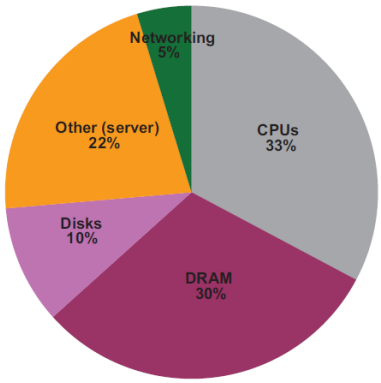


Figure 1.1: Approximate distribution of peak power usage by hardware subsystem in one of Google's datacenters. (reprinted from [10])

1.3 Hypothesis

Caching of inputs in with orchestrated scheduling of tasks generalizes the map-reduce paradigm and optimizes performance for iterative problems.

Power capping mitigates imbalances in a computationally skewed cloud-based workload and reduces its power consumption without affecting its execution functionality

Chapter 2

Background

2.1 RAPL

Running Average Power Limit (RAPL) [28] is a novel interface provided by Intel, starting with Sandy Bridge based Xeon processors, for measuring and capping the power consumption of CPU and DRAM. We have used Intel's RAPL for capping the power consumption of a processor depending on the nature of a workload. RAPL is a hardware-implemented mechanism, and it supports client and server platforms, of which the later is the one we have used for our experiments. The reason for this decision is that most of the machines used in a datacenter belong to the server segment. RAPL groups the components of a system into domains to provide a common interface to manage their power consumption. For the client and the server platforms, RAPL supports a total of four domains, i.e., Package (PKG), Power Plane 0 (PP0), Power Plane 1 (PP1) and DRAM. The PKG domain includes all the components of a processor die. PP1 and PP0 domains consist of processor cores and uncore components, respectively. Lastly, as the domain name suggests, the DRAM domain consists of directly attached memory. The client platform supports PKG, PP0 and PP1 domains whereas the server platform supports PKG, PP0 and DRAM domains. To make use of these interfaces, *msr* and *cpuid* kernel modules are required to be enabled. These kernel modules allow a user with appropriate permissions to read and write the model specific registers (MSRs) corresponding to different power domains.

A series of non-architectural MSRs constitute the RAPL interfaces. MSR_RAPL_POWER_UNIT is a 64-bit register that provides information about the power, energy and time measurement units across all RAPL domains. We have used the default unit increments of 0.125 Watts for power, 15.3 micro-Joules for energy and 976 micro seconds for time. The POWER_LIMIT register per domain allows a user to specify a time window and an average power cap. The hardware ensures that the power bound is met by adjusting P-states [30] of a processor and other undocumented mechanisms. A P-state is a voltage-frequency setting of a processor that

determines its power consumption and speed. ENERGY_STATUS, a read-only register, reports the energy usage of a domain. This register is updated approximately every millisecond. The wrap around time for this register can be around 60 seconds or higher depending on the power consumption. POWER_INFO is another read-only register per domain that provides information about the range of power values and thermal specifications pertaining to a domain. It also reports the maximum possible time window that the RAPL interface can be programmed for. RAPL also provides platform specific PERF_STATUS registers that report the amount of time a domain has spent below the OS-requested P-state.

2.2 Map-Reduce programming model

Map-reduce [20] is a high level programming model that requires a user to define a few functions towards implementing a parallel application. The application is automatically parallelized and executed on a large cluster by a map-reduce framework such as Hadoop [2]. In this programming paradigm, a programmer needs to define a map function and a reduce function, which are invoked during a map-reduce job execution. As indicated by Figure 2.1, a map-reduce job consists of a map phase and a reduce phase. During the map phase, all of the mappers run in parallel to process a corresponding chunk of input files. Each mapper produces intermediate key, value pairs ¡k_{2i}, v_{2i}¿ by applying the user-defined map function on each of the input key, value pairs ¡k_{1i}, v_{1i}¿. This intermediate data is partitioned into chunks that are assigned to a reduce task based on a user-defined partitioning function. During the reduce phase, all reducers run in parallel and apply the user defined reduce function on the corresponding chunk of intermediate data, a key and its corresponding list of values to produce the final key value pairs ¡k_3, v_3¿ as output.

Map-reduce follows the shared-nothing architecture paradigm, where parallel running tasks are independent of each other. Hence, it is more suited for simple and embarrassingly parallel algorithms. Though map-reduce can only be applied to a specific set of problems, this programming model has come to prominence in the field of distributed computing due to its ease of use, scalability and fault tolerance. Programmers without any prior experience with distributed systems can easily utilize large clusters through a map-reduce system. They do not have to worry about the data distribution, task parallelization, communication among the parallel tasks, load balancing or the reliability of the underlying system.

2.3 Hadoop YARN

Hadoop [2] is a map-reduce system capable of running applications on large clusters built from commodity hardware. Yahoo started to work on Hadoop, a sub-project of the widely used text

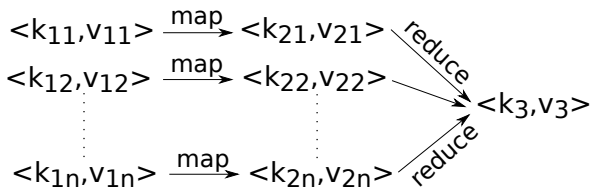


Figure 2.1: Map-reduce programming model.

search library called Apache Lucene, in 2006. In 2008, Apache made Hadoop its top-level project, which confirmed the framework's success. This framework provides fault tolerance and data motion across the cluster. As depicted in Figure 2.2, the Hadoop map-reduce framework consists of four independent entities: a client to submit map-reduce jobs, a job tracker to manage cluster resources and monitor job execution, a task tracker to manage the tasks that constitute a job and the Hadoop Distributed File System (HDFS), which manages data distributed across a network of machines. As this framework has a single entity to handle both resource management and job monitoring, it faces scalability bottlenecks on large clusters. Hadoop YARN [42] (Yet Another Resource negotiator) is an answer to the classic Hadoop map-reduce's scalability problems.

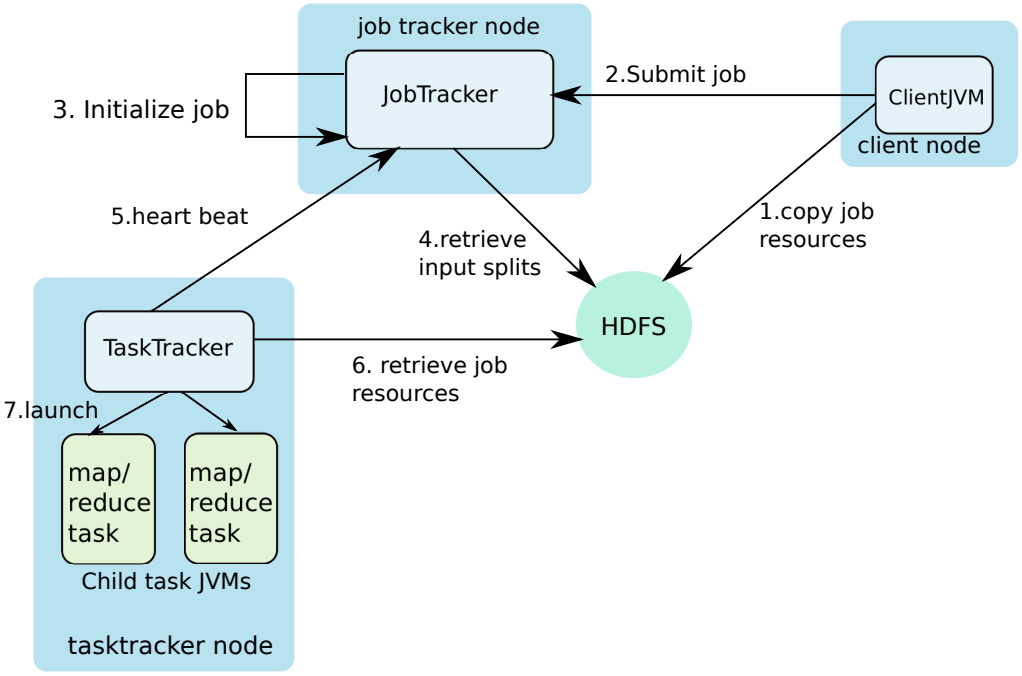


Figure 2.2: Classic Hadoop map-reduce framework.

The YARN framework mainly consists of three daemons: a resource manager to manage the overall cluster resources, an application master to oversee the lifecycle of an application running on the cluster and a node manager to manage the task lifecycle within a node. YARN addresses the scalability issues of its predecessor by splitting up the responsibilities of the job tracker among two entities, i.e., the resource manager and the per job application master. Figure 2.3 depicts the architecture of the Apache YARN framework. An application master (AM) is spawn for every job submitted by the client to the framework for execution. The AM is responsible for negotiating resource allocation with the resource manager for job execution. Once the AM allocates the resources, it forwards container launch requests to the corresponding node manager, which is responsible for launching the containers locally for each of the map/reduce tasks. These map-reduce tasks report their status back to the application master to allow it to keep track of overall job progress.

In the Hadoop map-reduce framework, the map/reduce tasks are launched as separate JVMs primarily for isolating bugs due to user-defined map and reduce functions from the regular execution of the framework. As they are separate processes, output from the map tasks is communicated to the reduce tasks through the underlying Hadoop Distributed File System (HDFS). The map tasks write their output chunks to the HDFS after reading the corresponding input splits. Before the output is written to the file system, it is written to a circular buffer till the contents of the buffer reaches a threshold. Once a threshold is reached, a separate thread spills the data to the disk. Just before writing to the disk, the data is partitioned to determine the reduce task that will handle it. Every time there is a spill, data is written to a new file on the disk. At the end of the map task, when the last map output is written to disk, data from all files created by the spills is merged into a single sorted output file.

Following this, each of the reduce tasks reads the corresponding partition of the map output file. As a reduce task is not necessarily running on the same node where the output file is stored, this data is either read in from file and deposited to the memory of the map-reduce task's JVM or the local disk where the task is running. Similar to the map task, a reduce task utilizes a buffer for reading input. When this buffer reaches a threshold while reading the map output file, spills are written to local disk. Again, multiple files are created due to these spills, which need to be merged at the end of the copy phase. The (key, value) pairs are read from these merged files and the user-defined reduce function is invoked on each of them. The output thus generated is written to the file system by the same function.

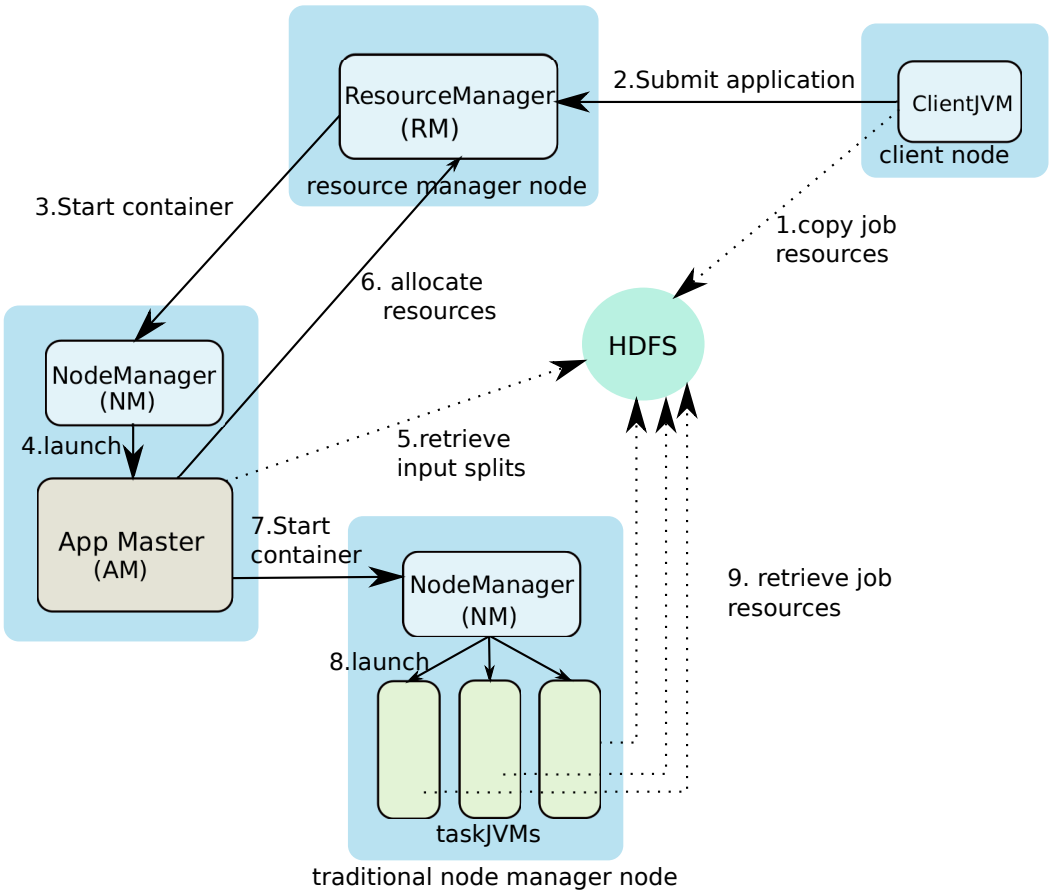


Figure 2.3: Hadoop YARN (map-reduce V2) architecture.

Chapter 3

Power Capping and Data Reuse Design

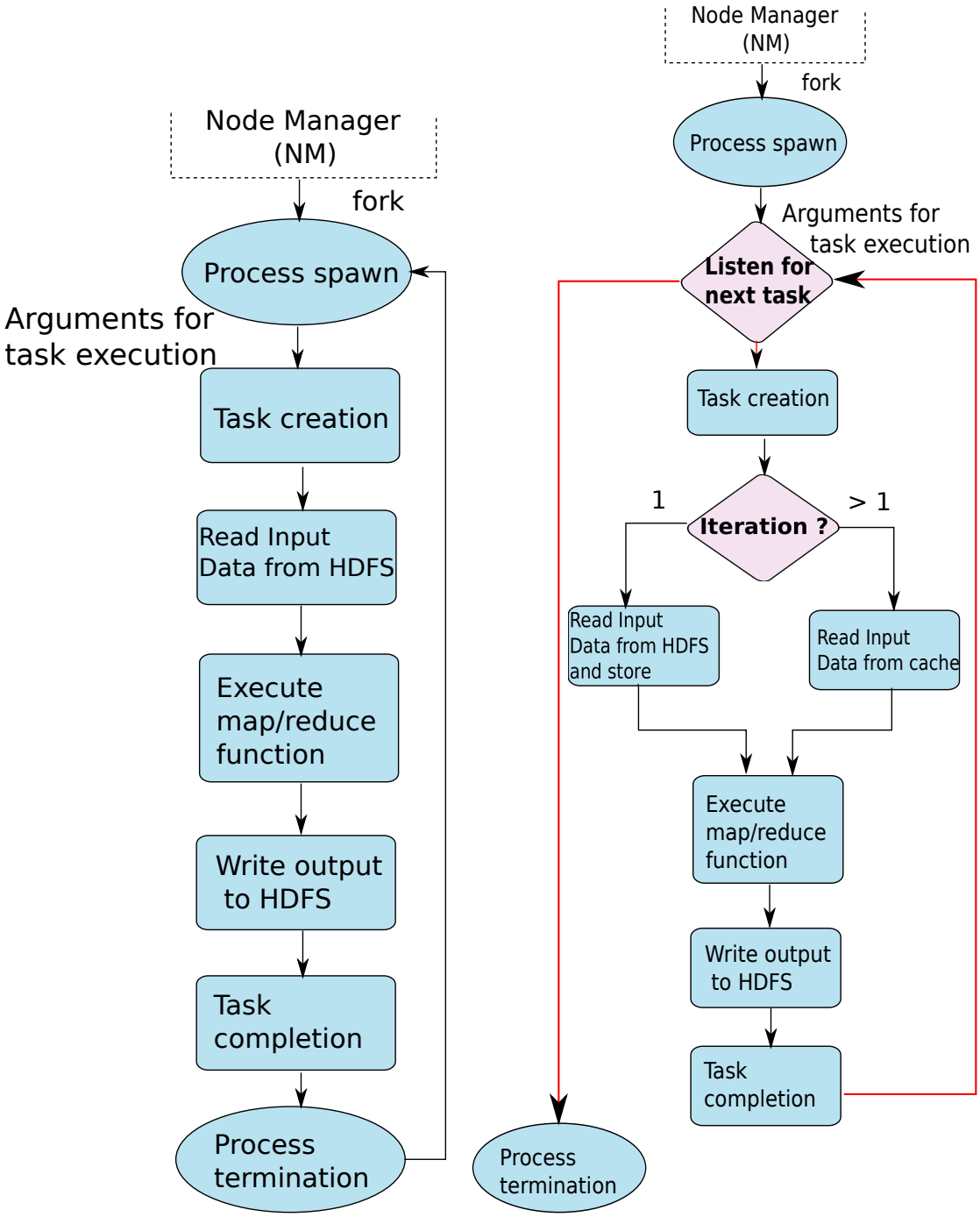
As described in the previous chapters, power capping can be effectively used to lower the power consumption of a framework executing compute intensive workloads. The Apache YARN (Map-reduce V2) framework is primarily designed for handling data intensive applications. We have modified YARN such that the I/O time for compute intensive jobs executed by the framework is reduced. Also, we have added a run-time system to the framework that minimizes the energy consumption without affecting the overall job execution functionality.

As can be observed from the description of the Hadoop YARN framework, an iterative workload involving redundant input reads from the file system suffers from I/O overhead. For instance, if we consider the map-reduce implementation of a parallel sparse matrix multiplication, each task in an iteration performs a redundant operation of reading the same block of input matrix B. This data has to go through the data path indicated above, which includes multiple disk accesses, both local and across the cluster. In order to mitigate this overhead, we have redesigned the Hadoop YARN framework to enable each of the map and reduce tasks to preserve data in-memory after reading it for the first time and to reuse the same across job invocations.

Phadoop uses a process pool to execute the user-supplied map/reduce functions, which enables it to reuse the processes for scheduling map and reduce tasks for subsequent iterations of the job. This avoids a process re-spawn in each iteration and provides a means to reuse the input chunks, read and deposited in-memory during the initial iteration, for task execution in further iterations. Figure 3.1 provides a comparison between the lifecycle of a map-reduce task in both traditional Hadoop and Phadoop frameworks. The state transitions of Figure 3.1(a), which are indicative of the lifecycle of a map/reduce task in the original Hadoop framework, illustrate that a process to run a map/reduce task is re-spawned in every iteration. In contrast, the state

transitions of Figure 3.1(b), which represent the lifecycle of a map/reduce task in Phadoop, show that a task is not re-spawn in every iteration, but is reused across the iterations after being spawned initially. Phadoop provides interfaces to enable the applications to utilize this enhancement and to reduce the overall I/O overhead by caching inputs. Currently, map/reduce tasks in Phadoop are restricted to read data from the cache after initialization. We have not yet generalized the approach to read data either from HDFS or the cache depending on the availability of input data in memory.

Phadoop also consists of a run-time system called the RAPL [28] service, which caps the power consumption of the nodes running map/reduce tasks. This service gathers map/reduce task performance data, determines power caps using this data and enforces power limits dynamically on the nodes executing the child tasks using RAPL interfaces. As indicated by Figure 2.3, an application master (AM) is primarily responsible for negotiating resource allocations with the resource manager for job execution and monitoring the progress of map/reduce tasks launched upon its request by the corresponding node managers. Since an AM collects information about its child tasks, we have integrated the RAPL service into the AM rather than running it as a separate daemon. Iterative map-reduce applications, which conventionally engage in repetitive reads of the same input data across iterations, can benefit from the enhancements in the Phadoop.



(a) Hadoop

(b) Phadoop

Figure 3.1: A comparison between map/reduce task execution states in traditional Hadoop and Phadoop.

Chapter 4

Implementation

To implement the task reuse feature, we have made enhancements to the node manager and the map-reduce client-application modules of the Hadoop YARN framework. Figure 2.3 indicates the section of the YARN framework affected by these enhancements. The node manager controls the launch of containers to execute the tasks. And the map-reduce client-application is responsible for executing child tasks with the arguments supplied by the node manager. During the initial iteration, the container manager spawns the required number of child processes to run the map/reduce tasks. Each of these processes listens on a port, which is computed from the identification number of the respective tasks handled by them, over the loop-back interface of the node. For instance, if a child process is spawn for executing the task with id 2, then this process would listen on port 16002, where 16000 is the base port number for all the processes spawned on a particular node. Once these processes are spawned, the node manager connects to a child process through TCP sockets and sends parameters (host address and port number of the application master and environment variables) required for task execution. On receiving the parameters for task execution, the child process uses them to set/reset its execution environment, which includes the path to the staging area of the current job and job critical information like the location of the security tokens. Once the child process finishes its execution, it reports the status to the application master (similar to the original implementation) and waits for its next assignment, listening over the port indicated during initiation. Each task, during the initial iteration, reads the input data chunks and deposits it in memory using the task context object. Tasks in subsequent iterations handled by the corresponding processes reuse this data and thereby significantly reduce I/O overhead.

The implementation of the RAPL service involved modifications to the application master, child modules and the Umbilical protocol. The Umbilical protocol is used for communication between an application master and its children. Figure 4.1 illustrates the entities involved in the RAPL service and their interactions. A child map/reduce task reports information (host name,

package it is running on and its execution time) to the application master. The RAPL decision engine (RDE) on the application master uses this information combined with a power-saving policy to determine the target execution times for each of the tasks. The RDE also chooses a single task on each package as a Power Limit Enforcer (PLE), which is responsible for setting a power cap for its package. The targets are propagated back to the corresponding child tasks via the umbilical protocol at the start of next iteration. PLE uses the target execution time along with the task's previous execution time to determine a power cap, which is set on the package using RAPL interfaces. Power caps are determined based on the data gathered during inspection runs, which are performed for each package. The inspection runs for each application are performed offline to collect information such as the execution time at different power limit settings.

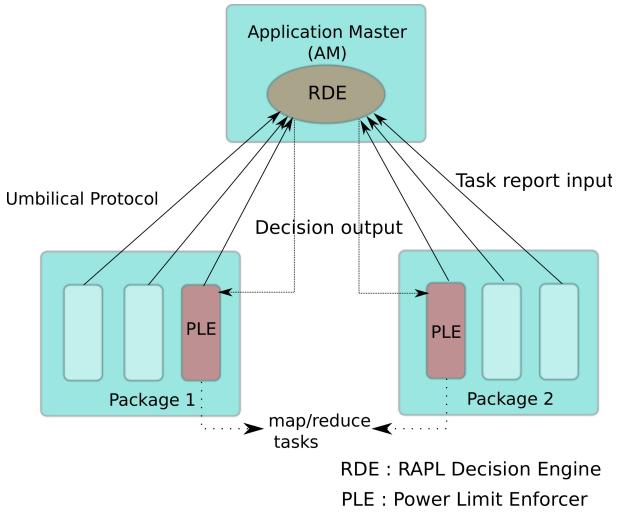


Figure 4.1: An illustration of the RAPL run-time system in Phadoop with its entities and their interaction.

Chapter 5

Applications

While many benchmarks and applications are available for traditional map-reduce frameworks, we decide to implement a set of *iterative* map-reduce applications using the Phadoop framework, which cannot run in traditional map-reduce frameworks as they lack our Phadoop extensions. These iterative workloads consist of a chain of map-reduce jobs. Hence, the reduce phase in each iteration results in a global synchronization as it involves communication among the tasks via HDFS. The following sections describe these applications and the evaluation chapter provides a comparison of their performance and power consumption with the original Hadoop framework.

5.1 Sparse matrix multiplication

Sparse matrix multiplication is a building block for many graph algorithms like graph contraction [26], multiple source breadth-first search [13], recursive all-pair shortest-path [18], and cycle detection [45]. Numerical methods, parsing context-free languages [32] and colored intersection searching [29] are some of the general computing applications of sparse matrix multiplication. Due to its wide range of applications, we have chosen sparse matrix multiplication as one of our workloads.

Our iterative map-reduce implementation of sparse matrix multiplication (ParSpMM) is an adaptation of a scalable, parallel sparse matrix multiplication algorithm (SpGEMM) by Aydin Buluc et al. [14]. In SpGEMM, each of the parallel processes updates a specific block of the resultant product matrix. In contrast, each of the parallel processes is responsible for all multiplications pertaining to a single block of matrix B in ParSpMM.

The following notation is used to describe the implementation and experimental evaluation of sparse matrix multiplication. A, B and C are sparse matrices with the property that $A \in S^{m*l}$ and $B \in S^{l*n}$ and C = A*B, where S is a sparse matrix of the size indicated by the superscript. A_{ij} denotes a block of matrix A with the ith row and jth column. A(i;) and A(;j) denote all the

blocks of matrix A with its ith row and jth column, respectively. The blocking factor for the input matrices is b. $nnz(A)$ denotes the number of non-zero elements in matrix A. P denotes a set of processes running on a processor grid and P_{ij} denotes a process running on the processor in the ith row and jth column of the processor grid. P(i;), P(;j) denote all the processes running on processors in the ith row and the jth column of the processor grid, respectively.

The ParSpMM map-reduce application has two stages. The first stage consists of an iterative computation of partial sums of the product matrix and second stage is an aggregation of these partial sums to compute the final result matrix. Each iteration in the first stage is a map-reduce job that multiplies a block row from matrix A with matrix B. The map function partitions the blocks from input matrices such that a reduce task is assigned a single block from matrix B and a corresponding block from matrix A as required for the computation of the partial product of matrix blocks. As indicated in the Figure 5.1, blocks of matrix A colored red and all blocks of matrix B are involved in a single iteration. During the ith iteration, A(i;) is partitioned by the map task such that all the reduce tasks assigned to B_{kj} (where k = [0, l/b] and j = [0, n/b]) receive the block A_{ik}, i.e., block A_{ij} is broadcast across P(j;). After receiving the assigned blocks of matrices A and B, each reduce task performs the HyperSparseGEMM operation [12] on them and writes the partial result to HDFS. This partial result from all iterations is reduced to the final resultant matrix C during the second stage of ParSpMM.

In this algorithm, the entire matrix B is required to be read from HDFS in every iteration. Phadoop eliminates the need for this redundant operation by utilizing a process pool for executing the reduce tasks and reusing the data pertaining to matrix B instead of reading it from HDFS during the initialization of the task.

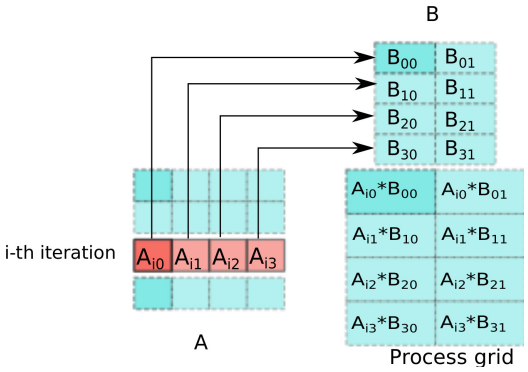


Figure 5.1: A depiction of the operations performed by each process during i-th iteration of the blocked matrix multiplication (A*B)

5.2 Blocked matrix multiplication

Similar to the sparse matrix multiplication, blocked matrix multiplication is used in a variety of applications. It constitutes the core of many scientific computations and is used in computer vision, computer graphics, and internet security applications [41, 44]. The iterative map-reduce implementation of this algorithm is similar to that of ParSpMM described in the previous section. As shown in Figure 5.1, during each iteration of the first stage of this algorithm, a row of blocks from matrix A is multiplied with matrix B by the corresponding reduce tasks to produce a partial result that is later aggregated during the second stage. The difference between this application and ParSpMM is that each reduce task performs a regular matrix multiplication on blocks from matrix A and B instead of the HyperSparseGEMM [12].

5.3 K-means clustering

K-means is a simple and well-known algorithm for cluster analysis in data mining. This algorithm has been successfully employed in a variety of areas such as general statistics, market segmentation, computer vision, geo-statistics, agriculture, biology and astronomy [5, 21]. The parallel map-reduce implementation of this clustering algorithm has become popular in the scientific community with its added ability to handle large data sets while requiring relatively short execution times. Due to its iterative nature along with its popularity in a variety of domains, we have chosen k-means clustering as a workload.

In this work, we have implemented a general k-means algorithm [6] with the map-reduce programming model using the Phadoop framework. The k-means algorithm classifies all the input vectors in an n-dimensional space into a user specified number of groups (referred to as k). It starts with a set of randomly chosen k vectors as centroids and assigns each input vector to its closest centroid. After all the input vectors are classified, new centroids are computed for each group. The classification and re-computation of centroids is performed iteratively until the process satisfies a converge criteria. Each iteration of the k-means clustering is implemented as a map-reduce job. Each of the user-specified number of map tasks reads a chunk of the input vectors along with the centroids either computed during the previous iteration or randomly chosen from input vectors during the initial iteration. It then classifies the input vectors among k groups and computes partial centroids for each group, which are passed on to a reduce task. The reduce task receives the partial centroids from all the map tasks and computes the new centroids, which are written to HDFS. These new centroids serve as an input for the next iteration until the centroids converge.

In this algorithm, map tasks in each iteration read the same input chunk. In Phadoop, this redundant operation of reading the loop invariant data in every iteration is eliminated by using

a process pool for executing the map tasks and caching the input data read during initialization of these map tasks.

Chapter 6

Results

For the performance and power consumption analysis, we have installed the Phadoop and Hadoop frameworks on a RAPL enabled Intel Xeon E5-2667 3.2 GHz server machine. This CPU consists of two packages with 8 cores per package and 16 GB memory. Sparse matrix multiplication, blocked matrix multiplication and k-means are the iterative map-reduce workloads run on this setup to monitor performance and energy consumption for different input loads. For all experiments with Phadoop, the RAPL-runtime uses a power-saving policy that aims at utilizing the slack time of the slowest map/reduce task among all concurrent tasks in an iteration. According to this policy, a task per package with the highest execution time among all the package-local tasks is chosen as a representative task. The execution time of the slowest among these representative tasks is chosen as the target time, which is propagated to all representatives. As described in Chapter 4, the target time is used by the representative tasks in subsequent iterations to enforce power caps on their respective packages.

Calibration

For each application, we have performed inspection runs during which we execute the workload using the same input as an actual run at different power caps and record the power cap to execution time mapping for each map/reduce task. This mapping is provided to the RAPL-runtime system for its decision-making process.

Performance and Power consumption analysis

We have performed experiments using the applications described in Chapter 5 with two types of inputs,

1. *uniform* chunks, and

2. *non-uniform* chunks,

processed in parallel by each map/reduce task of a job. *Non-uniform* chunks tend to result in computational imbalance among concurrent tasks. We have experimented with the Phadoop framework in two different modes,

1. the *power-saving* mode, and

2. the *caching* mode.

The Phadoop framework in *power-saving* mode has both the RAPL-runtime and the caching features enabled. In contrast, in caching mode, RAPL is disabled for Phadoop. In the first set of experiments, all workloads have *uniform* chunks of input data. This allows us to compare individual map/reduce task execution times between the *Phadoop* and *Hadoop* frameworks. Figures 6.1(a), 6.2(a) and 6.3(a) depict the data from these experiments. For these experiments, the Phadoop framework operates in *caching* mode, which allows us to observe the benefits of input caching in Phadoop. Colored bars in these graphs depict the task execution time averaged over all the iterations in a single run, and the error bars on top of the colored bars indicate the range of execution time values for the corresponding tasks. In the second set of experiments, all the workloads have *non-uniform* chunks of input data that tend to create imbalance among the map/reduce tasks executing in parallel on *Phadoop* and *Hadoop*. For these runs, Phadoop operates either in the *power-saving* mode or the *caching* mode. The imbalances across concurrent tasks trigger the RAPL-runtime environment. We have obtained the energy readings of a CPU executing each of the above described workloads on Phadoop and Hadoop. Figures 6.1(b), 6.2(b) and 6.3(b) compare the power consumption of Phadoop operating in *power-saving* mode and Hadoop executing the SpMM, BMM and k-means clustering workloads, respectively. The energy readings reported in this work are calculated using the average power consumption of an application obtained via RAPL power monitoring, and its corresponding total execution time. For all experiments, the task execution times, total job execution times and the energy consumption readings are averaged over 5 runs. We have also shown error bars with minimum and maximum values in all comparison graphs. All applications are run with 8 concurrent map/reduce tasks.

6.1 Sparse matrix multiplication (SpMM)

Figure 6.1 depicts the data gathered from experiments pertaining to the Sparse matrix multiplication workload. This application multiplies two input sparse matrices A and B, which are initialized with seeded random numbers. B is a 2000×2000 matrix and A is a $n \times 2000$ matrix, where n is varied from 1200 to 3200 in steps of 400 for each run. This gradual variation of n

leads to an increase in the number of iterations. Each block of A is of size 400×500 and each block of B is of size 500×1000. The matrices are generated such that the densities, i.e., the number of non-zero elements, of its blocks are in accordance with a uniform or a quadratic distribution. As explained in section 6.1, map tasks in a map-reduce job per iteration are responsible for assigning blocks from input matrices A and B to reduce tasks, which actually perform the multiplication of these blocks in parallel. As a reduce task multiplies a block of B with a corresponding block of A, its computational load is determined by the densities of the blocks that it handles. Since the densities of blocks in one of the matrices are varied while the density of the other matrix is uniform across all its blocks, an imbalance exists among the reduce tasks of a job.

Figure 6.1(a) compares the task execution times of SpMM on Hadoop without caching and Phadoop in *caching* mode. The graph shows the execution time on the y-axis corresponding to each task index on the x-axis. For these experiments, the value of n is set to 2000 and $nnz(B_{ij})$ is a constant for all permissible values of i and j for matrix B with a block size of 500×1000. The bars in this graph show the task execution time averaged over all iterations in a single run, and the error bars on top of the histograms indicate the range of execution time values for the corresponding tasks. The results indicate that the tasks in Phadoop outperform Hadoop by 66% on average. This is due to the fact that reduce tasks of Phadoop cache data from matrix B while the reduce tasks of Hadoop read this data from HDFS repeatedly in every iteration. Across all iterations, a reduce task multiplies a single block B_{ij} of matrix B with all the blocks of matrix A from the block column A(;i). Although the densities of the blocks of matrix A are the same, the distribution of non-zero elements in each of these blocks is different. Due to this reason, the number of integer multiplication operations performed by a reduce task varies from one iteration to the other. This contributes to the variation in the task execution times.

In Figure 6.1(b), the y-axis of the graph shows the average energy consumption and the x-axis depicts the total number of rows of matrix A. For each of these experiments, $nnz(B_{ij}) = x^2$ where $x = 20+(i+2\times j)\times 30$. According to this equation, the density of blocks of matrix B is quadratically varied. But all the blocks of matrix A are of uniform density, i.e., they contain an equal number of non-zero elements. According to our iterative matrix multiplication algorithm described in Section 6.1, an increase of 400 in the number of rows of matrix A with blocks of size 400×500 augments the number of iterations by 1. The increase in the number of iterations with an increase in the number of rows of matrix A explains the increase in the total execution time of the workload across the runs on both Phadoop and Hadoop. We observe that the energy consumption of Phadoop is lower than that of Hadoop for all inputs. The percentage reduction in energy consumption by Phadoop is in the range of 17% to 25%. Figure 6.1(c) depicts the reduction in average power by Phadoop compared to Hadoop, which is in the range of 11% to 30%. As the densities of all blocks of matrix B are varied quadratically, each of the

concurrently executing reduce tasks holds a block of B with a different density. This results in significant computational imbalances among these reduce tasks and triggers the RAPL-runtime capabilities. The RAPL-runtime system applies the power-saving policy described above to reduce the average power consumption of the workload. Figure 6.1(d) shows the percentage decrease in total execution time of a job on the y-axis corresponding to the number of rows of matrix A on the x-axis. If $T_{Phadoop}$ denotes the total job execution time of Phadoop and T_{Hadoop} denotes the total job execution time of Hadoop, then the percentage decrease in the job execution time is given by:

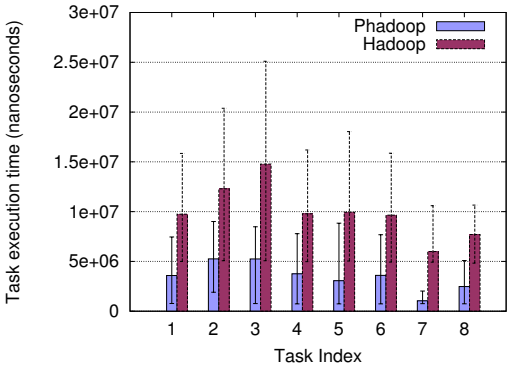
$$\frac{(T_{Hadoop} - T_{Phadoop})}{T_{Hadoop}} \times 100\%$$

A positive value indicates a decrease and a negative value indicates an increase in $T_{Phadoop}$ compared to T_{Hadoop}. A decrease in total execution time for some of the inputs is observed because of caching in Phadoop. Each concurrent reduce task stores a block of matrix B in memory during the initial iteration and reuses it in subsequent iterations. For some of the inputs, the percentage is below 0% indicating a decrease in execution time due to the fact that the overhead due to RAPL-runtime exceeds the benefit due to caching.

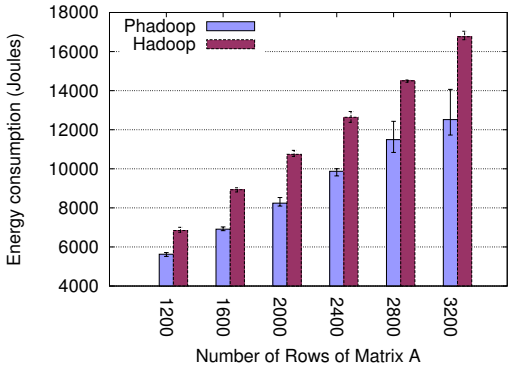
Figure 6.1(e) compares the total job execution times of the workload on Phadoop in *power-saving* mode and Phadoop in *caching* mode. This shows that there is a performance overhead of up to 30% associated with the RAPL-runtime in Phadoop. Although this overhead is diminishingly small for few iterations, it becomes more significant as the number of iterations increases.

6.2 K-means clustering

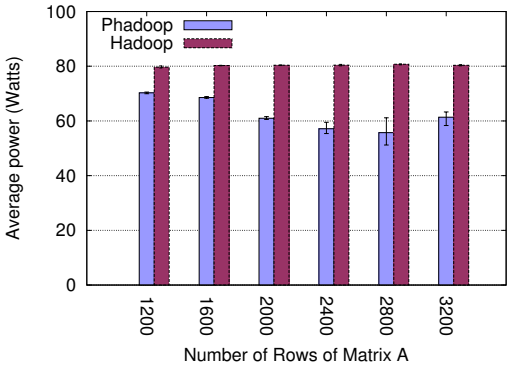
Figure 6.2 depicts the data from the experiments pertaining to the k-means clustering workload. This application requires a set of vectors and centroids as input. We have used seeded random number generators to create the vectors. The initial centroids are randomly selected from these vectors. We divide the n-dimensional vector space into subspaces and generate the input vectors such that the number of vectors across these subspaces follows an exponential distribution. As explained in Section 6.2, each map task of a map-reduce job per iteration processes a chunk of the input vectors. A chunk here consists of all vectors in a particular subspace. As the number of input vectors that each map task processes determines its computational load, varying the vector distribution among the subspaces varies the load handled by each of the concurrent map tasks. For our experiments, we have used 3-dimensional vectors as inputs distributed among 8 subspaces. The input vectors are clustered into 40 groups. Unlike the matrix multiplication inputs where the number of iterations are varied across the runs, the k-means algorithm runs



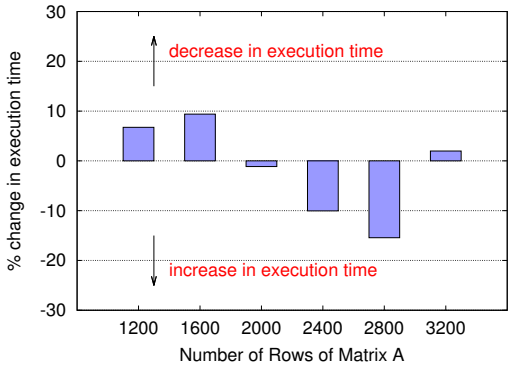
(a) Phadoop (caching) vs. Hadoop



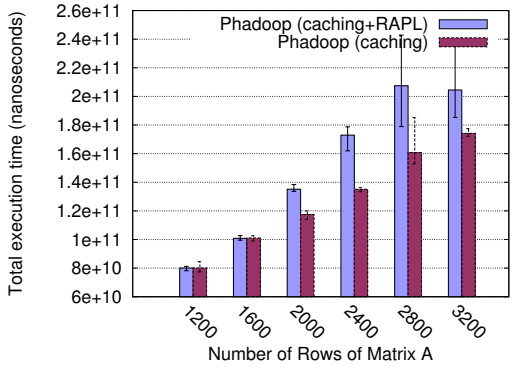
(b) Phadoop (power-saving) vs. Hadoop



(c) Phadoop (power-saving) vs. Hadoop



(d) Phadoop (power-saving) vs. Hadoop



(e) Phadoop (power-saving) vs. Phadoop (caching)

Figure 6.1: Sparse matrix multiplication

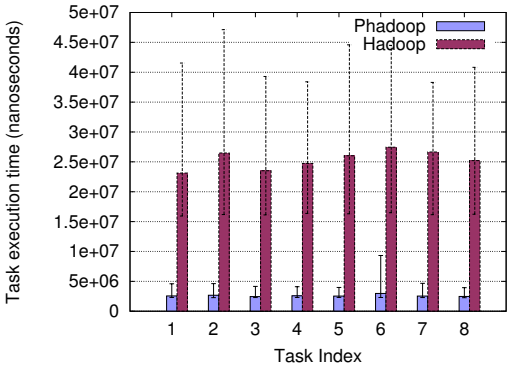
for a maximum of 20 iterations irrespective of the input size.

Figure 6.2(a) compares the task execution times of the k-means clustering workload of Hadoop and Phadoop in *caching* mode. The graph shows the execution time on the y-axis corresponding to each task index on the x-axis. For this run, input vectors are distributed uniformly across all the 8 subspaces. Hence, all concurrent map tasks process an input chunk of same size, i.e., 6144 input vectors. The average task execution time for Phadoop is shorter than that of Hadoop. The results indicate that the tasks in Phadoop outperform Hadoop by 90% on average. For the k-means workload, Phadoop utilizes a process pool for executing the map tasks and caches the input vector data read during the initialization, which is reused during subsequent iterations. This accounts for the shorter average task execution times and also a longer task execution time for the first iteration.

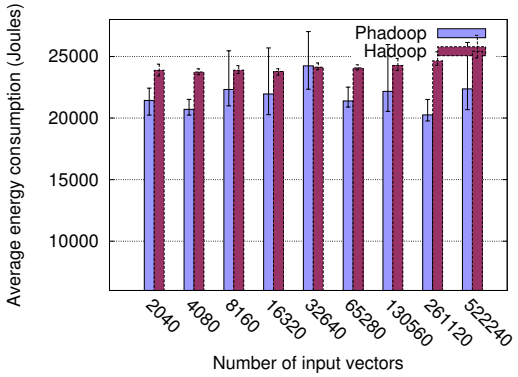
Figure 6.2(b) shows the input sizes on the x-axis and their corresponding energy consumption on the y-axis. For these experiments, the input vectors are distributed non-uniformly across all the subspaces such that each map task receives $n \times 2^i$ vectors, where i is the index of a map task and $n = 4 \times 2^j$ such that j is varied from 1 to 9 across all the runs. We observe that the energy consumption of Phadoop is lower than that of Hadoop for all inputs by a percentage of up to 17%. Figure 6.2(c) depicts the reduction in average power consumption of the workload on Phadoop compared to Hadoop, which is in the range of 18% to 31%. An increase in input size while keeping the number of iterations constant across all runs does not affect the total job execution time. Similar to the above workload, Figure 6.2(d) depicts the percentage decrease in the total job execution time for k-means clustering on Phadoop compared to that of Hadoop. We observe an overall decrease in the performance of Phadoop compared to Hadoop as the overhead due to the RAPL-runtime exceeds the performance benefit of caching. Figure 6.2(e) compares the total job execution times for k-means on Phadoop in *power-saving* mode and Phadoop in *caching* mode. It shows that there is a significant performance overhead of up to 29% associated with the RAPL-runtime in all cases as the number of iterations is the same irrespective of the size of the input processed per run.

6.3 Blocked matrix multiplication (BMM)

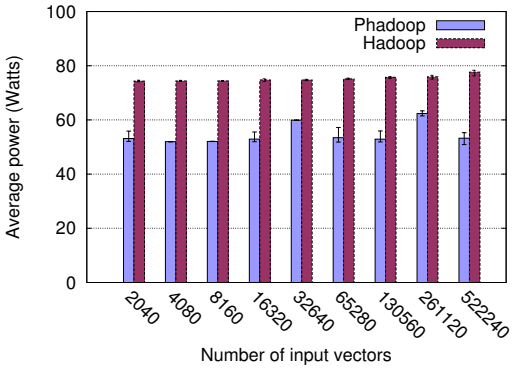
The blocked matrix multiplication requires two regular matrices, A and B, as inputs, which are initialized as seeded random numbers. B is a 1000×500 matrix and A is a $n \times 1000$ matrix, where n is varied from 750 to 2000 in steps of 250 for each run. The block size for both matrices A and B is 250×250. Figure 6.3 depicts the results of the experiments of blocked matrix multiplication for Phadoop and Hadoop. Figure 6.3(a) depicts the caching benefits of Phadoop for this workload. The results indicate that the tasks in Phadoop outperform Hadoop by 53% on average. Similar to the inputs for Sparse matrix multiplication, matrix A consists of blocks



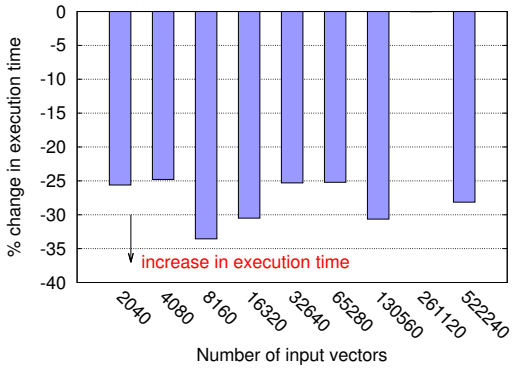
(a) Phadoop (caching) vs. Hadoop



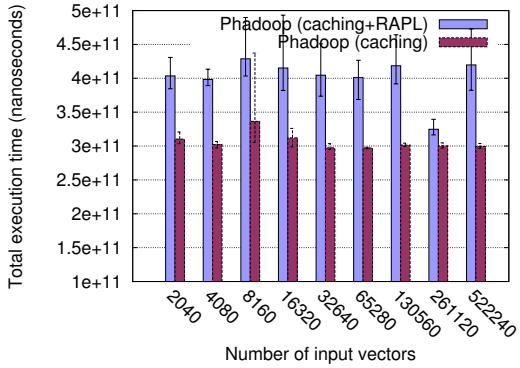
(b) Phadoop (power-saving) vs. Hadoop



(c) Phadoop (power-saving) vs. Hadoop



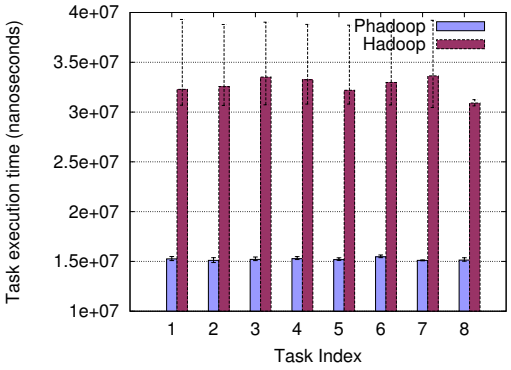
(d) Phadoop (power-saving) vs. Hadoop



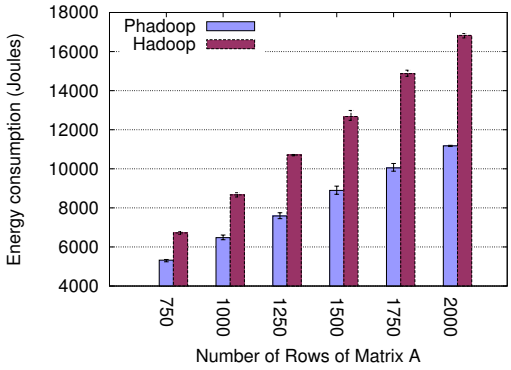
(e) Phadoop (power-saving) vs. Phadoop (caching)

Figure 6.2: K-means clustering

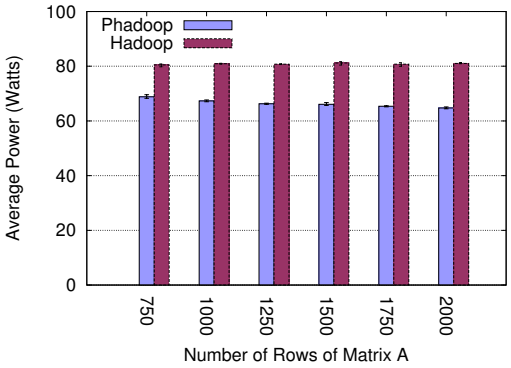
with uniform density and matrix B has blocks with densities that vary quadratically. But the input matrices used here are of smaller dimension and are dense compared to that of sparse matrix multiplication. As the regular matrix multiplication algorithm used by the reduce tasks in the BMM implementation does not perform a multiplication when either of the numbers is 0, the densities of the multiplied blocks determine the computational load of a reduce task. From Figure 6.3(b), we observe that BMM results in a decrease of 25% to 42% in energy consumption on Phadoop compared to Hadoop. Figure 6.3(c) depicts the reduction in average power consumption by Phadoop compared to Hadoop which is in the range of 14% to 20%. Figure 6.3(d) depicts the performance improvement of Phadoop over Hadoop for this workload. The percentage decrease in execution time, calculated as shown in Section 6.1, on the y-axis corresponding to different number of rows of input matrix A on the x-axis is shown in this figure. We observe that the BMM workload has a maximum of 16% performance improvement, which is the highest among all three workloads. Similar to the above two workloads, Figure 6.3(e) shows that there is a significant performance overhead of up to 27% associated with the RAPL-runtime for larger numbers of iterations. The high variation in power consumption of Phadoop for a few runs in case of k-means clustering is due to the power-saving policy used by the RAPL-runtime. The RAPL-runtime system uses a power-saving policy that always tries to conserve energy of a CPU by slowing down the faster running tasks to utilize the slack time introduced by the slowest task running concurrently. If this slowest task was accidentally slowed down due to reasons other than its computational load, the RAPL-runtime may force a greedy slow down of other tasks, which can be counter productive. In a worst case scenario, this can propagate along subsequent iterations and can lead to the lowest possible power cap being set on all packages, which will ultimately slow down the entire job.



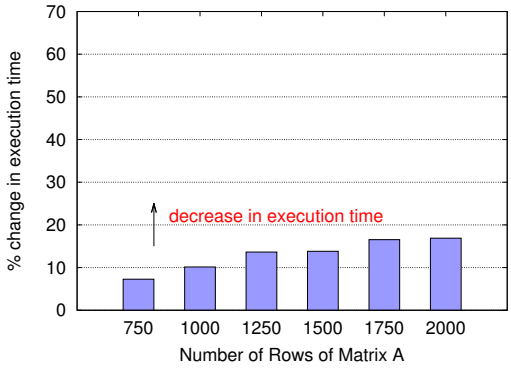
(a) Phadoop (caching) vs. Hadoop



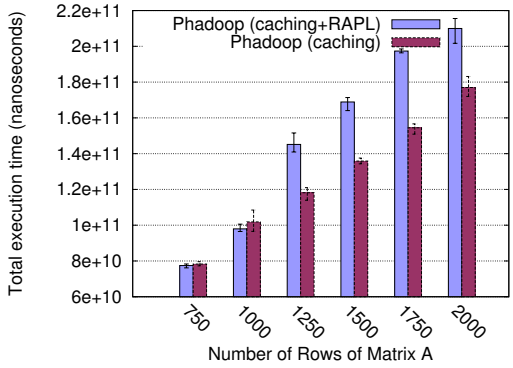
(b) Phadoop (power-saving) vs. Hadoop



(c) Phadoop (power-saving) vs. Hadoop



(d) Phadoop (power-saving) vs. Hadoop



(e) Phadoop (power-saving) vs. Phadoop (caching)

Figure 6.3: Blocked matrix multiplication

Chapter 7

Related work

The simplicity of the map-reduce programming model has caused frameworks such as Hadoop [2], Cassandra [1], Hbase [3], Hive [4], Pig [7] and Spark [8] to be increasingly adopted by cloud vendors to process extremely large amounts of data quickly. On the flip side, the ease of this programming model restricts its application domain. To overcome its shortcomings, there have been many extensions to the Hadoop framework, and new frameworks have been created by both academia and industry [43, 33, 27, 25, 46, 11, 24, 16]. Among these, our work is closely related to CGL-MapReduce [24], Hadoop Spark [46], Twister [25] and HaLoop [11], which support the concept of reusing a working set of data across multiple iterations.

Ekanayake et al. [24, 25] have developed frameworks that support iterative applications. Both their frameworks use map-reduce workers to execute the map/reduce tasks and streaming for all communications. Unlike the distributed file system (HDFS) used by Phadoop, these streaming-based content dissemination networks used for communication between map and reduce tasks are sensitive to failures. HaLoop [11] also supports iterative applications by caching loop invariant data on local disks of slave nodes and schedules the map/reduce tasks that handle the same data across iterations on the same slave nodes. In contrast, our Phadoop framework, which targets compute intensive jobs, uses a process pool and caches the data in-memory to lower data access latency. Spark [46] supports iterative jobs and interactive analytics where map-reduce is deficient. It is based on two main abstractions: resilient distributed datasets (RDD) and parallel operations. RDD is a collection of read-only objects used by Spark to cache data across map-reduce-like task executions. These RDDs can be reconstructed using the lineage data store by the framework in case of a node failure. But it uses a single task to collect results from concurrent reduce task execution, which hampers the scalability of the algorithms.

Several systems and techniques have been developed to conserve energy without significantly increasing execution time. Rountree et al. [35] and Sarood et al. [36, 39, 37] have used DVFS and temperature-aware load balancing to develop systems that can trade execution time for

lower energy consumption. Although these systems can achieve energy savings, DVFS does not guarantee strict bounds on the power consumption of a processor. Dongarra et al. [23], Demmel et al. [22], and Guilherme et al. [15] have used RAPL power monitoring to gather energy readings. Our work utilizes RAPL interfaces to measure the power consumption and enforces power bounds on CPU cores. David et al. [19] use RAPL to measure and limit the power consumption of main memory. Rountree et al. [34] have presented the opportunities of using RAPL to control power consumption of a processor as an alternate to DVFS. They also demonstrate that the manufacturing variations in processors convert to variation in performance under a power bound. Sarood et al. [38] focus primarily on HPC applications while our work focuses on cloud-based workloads. Subramanium et al. [40] use RAPL interfaces to ensure that energy can be controlled in a proportional manner for an enterprise-class server workload, from the SPECpower benchmark. They characterize the power profile of this benchmark within different subsystems using on-chip RAPL power meters. For different load levels, they analyze the power consumption and performance of the benchmark under enforced power bounds using RAPL.

Chapter 8

Conclusion and Future Work

To the best of our knowledge, this work provides the first experimental study to optimize the energy consumption of a cloud-based workload using the RAPL power capping interfaces. This work presents the design, implementation and evaluation of *Phadoop*, an enhanced version of the Hadoop YARN framework. The *Phadoop* framework improves the performance of iterative applications using a process pool for scheduling the map-reduce tasks and caching the loop-invariant data in memory. This state-of-the-art framework uses RAPL interfaces to enforce power bounds. We have evaluated our *Phadoop* framework by comparing its performance and energy consumption with that of a traditional Hadoop framework using sparse matrix multiplication, k-means clustering and blocked matrix multiplication workloads. Experimental results indicate a reduction in energy consumption of Phadoop up to 34% compared to Hadoop. The results also indicate that the map/reduce tasks in Phadoop outperform Hadoop for iterative applications.

In future work, we plan to focus on the following topics:

1. Extend the Phadoop framework to generalize the process pool such that it supports reading data either from HDFS or the cache depending on its availability in memory.

2. Enhance the Phadoop framework to support the usage of a process pool for scheduling map/reduce tasks in a multi-node environment.

3. Enhance the RAPL-runtime system with learning algorithms that will eliminate the need for inspection runs and improve its power cap prediction accuracy.

4. Devise fault-tolerance mechanisms for the Phadoop framework in the presence of input caching.

REFERENCES

[1] Cassandra. http://cassandra.apache.org/.

[2] Hadoop. http://hadoop.apache.org/.

[3] Hbase. http://hbase.apache.org/.

[4] Hive. http://hive.apache.org/.

[5] K-means clustering. http://en.wikipedia.org/wiki/K-means_clustering.

[6] Mahout k-means. https://mahout.apache.org/users/clustering/k-means-clustering.html.

[7] Pig. http://pig.apache.org/.

[8] Spark. http://spark.incubator.apache.org/.

[9] Dcd industry census 2013: Data center power, January 2014.

[10] Luiz Andr Barroso and Urs Hlzle. *The Datacenter as a Computer*. Morgan and Claypool, 2009. An Introduction to the Design of Warehouse-Scale Machines.

[11] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[12] A Buluc and J.R. Gilbert. On the representation and multiplication of hypersparse matrices. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11, April 2008.

[13] Aydin Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011.

[14] Aydin Buluç and John R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *CoRR*, abs/1109.3739, 2011.

[15] Guilherme Cal, Alfredo Gardel, Ignacio Bravo, Pedro Revenga, Josl Lzaro, and F. Javier Toledo-moreo. Power measurement methods for energy efficient applications, 2013.

[16] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[17] Louis Columbus. Gartner predicts infrastructure services will accelerate cloud computing growth. Technical report, Forbes, February 2013.

[18] Paolo D'Alberto and Alexandru Nicolau. R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks.

[19] H. David, E. Gorbatov, Ulf R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194, Aug 2010.

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

[21] Briti Deb and Satish Narayana Srirama. Article: Parallel k-means clustering for gene expression data on snow. *International Journal of Computer Applications*, 71(24):26–30, June 2013. Published by Foundation of Computer Science, New York, USA.

[22] James Demmel, Andrew Gearhart, James Demmel, and Andrew Gearhart. Instrumenting linear algebra energy consumption via on-chip energy counters.

[23] Jack Dongarra, Hatem Ltaief, Piotr Luszczek, and Vincent M. Weaver. Energy footprint of advanced dense numerical linear algebra using tile algorithms on multicore architecture.

[24] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 277–284, Dec 2008.

[25] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[26] John R. Gilbert, Viral B. Shah, and Steve Reinhardt. A Unified Framework for Numerical and Combinatorial Computing. *Computing in Science & Engineering*, 10(2):20–25, 2008.

[27] Rong Gu, Xiaoliang Yang, Jinshuang Yan, Yuanhao Sun, Bing Wang, Chunfeng Yuan, and Yihua Huang. Shadoop: Improving mapreduce performance by optimizing job execution mechanism in hadoop clusters. *J. Parallel Distrib. Comput.*, 74(3):2166–2179, March 2014.

[28] Intel. Intel 64 and ia-32 architectures software developer's manual, volumes 3a and 3b, 2012.

[29] Haim Kaplan, Micha Sharir, and Elad Verbin. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SCG '06, pages 52–60, New York, NY, USA, 2006. ACM.

[30] Taylor Kidd. Power management states: P-states, c-states, and package c-states, April 2014.

[31] Joe Panettieri. Top 100 cloud services providers list 2013: Ranked 10 to 1. Technical report, Talkin' Cloud, July 2013.

[32] Gerald Penn. Efficient transitive closure of sparse matrices over closed semirings. *Theor. Comput. Sci.*, 354(1):72–81, March 2006.

[33] Md Wasi-ur Rahman, Xiaoyi Lu, Nusrat Sharmin Islam, and Dhabaleswar K. (DK) Panda. Homr: A hybrid approach to exploit maximum overlapping in mapreduce over high performance interconnects. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 33–42, New York, NY, USA, 2014. ACM.

[34] B. Rountree, D.H. Ahn, B.R. De Supinski, D.K. Lowenthal, and M. Schulz. Beyond dvfs: A first look at performance under a hardware-enforced power bound. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 947–953, May 2012.

[35] B. Rountree, D.K. Lowenthal, S. Funk, Vincent W. Freeh, B.R. De Supinski, and M. Schulz. Bounding energy consumption in large-scale mpi programs. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–9, Nov 2007.

[36] O. Sarood and L.V. Kale. A 'cool' load balancer for parallel applications. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11, Nov 2011.

[37] O. Sarood and L.V. Kale. Efficient 'cool down' of parallel applications. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 222–231, Sept 2012.

[38] O. Sarood, A Langer, L. Kale, B. Rountree, and B. de Supinski. Optimizing power allocation to cpu and memory subsystems in overprovisioned hpc systems. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, Sept 2013.

[39] Osman Sarood, Phil Miller, Ehsan Totoni, and Laxmikant V. Kal. 'cool' load balancing for high performance computing data centers.

[40] Balaji Subramaniam and Wu chun Feng. Towards energy-proportional computing for enterprise-class server workloads, 2013.

[41] P.M. Vaidya. Speeding-up linear programming using fast matrix multiplication. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 332–337, Oct 1989.

[42] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas

Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[43] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 57:1–57:10, New York, NY, USA, 2011. ACM.

[44] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.**
URL: `http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`.

[45] Raphael Yuster, Uri Zwick, and Raphael Yuster Uri Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming.

[46] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.