Introducing Map-Reduce to High End Computing

Grant Mackey, Saba Sehrish, John Bent, Julio Lopez, Salman Habib, Jun Wang
University of Central Florida, Los Alamos National Lab, Carnegie Melon University
gmackey@cs.ucf.edu ssehrish@cs.ucf.edu johnbent@lanl.gov
jclopez@andrew.cmu.edu habib@lanl.gov jwang@cs.ucf.edu

Abstract

In this work we present an scientific application that has been given a Hadoop MapReduce implementation. We also discuss other scientific fields of supercomputing that could benefit from a MapReduce implementation. We recognize in this work that Hadoop has potential benefit for more applications than simply datamining, but that it is not a panacea for all data intensive applications.

We provide an example of how the halo finding application, when applied to large astrophysics datasets, benefits from the model of the Hadoop architecture. The halo finding application uses a friends of friends algorithm to quickly cluster together large sets of particles to output files which a visualization software can interpret. The current implementation requires that large datasets be moved from storage to computation resources for every simulation of astronomy data. Our Hadoop implementation allows for an in-place halo finding application on the datasets, which removes the time consuming process of tranferring data between resources.

1 Introduction

Recently, petabyte data sets have become frontiers for High End Computing (HEC) applications. These applications generate, and process petabytes of data. The applications come from a diverse range of disciplines such as Cosmology [4], Bioinformatics [9], Earthquake Modeling [8], Data Mining [16], MRI scan data [21], Astronomy data [23], realistic graphic animations [22], etc. We characterize HEC applications on the basis of the following properties, as shown in Figure 1:

1. **Data Access Pattern** - Does the application access a file sequentially or non-sequentially.
2. **Worker Communication** - An application can have *loosely coupled* worker communication where multiple workers doing the same task on different data sets without communicating with each other. In contrast, an application may require all the workers to communicate with each other to synchronize tasks especially when they are processing overlapping data sets, hence called a *tightly coupled* worker communication.
3. **Multi-level** - We define levels of an application as whether a single MapReduce operation can be used to implement an application or if multiple MapReduce operations are needed to meet the application requirements.

In HEC environment, parallel jobs are run in multiple ways. Traditionally, researchers in HEC have utilized the Message Passing Interface (MPI library) [18, 24] and other customized models to implement these demanding scientific applications. A taxonomy of the existing parallel programming models is shown in the Figure 2. MPI and the corresponding I/O interface provides a rich set of functions that are optimized for specific scenarios. MPI/MPI-IO offers collective and non-collective operations to improve communication and I/O performance. MPI provides point-to-point and collective communication options for blocking and non-blocking send and receive operations. MPI-IO provides basic file manipulation operations like open, close, delete, resize, allocate, size, properties, etc. It provides different file view options to support a variety of access patterns. In short, the comprehensive set of MPI functions gives full control to application developers (i.e. scientists) that allows them to manually optimize the application.

However, the MPI processes are not reliable; if one compute node fails the entire application needs to be restarted. There are solutions for a fault tolerant MPI implementation, however, they are applicable only to a specific set of problems [20]. Also, the learning curve of

MPI coupled with the inherent complexity of its code imposes challenge in scientific application development. These limitations and the growing scale of applications in terms of both computation and data require a high level programming language for automatic data and task distribution as well as failure recovery. Programming models should be sufficiently independent of the underlying architecture for portability, yet they should expose common architecture features to enable efficient mapping of the programs onto architectures [15]. Programming models should remain, however, simple for ease of use. For example, machine configuration for parallel jobs should be a transparent factor.

In contrast to MPI, there are customized computing models include submitting a job script manually to a scheduler. The job script consists of multiple serial jobs, written in any language, that are distributed across multiple nodes in the cluster environment by the scheduler. The user is responsible for making the tasks parallel, and then the scheduler just sends the jobs to the available nodes. In the case of failure, the user will be notified of the failure by the job scheduler. However, it is the responsibility of the user to re-instantiate the failed task(s). In short, keeping track of multiple failed tasks in a large application is very cumbersome. These aforementioned approaches have presented scientists and researchers with two major fundamental challenges:

1. Ease of programming
2. High availability and reliability

MapReduce [14] is a new programming abstraction used by Google in its search engines and other data-intensive applications running on clusters. MapReduce is an attempt to ease the programming burden while managing and processing large data sets. MapReduce follows a SIMD or SPMD model as it performs single instruction or all instructions in a single program on multiple large data sets. MapReduce provides automatic failure recovery by restarting the failed tasks. Programmers find MapReduce easy to use, and there are more than ten thousand distinct MapReduce programs implemented at Google [13]. The applications that can benefit from the MapReduce style programming exhibit data parallelism, task parallelism and resilience. Applications like web search engines are data parallel and MapReduce is an ideal candidate to implement them. Contrary to this, partially data parallel applications, such as FLASH [17], uses MPI to parallelize the simulation code for thermonuclear detonations.

Recently, an open source implementation of MapReduce, Hadoop [1], has also been successfully used to program data parallel applications. Conceptually,

Figure 2. Taxonomy of Parallel Programming Models

Hadoop can be used for data intensive scientific applications, because it facilitates ease of programming and high availability and reliability. Currently, whether it is feasible to implement scientific applications with Hadoop is still an open question.

In Section 2 we give a quick introduction of Hadoop including the Map Reduce language as well as Hadoop's distributed file system (HDFS). Section 3 discusses a few potential applications and their suitability to the Hadoop framework. Section 4 describes preliminary results of a scientific Map Reduce application as compared to a current scientific application. Section 5 discusses related and future work, and presents the conclusion.

2 Background of the Hadoop Architecture

The Hadoop architecture consists of a *namenode* and a *jobtracker*, both of which are servers, and then an 'N' amount of servers that function as *tasktrackers* and *datanodes*. The *namenode* is responsible for managing all file system data within the Hadoop file system. It is also responsible for handling all read/write access to files as well as file replication. The *datanodes* service all read/write requests from clients based on direction from the *namenode*. They are also responsible for performing replication tasks and more importantly storing the file system data [10]. These two components comprise the backbone of the Hadoop file system. The *jobtracker* is responsible for handling all jobs submitted by a client application. The *jobtracker* makes all scheduling decisions and is responsible for parallelizing the client application across the cluster. The *jobtracker* is also responsible for task resiliency in the cluster. It monitors all running tasks on the cluster and will kill and restart tasks that fail, hang or otherwise disappear from operation. The *tasktracker* is responsible for running the client application via instructions from the *jobtracker* [10]. The *jobtracker* and *tasktrackers* comprise the architecture for map-reduce programs to run

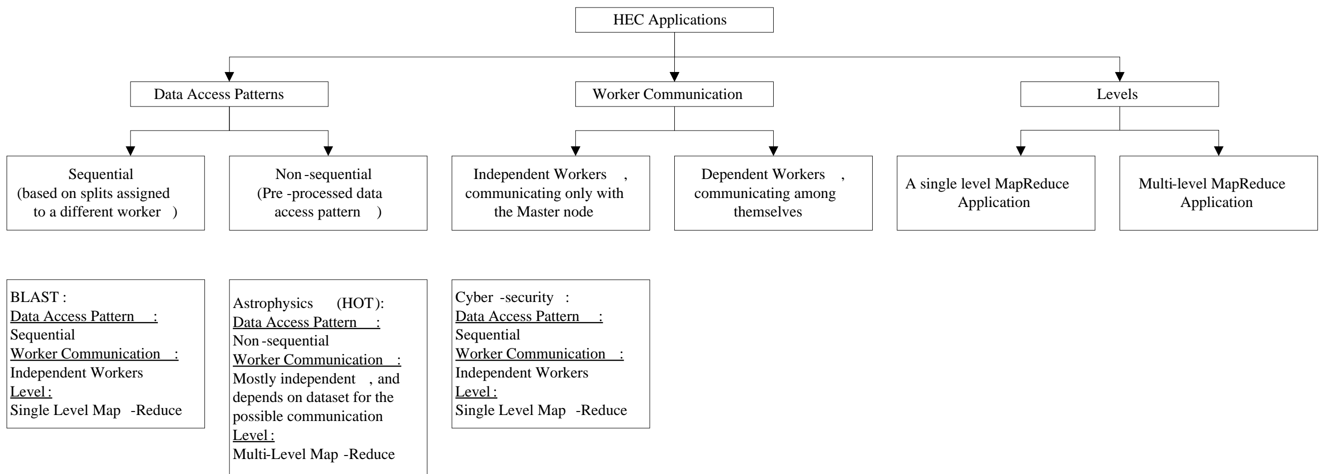


Figure 1. Properties of HEC Applications: Data Access Patterns, Worker Communication, and Application Levels

on.

From the Figure 3, it should be noted that in its configuration, Hadoop prefers that one server be allocated for the *namenode*, one server for the *jobtracker*, and then all other servers be a coupling of a *datanode* and *tasktracker*. This is preferred mainly for data locality. The strength of Hadoop lies in the lack of bandwidth needed for the cluster to function appropriately, thus allowing performance on commodity computing without a fast, expensive interconnect. Because Hadoop keeps all filesystem metadata in main memory, it is necessary for the *namenode* to be its own server, this way file access is not slowed because of strain on the *namenode* from serving data and metadata requests. Similarly, the *jobtracker* is running multiple daemons to ensure task resiliency in the cluster, so Hadoop recommends that this also be its own dedicated server [1].

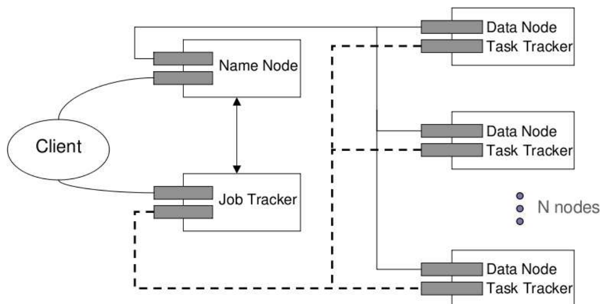


Figure 3. Hadoop Architecture

Beyond the architecture, the Hadoop framework consists of two main components: 1) The Map Reduce language, and 2) Hadoop's Distributed File System (HDFS). These two components working together allow for Hadoop to promise ease of programming and high reliability.

2.1 Map-Reduce

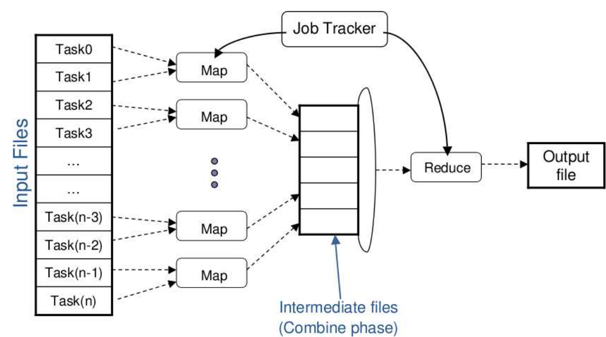


Figure 4. Map Reduce Data Flow

MapReduce is straight-forward in its programming design. The programmer has a map operation in which one parallel operation is performed during the map operation in which results are collected at the intermediate combine phase; and then another operation, reduce, is performed before the output data becomes persistent storage. The MapReduce framework works exclusively on [key,value] ([k,v]) pairs. The map operation is expecting an input of [k,v] and subsequently outputs a set of [k,v] pairs for the reduce phase of the operation [1]. From Figure 4 it is shown that all map and reduce operations are tasks run on the *tasktrackers* in the Hadoop cluster. These individual map and reduce tasks are

monitored from incepetion to completion by the *job-tracker*. During the combine phase of the map reduce operation, intermediate output data from all map tasks on an individual *tasktracker* is written to local storage for the reduce phase. It is possible at this stage to write a combiner operation within the map reduce program to try and sort [k,v] data from the map operation while it is still in main memory [1]. This operation can result in a quick local reduce before the file is passed to a global reduce function. This is a high level view of the steps involved in a map reduce operation. There is no interprocess communication between any map task during the map phase, and likewise no communication between reducers. These two operations, map and reduce, allow for a large parallel dataset to be operated upon very quickly with the assurance of task resiliency.

2.2 HDFS

The HDFS is modeled very closely on the Google file system [12]. The approach to this file system assumes that failure in a large scale computing environment is commonplace, rather than a unique event. HDFS uses a scheme of three way replication to ensure that the files stored are always intact in three seperate places across a Hadoop cluster. This distributed file approach allows for Hadoop to guarantee system resiliency. In Figure 5 is a diagram of a client application accessing the file system. Client applications will first direct file queries to the *namenode*, the *namenode* then directs the file request to the appropriate *datanode(s)* and the *datanodes* supply the client application with the data. What is also shown in Figure 5 is the replication of the file across servers in a rack and across server racks. When Hadoop writes new data to its file system, it tries to apply some amount of data locality if possible. That is, as file chunks are written to *datanodes* across the HDFS, *namenode* tries to group at least one replicated chunk on the same server rack as the primary and then another chunk to an adjacent rack of *datanodes*, while also ensuring that no two replications of a chunk are stored to the same *datanode*. In the event of hardware failure of a server, the *namenode* takes an active role in re-establishing the health of the cluster without need for intervention by the user [1]. The ability of the HDFS to automatically handle system failures without loss of service or need for the user to intervene makes the Hadoop file system a very powerful tool for highly data intensive applications.

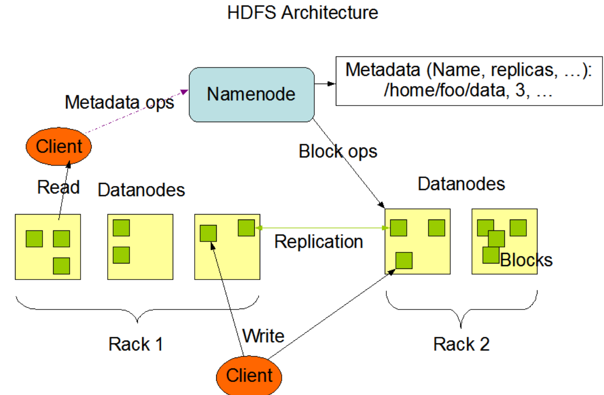


Figure 5. Conceptual Model of the HDFS [12]

3 Applications

MPI Example - Astrophysics (HOT): In the field of cosmology, running simulations are computationally expensive. For example, some algorithms run an FFT-based solver for the Poisson equation [2] and for terabyte or larger data sets, computation can take days [3]. More so than that, the amount of processed data that a simulation generates can take longer to store than the time of calculation. The amount of processed data can scale up to petabytes depending on the simulation. A current practice of theoretical physicists is to simply pull out relevant data from a simulation at run time and immediately discard all other data. This leaves the time consuming need to re-run an old simulation every time a new region of the dataset needs to be examined. One such application that is used in astrophysics/cosmology is called **Hashed Oct-Tree N-body code** (HOT) [4].

The basic operation of the HOT code occurs in several steps which can be adapted nicely to Hadoop. Essentially, particles are domain decomposed into spatial groups. Then, a distributed tree structure is constructed which is traversed independently on each processor in a cluster. The goal of which is to output a high-resolution N-body simulation of the calculated data. In this implementation, simulation sizes are limited to the ability of a cluster to handle the amount of parallel processing and proper storage of the data of these simulations. It has been observed that scientists are simply re running these simulations every time they need to examine a different region of the data set, rather than running the simulation on the entire recorded region and then simply storing it for review when needed. This limitation can be overcome by using Hadoop's ability to store large amounts of data in parallel across a large computing cluster resiliently.

Because of Hadoop's distributed file system, data calculated by a map-reduce version of HOT will be stored on the local node, rather than being shuffled between nodes on a cluster by some management software.

While there is no immediate need to port an implementation of HOT to Hadoop's Map Reduce framework, there is present need for Hadoop's distributed file system(HDFS). The HDFS can be used to analyze the data output by astrophysics applications such as HOT. In large astrophysics simulations, data in the multi-terabyte domain and larger can be generated from computation of raw data from scientific instruments[3]. This data must then be recomputed into comprehensible data for simulation. At Los Alamos National Lab, the current process for astrophysics work follows that raw data is computed into particle position and velocity data (via applications like HOT) on the large supercomputing environments and then written to external storage. When the data needs to be used for visualizations, the large datasets must then be moved back from storage to one of the LANL supercomputers, and then off again when the simulation is over[3]. This whole process is time consuming for the operator to have to move large multi-terabyte datasets back and forth from storage to computing resources everytime a new simulation needs to be run.

Hadoop is a coupling of a highly scalable storage system and a powerful parallel programming language. Using Hadoop as a storage server would allow for in place analysis of the computed vector position data that comes from astrophysics applications. Because the Hadoop file system would allow the data to be analyzed in place on a storage server, once the large scale computation of the raw data has been completed and the particle data is moved to a Hadoop storage implementation, the data will no longer be shuffled between computing and storage resources.

Job Script Example - BLAST: The BLAST application [9] and its different variations are used widely in the field of Bioinformatics as a comparison tool when determining the origin and nature of an organism. Datasets for these operations are large in scale (GBytes). Current methods of processing datasets in Bioinformatics include using what is known as an *array job* [19]. This process entails a scientist taking a file or files to be run by BLAST and splitting it into sub pieces which are submitted to a cluster scheduler and are parallelized across multiple computers running BLAST. At the completion of the BLAST job, the scientist must then take the multiple output files and manually compile them into a single comprehensible output file.

This array job was run on S_suis strain of Streptococcus on a cluster of 36 Pentium III nodes each with 4GB of main memory and the job was submitted via the MOAB scheduler [5]. Total time to completion of the operation was approximately *17 minutes*. This time includes the manual partitioning and reassembling of the S_suis file by the lab worker and it should be noted that there was no failure during the job. The run time of the BLAST application was close to *11 minutes*.

An array job is widely used in Bioinformatics at institutions such as the **Joint Genome Institute** and **Los Alamos National Lab** (LANL). This process is essentially a Map Reduce operation. The difference is that there is no need to manually split an input file or files and reassemble the output, Hadoop does this intrinsically in a map reduce operation. If BLAST were properly ported to the map reduce framework, a user would gain the benefit of no longer needing to manually parallelize the BLAST tasks. With a batch scheduler, no task resilience is guaranteed. It is the responsibility of the user to manage and accommodate process failures in a cluster computing environment. While no task failures occurred on this BLAST array job, failures across large cluster environments are common. Hadoop's architecture guarantees process resiliency.

For BLAST applications, the Hadoop file system would be beneficial solely because of high data reliability and the dependence of MapReduce on the HDFS. The HDFS is responsible for properly handling the parallelization of files. Map reduce is responsible for making the calls to the HDFS, but the HDFS is the backbone of the file splitting process [11].

Cyber-security: LANL uses mined data for real-time network security. When malicious hosts are detected on the network, they are automatically quarantined from the network. Quarantine is accomplished by immediately reconfiguring the Ethernet switch that is attached to the offending devices. LANL's Topology Reconstruction system is responsible for calculating an accurate network map and locating the switch port for each system on the network. The mined data that the lab uses is read in from a streaming binary source into a computer for analysis. These Binary files simply contain multiple network events that need to be identified as either legitimate or harmful network traffic. Since no binary event in the file relates to another, the operation is entirely data parallel on a monthly petabyte scale. LANL is interested in using Hadoop with its cluster computing environment for network analysis [7].

Cyber security's demand for the HDFS is similar to BLAST applications. The Map Reduce language relies on the HDFS in order to function appropriately. The HDFS can handle streaming files into the file system; as well as provide for analysis on the files incoming to

the system. The HDFS allows for an easy partitioning of the incoming binary data to multiple different compute nodes across a Hadoop cluster. Via the HDFS architecture, no streaming data is ever sent through the *namenode* before being transmitted through to the *datanode*. All data read into the HDFS is sent directly to the location it will be stored [11]. Due to this property, network events streaming into the HDFS can be analyzed as they are being written to a *datanode*. In present cluster environments, incoming network data would first be written to a networked file system (NFS) to then be read and processed by the analysis application [6]. This extra overhead provides for an immutable delay in time between receiving and processing live network traffic in an analysis of data where time is critical. Hadoop's ability to do direct data analytics is useful to real-time applications because of the low amount of latency between the time data is received and the time it is processed.

4 Preliminary results of the HDFS in a scientific computing environment

We implement an Astrophysics example as discussed in Section 3 using Hadoop. This application constantly cycles data between the same resources to re-run simulation data as shown in Figure 6(a) and is a time consuming process. Because of this, our group has developed an application for Hadoop to perform in-place friends of friends (FOF) analysis of astrophysics data. As shown in Figure 6(b), once data is written to the HDFS, it does not have to be transfered elsewhere to be analyzed. Our approach to this problem is straightforward and completed in a single level of MapReduce operations. The map operation is responsible for reading binary data from the local file system. The map operation takes the multiple binary files from the local filesystem and distributes them across the HDFS. The map operation sends out the parsed binary file data from the output of the astrophysics data and passes it to the reduce operation. Once the reduce operation begins it runs a friends of friends algorithm. The FOF code generates its output file that the reduce operation then appends to the HDFS This is a similar process to what happens on a large HEC environment, a file is read into the FOF algorithm and an output file is generated. However, the MapReduce operation was completed on relatively inexpensive commodity based hardware and did not suffer an unavoidable loss of productivity from data transfer time from storage to computer resources as did the HEC solution. Nor will the Hadoop version ever have to transfer data across a network to another computing resource to re-run the FOF algorithm, as with the HEC solution shown in Figure 6(a).

Data transfer time is a large overhead to this type of scientific application. 30TB+ datasets are representative of the simulations done at LANL for a FOF code. These datasets do not have the benefit of a high-speed interconnect between storage and computing resources, they must be transferred over the LANL network. This transfer obviously takes a very long time to complete. In theory, if a storage system were able to fully utilize a 10 Gigabit ethernet connection from a storage to computing resource, without a drop in bandwidth, the time to transfer a 30TB file would be approximately 6.826 hours. Figure 7 shows a relation between our Hadoop implementation and the current HEC solution. The first 6.826 hours of transfer time is unavoidable, data must be moved from compute resources to some type of storage, whether it be a storage server or the HDFS. However, after this intial move, our Hadoop implementation gains a time savings of approximately 13 hours without even considering runtime of the FOF calculations. Over the course of several simulations, as indicated in Figure 7, the time savings becomes significant.

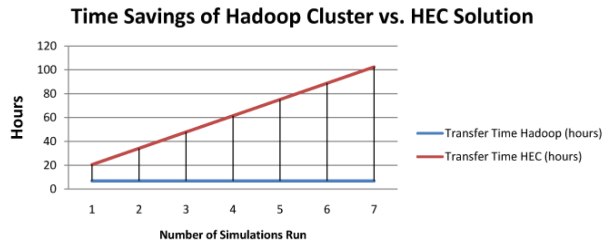
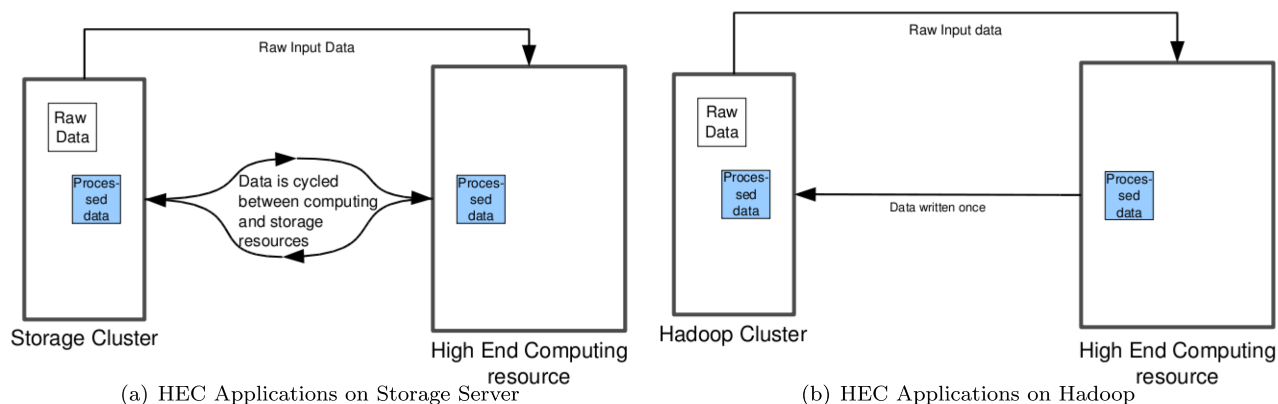


Figure 7. Time to Transfer a 30TB File Across a 10Gigabit Ethernet Connection

5 Conclusion and Future Work

In this work, we have studied a few representative scientific applications in astrophysics and Bioinformatics. We observed that the behavior of these applications is similar to a typical MapReduce application. Most of the scientific applications consist of a simulation phase that generates the data sets, a storage phase to store the results and an analysis phase to analyze these data sets. The data and task parallelism in these applications along with high demands on reliability are motivation to explore migration of the partial or full application to Hadoop. Implementing these applications in Hadoop will alleviate the burden of managing tasks on a cluster computing environment. We have implemented the storage and analysis phase of



(a) HEC Applications on Storage Server (b) HEC Applications on Hadoop

Figure 6. Comparing the HEC Application Implementations on current Storage Systems and Hadoop

an Astrophysics application and our results show that Hadoop enables the in-place analysis of halo data using FOF algorithm, and minimizes the time to move data back and forth between compute and storage resource.

There are many other fields in scientific data intensive computing that could benefit from Hadoop. Fields expressly discussed in this paper, Bioinformatics and Cyber-Security, would be good candidates for Hadoop and Map Reduce. Currently, our group is actively working on improving and broadening the range of Astrophysics applications that would benefit from Hadoop.

6 Acknowledgment

This work is supported in part by the US National Science Foundation under grants CNS-0646910, CNS-0646911, CCF-0621526, CCF-0811413, and the US Department of Energy Early Career Principal Investigator Award DE-FG02-07ER25747.

References

[1] http://hadoop.apache.org/core/.

[2] http://t8web.lanl.gov/people/heitmann/arxiv/codes.html.

[3] http://t8web.lanl.gov/people/salman/.

[4] http://t8web.lanl.gov/people/salman/icp/hot.html.

[5] http://www.clusterresources.com/products/mwm/docs/moabusers.shtml.

[6] http://www.lanl.gov/orgs/hpc/roadrunner/rrinfo/rrm

[7] http://www.woozle.org/ mfisk/.

[8] Vokan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez, Omar Ghattas, Eui Joong Kim, Julio Lopez, David O'Hallaron, Tiankai Tu, and John Urbanic. High resolution forward and inverse earthquake modeling on terascale computers. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 52, Washington, DC, USA, 2003. IEEE Computer Society.

[9] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[10] Apache. *Introduction to Hadoop*, 2008.

[11] Dhruba Borthaku. The hadoop distributed file system: Architecture and design, 2007.

[12] Devaraj Das. Meet hadoop! open source grid computing.

[13] Jeffrey Dean. Experiences with mapreduce, an abstraction for large-scale computation. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–1, New York, NY, USA, 2006. ACM.

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

[15] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: A NPB experimental study study. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2002.

[16] Tapio Elomaa, Heikki Mannila, and Hannu Toivonen, editors. *Principles of Data Mining and Knowledge Discovery, 6th European Conference, PKDD 2002, Helsinki, Finland, August 19-23, 2002, Proceedings*, volume 2431 of *Lecture Notes in Computer Science*. Springer, 2002.

[17] B. Fryxell, A. C. Calder, L. J. Dursi, D. Q. Lamb, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, J. W. Truran, H. M. Tufo, and M. Zingale. Adaptive mesh simulations of astrophysical detonations using the ASCI flash code. In *American Institute of Physics Conference Series*, volume 583 of *American Institute of Physics Conference Series*, pages 223–225, August 2001.

[18] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[19] E. Ruben S.M. Ignacio M.L. Huedo. Adaptive grid scheduling of a high-throughput bioinformatics application. *Parallel Processing and Applied Mathematics.*, 2004.

[20] Ewing Lusk. Fault tolerance in MPI programs. *Special issue of the Journal High Performance Computing Applications (IJHPCA*, 18:363–372, 2002.

[21] Tom M. Mitchell, Rebecca Hutchinson, Radu S. Niculescu, Francisco Pereira, Xuerui Wang, Marcel Just, and Sharlene Newman. Learning to decode cognitive states from brain images. *Mach. Learn.*, 57(1-2):145–175, 2004.

[22] Liu Ren, Gregory Shakhnarovich, Jessica K. Hodgins, Hanspeter Pfister, and Paul Viola. Learning silhouette features for control of human motion. *ACM Trans. Graph.*, 24(4):1303–1331, 2005.

[23] Alexander S. Szalay, Peter Z. Kunszt, Ani Thakar, Jim Gray, Don Slutz, and Robert J. Brunner. Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey. *SIGMOD Rec.*, 29(2):451–462, 2000.

[24] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, New York, NY, USA, 1999. ACM.