Multicore-Enabled Smart Storage for Clusters

Zhiyang Ding†, Xunfei Jiang†, Shu Yin§, Xiao Qin†∓, and Kai-Hsiung Chang†
Department of Computer Science and Software Engineering
†Auburn University, Auburn, AL 36849-5347
∓Email: xqin@auburn.edu
§Hunan University, Hunan, P.R. China

Xiaojun Ruan‡, Mohammed I. Alghamdi∗
Department of Computer Science
‡West Chester University of Pennsylvania, West Chester, PA 19383
∗Al-Baha University, Kingdom of Saudi Arabia
Email: mialmushilah@bu.edu.sa

Meikang Qiu
Department of Electrical and Computer Engineering
University of Kentucky, Lexington, KY 40506-0046
Email: mqiu@engr.uky.edu

***Abstract*—We present a multicore-enabled smart storage for clusters in general and MapReduce clusters in particular. The goal of this research is to improve performance of data-intensive parallel applications on clusters by offloading data processing to multicore processors in storage nodes. Compared with traditional storage devices, next-generation disks will have computing capability to reduce computational load of host processors or CPUs. With the advance of processor and memory technologies, smart storage systems are promising devices to perform complex on-disk operations. The proposed smart storage system can avoid moving a huge amount of data back and forth between storage nodes and computing nodes in a cluster. To enhance the performance of data-intensive applications, we have designed a smart storage system called Multicore-enabled Smart Storage (McSD), in which a multicore processor is integrated in storage nodes. We have implemented a programming framework for data-intensive applications running on a computing system coupled with McSD. The programming framework aims at balancing load between computing nodes and multicore-enabled smart storage nodes. To fully utilize multicore processors in smart storage nodes, we have implemented the MapReduce model for McSDs to handle parallel computing on a cluster. A prototype of McSD has been implemented in a cluster connected by Gigabit Ethernet. Experimental results show that McSD can significantly reduce the execution times of three real-world applications - word count, string matching, and matrix multiplication. We demonstrate that the integration of multicore-enabled smart storage with MapReduce clusters is a promising approach to improving overall performance of data-intensive applications on clusters.**

I. Introduction

Since large-scale and data-intensive applications have been widely deployed, there is a growing demand for high-performance storage systems to support data-intensive applications. Compared with traditional storage systems, next-generation data storage will embrace computing capacity to reduce computational load of host processors in computing nodes. Existing hard disks have limited processing power, which can only be used to handle on-disk scheduling and physical resource management. With the advance of processor and memory technologies, future smart storage systems are promising devices to perform complex on-disk operations [18].

Smart disks (*a.k.a.*, active disks), in which processors are embedded, leverage computational power available in commodity disk drives to deal with application-level processing [15]. Recently, smart disks coupled with embedded processors have been proposed to address the needs of on-drive-data-intensive workloads. In the past, single-core processors were integrated to smart disks to improve I/O performance of data-intensive applications by manipulating data directly on the disks [6][8][10][18]. Smart disks avoid moving data back and forth between storage devices and host processors. Since there is no smart disk product on the market, we decided to investigate smart storage nodes rather than smart disks in this study.

For the past two decades, hardware designers have used the rapidly increasing transistor speed made possible by silicon technology advances to double performance every 18 months [2]. Unfortunately, approaches to increasing transistor count and clock cycle crashed into the power wall recently; increasing transistor count hits the power limit that a chip is able to dissipate. The industry decided to replace single power-inefficient processors with multi-core processors. There is an increasing need to integrate multi-core processors with devices and peripherals. Thanks to the escalating manufacturing technology, it is feasible to embed multi-core processors into storage nodes for high-performance computing. These demand and trend motivate us to develop a programming framework and a prototype for Multicore-enabled smart storage called McSD for clusters in general and MapReduce clusters in particular.

Architecture of energy-efficient processors on a single chip does not necessary guarantee high performance. Thus, improving utilization of advanced multi-core processors has been a thorny subject in computer systems research. Phoenix–an implementation of the MapReduce model–automatically manages thread creation, dynamic task scheduling, data partitioning, and fault tolerance in multicore processor systems [13]. Phoenix includes a programming API and an efficient runtime system. Phoenix allows programmers to write functional-style code that improves the utilization of multicore processors by automatically parallelizing and scheduling. To fully utilize multi-core processors in McSD, we incorporated Phoenix into multicore-embedded smart disks. Note that MapReduce is Google's programming model for scalable parallel data processing [5]. In addition to Phoenix, there exist a wide range of MapReduce implementations tailored for various computing platforms [1][7][14].

The difference between our McSD approach and conven-

978-0-7695-4807-4/12 $26.00 © 2012 IEEE
DOI 10.1109/CLUSTER.2012.70

tional smart disks is two-fold. First, the focus of McSD is smart storage nodes rather than smart disks. Second, the goal of McSD is to take performance benefits of multi-core processors not single-core processors embedded in storage nodes.

Six New Features. When architecting a McSD smart storage system in a high-performance cluster, the following six features will be implemented:

- A two-layer cluster computing architecture contains host computing nodes and smart storage nodes.
- Improved I/O performance is achieved by combining processing capabilities of both computing and storage nodes.
- The McSD storage system allows programmers to write MapReduce-like code that can automatically offload data-intensive computation to smart storage.
- Smart storage nodes can communicate with their host computing nodes via a storage interface.
- A programming framework of McSDs allows smart storage nodes to take full advantages from multi-core processors in the storage nodes.
- The APIs and runtime environment in our McSD programming framework automatically handles computation offload, data partitioning, and load balancing.

We will show how to use the McSD programming framework to implement a few real world applications like word-count, string matching, and matrix multiplication.

Main Contributions. In summary, the four major contributions of this study are:

- A prototype of next-generation multicore-enabled smart data storage.
- A programming framework, which include MapReduce-like programming APIs and a runtime environment for multicore-based smart storage in the context of clusters.
- Development of three benchmark applications to test McSD for clusters.
- Single-application experimental results and multiple-application performance evaluation.

In the following Section II, we review the background information and previous related research that motivate and inspire this study. In Section III, we describe the design issues of the prototype of multicore-enabled smart storage. Section IV presents implementation details of the McSD runtime environment and McSD programming APIs. Experiment results and performance evaluation are discussed in Section V. Finally, Section VI concludes the paper with future research directions.

II. Smart Storage and MapReduce

An increasing number of large-scale data-intensive applications impose performance demands on storage systems, which are the performance bottleneck of various computing platforms. One way to boost I/O performance is to embed processors into hard disk drives, thereby offloading data-intensive computation from host CPUs to hard disks. While the processing capacity in today's disk drives is to manage on-disk scheduling and resource management, future disks can be equipped with dedicated processors to perform complicated data-intensive operations. We call disk drives in which processors are incorporated as smart disks or active disks.

A. From Smart Disks to Smart Storage

Five primary factors have catalyzed the evolution of storage architectures: I/O-bound workloads, improved disk drive attachment technologies, increased on-drive transistor density, emergence of new interconnects, and the cost of storage systems [6]. Existing smart disk prototypes consist of, from hardware perspective, an embedded processor, a disk controller, on-disk memory, local disk space, and a network interface controller (NIC). From software perspective, a smart disk is comprised of an embedded operating system, a database engine, programming APIs and the like.

Unlike stand-alone PCs, smart disks do not contain I/O components such as keyboard and display. Smart drives may directly connect to their host processors through NICs. In our prototype, smart disks or smart storage nodes are connected to host CPUs using a file-alternation-monitor mechanism, which allows a smart storage node to communicate with its host computing node without relying on a keyboard and display unit.

Single-core embedded smart disks have been implemented in various forms. For example, in an active disk model proposed by Uysal *et al.* , on-disk application processing becomes possible because of large on-disk memory [18]. Another active disk model designed by Mustafa *et al.* largely relies on stream-based programming, in which host-resident code interacts with disk-resident code using streams and a code-partitioning scheme [11]. Memik *et al.* developed a smart disks architecture, where the operation bundling concept was introduced to further optimize database query executions [10]. Chiu *et al.* investigated a fully distributed processor-embedded distributed smart disks [4].

The term smart disk may be used interchangeably with other terms such as intelligent disk (IDISK) [9], SmartSTOR [8], processor-embedded disks [4], and semantically smart disks [17]. Note that IDISK uses on-disk integrated processor-in-memory to exploit emerging VLSI technologies [9]; SmartSTOR [8] consists of a processing unit coupled to one or more disks [8]; and semantically smart disks [17] contains various high-level functionalities.

Due to a combination of reasons, no smart disk product is available on the market. If one has to investigate any research issues of smart disks, he/he has to simulate smart disks. We aimed at implementing a smart storage system for clusters and; therefore, we decided to focus on smart storage nodes rather than smart disks in this project.

B. Parallel Programming

The IT industry has improved the cost-performance of sequential computing by about 100 billion times over the past 60 years [12]. A high transistor density in multi-core

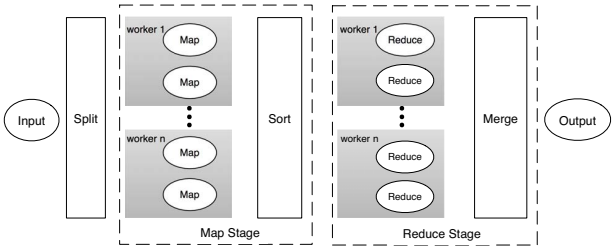


Fig. 1. The work flow of MapReduce.

processors does not always guarantee great practical performance on applications due to the lack of parallelism. There are cases where a roughly 45% increase in processor transistors have translated to roughly 10-20% increase in processing power [16]. Therefore, an open issue addressed in this study is how to enable data-intensive application to exploit parallelism in smart disks coupled with embedded multi-core processors. We believe that well-designed parallel programming APIs must be implemented for future smart disks that can benefit from multi-core processors.

OpenMP applications normally are non-trial because OpenMP only provides low-level APIs. Another diffierence between MapReduce and this model is the hardware for which each of this platform has been designed. MapReduce is supposed to work on commodity hardware, while interfaces such as OpenMP are only efficient in shared-memory multiprocessor platforms.

C. The MapReduce Programming Model

MapReduce is a programming model developed by Google for simplified data processing in data-intensive applications running on large clusters [5]. Map and Reduce - two primitives in MapReduce - are brought from the idea of functional testing. When the Map function is called, user's input data is partitioned into M pieces, which is processed by one copy of the program on each node in a cluster. One of the program copy becomes a master program managing the entire execution. In each MapReduce program, the Map function first takes an input data specified by the users, and outputs a list of intermediate key/value pairs *(key, value)*. Then, the Reduce function takes all intermediate values associated with the same key and produces a list of result key/value pairs. The Reduce function typically performs some kind of merging operation. Finally, the output pairs are sorted by their key value.

Fig. 1 illustrates the work flow of the MapReduce model. The main benefit MapReduce lies in its simplicity. Programmers only provide a simple description of an algorithm that focuses on functionality and leaves actual parallelization and concurrency management to a MapReduce runtime system.

Like the Google file system, MapReduce is not open source software. Google only describes the MapReduce idea without implementation details. Thus, various open source implementations of MapReduce are available for different computing platforms like clusters [1], multi-core systems [13], multiprocessor systems [13], and graphics processing units (GPU) [7].

Haddoop [1]. Hadoop is a Java software framework implemented by Apache to support data-intensive distributed applications. Inspired by Google's MapReduce and the Google file system, the Hadoop project created its own versions of MapReduce and Hadoop Distributed File System. Hadoop applications can be deployed easily by configuring some variables–some paths and nodes; Hadoop defines one master node that manages all systems and jobs, and other worker nodes. Hadoop is so far the most complete quasi-open-source version of MapReduce in the cluster arena. **Mars [7].**MapReduce has been implemented on NVIDIA GPUs using CUDA in the Mars project. Mars takes advantage of GPUs by using the capability of massive threading. In the Mars implementation, there are a large number of physically-collocated mappers and reducers run in multiple threads. The functions in Mars are classified in two categories: runtime system functions and user define functions. Our McSD is different from Mars in the sense that McSD automatically coordinates data-processing activities between host CPUs and multi-core processors embedded in smart disks. **Phoenix [13].** Phoenix is an implementation of Google's MapReduce for shared-memory multi-core and multi-processor systems. There are two categories of functions in Phoenix: the first group is provided by the Phoenix runtime environment; the second one is defined by programmers. Ranger *et .al* concluded that Phoenix is a promising MapReduce implementation for scalable performance on multi-core and multi-processor systems. Different from other MapReduce implementations on clusters, Phoenix is independent of any parallelizing compiler. In our study, we evaluated the suitability of the Phoenix implementation for multicore-embedded smart disks.

D. Integrating MapReduce with McSD

OpenMP and MapReduce are widely deployed parallel programming models for shared-memory systems supporting many large-scale parallel applications [3]. Compared with MapReduce, OpenMP is more generic and provides flexible solutions for a wider variety of parallel computing problems. When it comes to data-intensive applications, MapReduce is normally better than OpenMP. Since the goal of this study is to improve performance for data-intensive applications, we seamlessly integrated smart storage or McSD with the MapReduce framework.

III. Design Issues

A growing number of data-intensive applications coupled with advances in processors indicate that it is efficient, profitable, and feasible to offload data-intensive computations from CPUs to hard disks [15]. Our preliminary results show that we can improve performance of cluster computing applications by offloading computations from computing nodes to storage nodes. To improve the performance of large data-intensive applications, we designed McSD - a prototype of multicore-enabled smart data storage. Different from the existing smart-

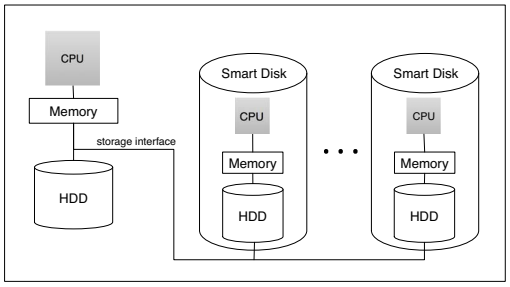


Fig. 2. McSD - The prototype of multicore-enabled smart storage. Each smart storage node in the prototype contains memory, a SATA disk drive, and a multicore-processor.

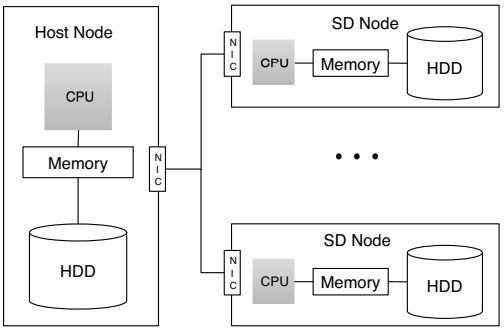


Fig. 3. A testbed for the McSD prototype. A host computing node and an McSD storage node are connected via a fast Ethernet switch. The host node can access the disk drives in McSD through the networked file system or NFS.

disk solutions, McSD not only addresses the performance needs of data-intensive applications using multi-core processors, but also focus on smart storage nodes rather than smart disks.

Fig. 2 depicts the McSD prototype, where each smart storage node contains a multicore-processor, memory, and a SATA disk drive.

A. Design of the McSD Prototype

In our McSD prototype, we integrate multi-core processor, a disk controller, main memory, a local disk drive, and a network interface controller (NIC) into a smart storage node. The storage interface of existing smart-disk prototypes (see Section II-A for details on existing smart disks) is not well designed, because the existing prototypes simply represented a case where host CPUs and embedded processors are coordinated through the network interfaces or NICs in smart disks. To fully utilize the storage-interface in smart data storage, we designed a communication mechanism similar to the file alteration monitor. In our McSD prototype, a host computing node communicates with a disk drive in McSD via its storage interface rather than the NIC. In doing so, we made smart disk prototypes cost-effective since no NIC is needed. Without using NICs, the McSD prototype can adequately represent all the important features of our proposed smart data storage. The design details are described in the following two subsections.

B. A Testbed for McSD

Recall that although a few smart disk prototypes have been developed, there is no off-the-shelf commodity smart disks. Instead of simulating a smart storage system, we built a testbed for the McSD prototype. Fig. 3 briefly outlines the McSD testbed, where two computers are connected through the fast Ethernet. The first computer in the testbed plays the role of host computing node, whereas the second one performs as the McSD smart storage node. The host computing node can access the disks in the McSD node through the networked file system or NFS, which allows a client computer to access files on a remote server over a network interconnect. In our testbed the host computing node is the client computer; the McSD node is configured as an NFS server. We chose to use NFS

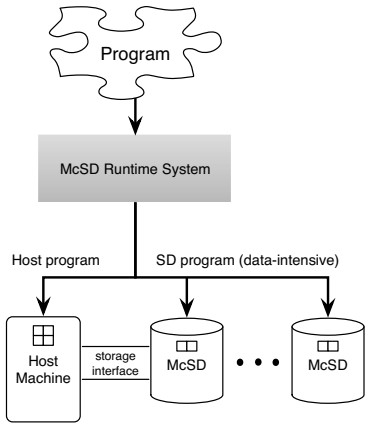


Fig. 4. The programming framework for a host computing node supported by a McSD smart storage node.

as an efficient means of connecting the host computing node and the smart storage node, because data transfers between the host and smart storage nodes are handled by NFS.

We run three real-world applications as benchmarks on this testbed to evaluate the performance of the McSD prototype. The benchmarks considered in our experiments (see Section V) include word count, string matching, and matrix-multiplication.

IV. Implementation Details

A. Implementation of smartFAM

Fig. 5 illustrates the implementation of smartFAM - an invocation mechanism that enables a host computing node to trigger data-intensive processing modules in a McSD storage node. smartFAM mainly contains two components: (1) the inotify program - a Linux kernel subsystem that provides file system event notification; and (2) a daemon program that invokes on-node data-intensive operations or modules.

To make our McSD prototype closely resemble future multicore-enabled smart storage, we connected the host node with the McSD smart-storage node using the Linux network

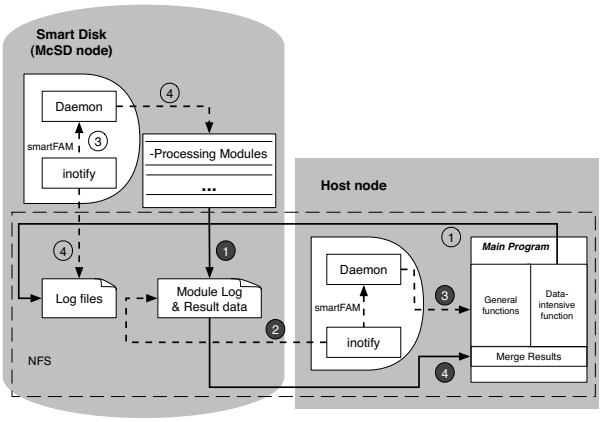


Fig. 5. The implementation of smartFAM - an invocation mechanism that enables a host computing node to trigger data-intensive processing modules in a McSD storage node.

file system or NFS. In the NFS configuration, the host node plays a client role whereas the McSD node performs as a file server. A log-file folder, created in NFS at the server side (i.e., the McSD smart-storage node), can be accessed by the host node via NFS. Each data-intensive processing module/operation has a log file in the log-file folder. Thus, when a new data-intensive module is preloaded to the McSD node, a corresponding log-file is created. The log file of each data-intensive module is an efficient channel for the host node to communicate with the smart-storage node (McSD node). For example, let us suppose that a data-intensive module in the McSD node has input parameters. The host node can pass the input parameters to the data-intensive module residing the McSD node through the corresponding log file. Thus, the host writes the input parameters to the log file that is monitored and read by the data-intensive module. Below we address the following two questions related to usage of log files in McSD:

- (1) How to pass input parameters from a host node to a McSD storage node?
- (2) How to return results from a McSD storage node to a host?

Passing input parameters from a host node to a McSD smart-storage node. When an application running on the host node offloads data-intensive computations to the McSD node, the following five steps are performed so that the host node can invoke a data-intensive module in the smart-storage node via the module's log file (see Fig. 5):
Step 1: The application on the host node writes input parameters of the module to its log file on in McSD. Note that NFS handles communications between the host and McSD via log files.
Step 2: The inotify program in the McSD node monitors all the log files. When the data-intensive module's log file in McSD is changed by the host, inotify informs the Daemon program in smartFAM of McSD.
Step 3: The Daemon program opens the module's log file to retrieve the input parameters passed from the host. Note that this step is not required if no input parameter needs to be transmitted from the host to the McSD node.
Step 4: The data-intensive module is invoked by the Daemon program; the input parameters are passed from Daemon to the module.
Step 5: Go to Step 1 is more data-intensive modules in the McSD node are invoked by the application on the host.

Returning results from a McSD smart-storage node to a host node. Results produced by a data-intensive module in the McSD node must be returned to the module's caller - a calling application that invokes the module from the host node. To achieve this goal, smartFAM takes the following four steps (see Fig. 5):
Step 1: Results produced by the module in the McSD node are written to the module's log file.
Step 2: The inotify program in the host node monitors the log file, checking whether or not the results have been generated by McSD. After the module's log file is modified by McSD (i.e., the results are available in the log file), This inotify program informs the Daemon program in the host node.
Step 3: The Daemon program in the host notifies the calling application that the results from the McSD node are available for further process.
Step 4: The host node accesses the module's log file and obtain the results from the McSD node. Note that this step can be bypassed if no result should be returned from McSD to the host.

B. Partitioning and Merging

A second implementation issue that has not been investigated in the existing smart-storage prototypes is how to process large data sets that are too large to fit in on-node memory. In one of our experiments, we observed that the Phoenix runtime system does not support any application whose required data size exceeds approximately 60% of a computing node's memory size. This is not a critical issue for Phoenix, because Phoenix is a MapReduce framework on shared-memory multi-core processor or multiple processors systems where memory size are commonly larger than those residing in smart storage nodes. On-node memory space in smart storage nodes is typically small compared with front-end high-performance computing nodes. Thus, before we attempted to apply Phoenix in McSD smart disks, we had to address this out-of-core issue - data required for computations in McSD is too large to fit in McSD memory.

Our solution to the aforementioned out-of-core issue is to partition a large data set into a number of small fragments that can fit into on-node memory before calling a MapReduce procedure. Once a large data set is partitioned, the small fragments can be repeatedly processed by the MapReduce procedure in McSD. Intermediate results obtained in each iteration can be merged to produce a final result. Our partitioning solution has two distinct benefits:

- Supporting huge datasets whose size may exceed the memory capacity of a McSD storage node.

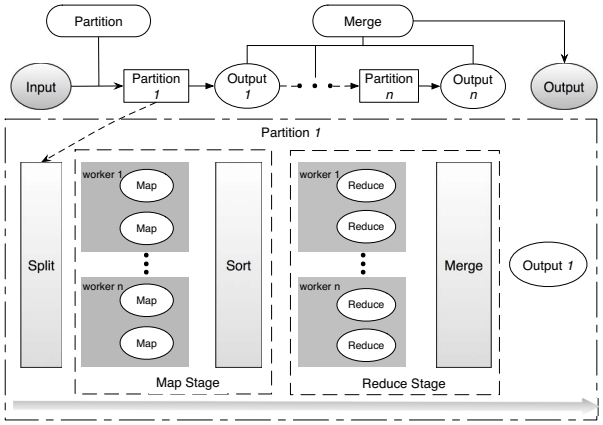


Fig. 6. Workflow of the extended Phoenix model with partitioning and merging

- Boosting performance of data-intensive applications (e.g., word-count) by improving the memory usage of McSD (see Fig. 8 in Section V).

Because both input data sets and emitted intermediate data are located in memory during the MapReduce stage, the memory footprint is at least twice of input data size. The partitioning solution, of course, is only applicable for data-intensive applications whose input data can be partitioned. In our experiments, we evaluated the impact of fragment size on the performance of applications. Evidence (see Fig. 8 in Section V) shows that data partitioning can improve performance of certain data-intensive applications.

C. Incorporating the Partitioning Module into Phoenix

Fig. 6 depicts the work flow of the modified version of Phoenix; it can be considered as a two-stage MapReduce process. The Partition function is provided by the runtime system, while the Merge function needs to be programmed by the user to support different applications. Take an example of a Word-count command: *wordcount [data-file] [partition-size]*. Fragment sizes of every new partitions are determined by (1) the number of [partition-size] provided by the programmer/system and (2) the extra displacements from integrity-check function in order to make sure the new partition is ended correctly. If there is no [partition-size] parameter, the program will run in native way. Otherwise, the number of [partition-size] can be manually filled in by the programmer or automatically determined by the runtime system (e.g., Phoenix). In order to achieve a better performance, the empirical data or the details of operator may be required for the automatic approach. The integrity-check function will automatically return the extra displacements by scanning from the starting point of [partition-size] till the first space, return or the symbol defined by the programmer. The reason we involved the integrity-check procedure to the Partition function is there exists the consistency issue of partitioned data files; the content of the source data file could be broken in shatters (e.g. a word could be cut and placed into two slitted files not on purpose). Fig. 7 describes the integrity-check procedure.

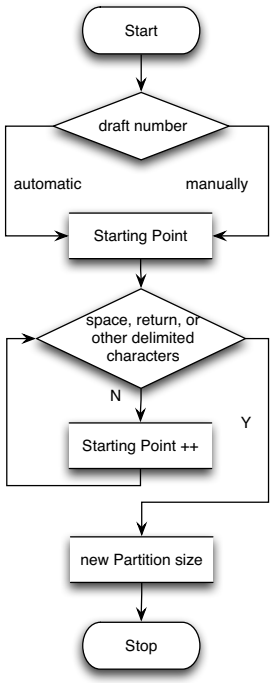


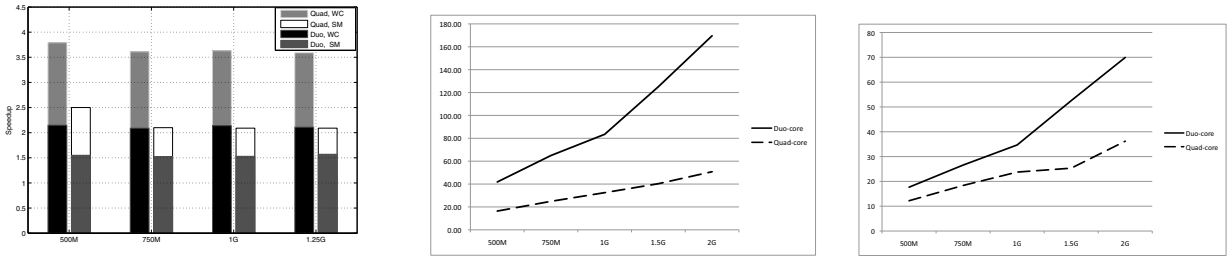
Fig. 7. The workflow diagram of integrity checking.

V. Evaluations

A. Experimental Testbed

We performed our experiments on a 5-node cluster, whose configuration is outlined in Table 1. There are three types of nodes in the cluster: one of host computing node, one of smart storage nodes, and three other general purpose computing nodes. Operating system running on the cluster is Ubuntu 9.04 64-bit version. The nodes in the cluster are connected by Ethernet adapters, Ethernet cables, and one 1Gbit switch. All the general purpose computing nodes share disk space on the host node through Network File System (NFS), while the host node is sharing one folder on the McSD node. The processing modules, extended Phoenix system and SmartFAM have been set up on both the host and SD nodes. Then in order to emulate the routine work, we run the Sandia Micro Benchmark (SMB) among all the nodes except the McSD smart-storage node. We choose MPICH2-1.0.7 as our the message passing interface (MPI) on the cluster. All benchmarks are compiled with gcc 4.4.1. We briefly describe the benchmarks running on our testbed in the following sub-section.

- **Word Count (WC)**: It counts the frequency of occurrence for each word in a set of files. The Map tasks process different sections of the input files and return intermediate data ⟨key, value⟩ that consist of a word and a value of 1. Then the Reduce tasks add up the



(a) Performance Speedup (b) Growth curve of WC on Duo and Quad (c) Growth curve of SM on Duo and Quad

Fig. 8. Single Application Performance. Fig. 8(a) depicts speedups of partition-enabled Phoenix vs. original Phoenix and the sequential approach on both duo-core and quad-core machines. Fig. 8(b) and Fig. 8(c) draws the growth curves of elapsed time on duo-core and quad-core machines. The data size is scaling from 500MB to 1.25GB on Fig. 8(a) and from 500MB to 2GB on Fig. 8(b) and Fig. 8(c).

TABLE I
THE CONFIGURATION OF THE 5-NODE CLUSTER

	Host	SD	Nodes ×3
CPU	Intel Core2 Quad Q9400	Intel Core2 Duo E4400	Intel Celeron 450
Memory	2GB		
OS	Ubuntu 9.04 Jaunty Jackalope 64bit version		
Kernel version	2.6.28-15-generic		
Network	1000Mbps		

values for each identity word. Finally, the words are sorted and printed out in accordance with the frequency in decreasing order.

- **String Match (SM)**: Each Map searches one line in the "encrypt" file to check whether the target string from a "keys" file is in the line. Neither sort or the reduce stage is required.
- **Matrix Multiplication (MM)**: Matrix multiplication is widely applicable to analyze the relationship of two documents. Each Map computes multiplication for a set of rows of the output matrix. It outputs multiplication for a row ID and column ID as the key and the corresponding result as the value. The reduce task is just the identity function.
- **Sandia Micro Benchmark (SMB)**: It is developed by Sandia National Laboratory to evaluate and test high-performance networks and protocols. We use it in our experiment to emulate the routine work.

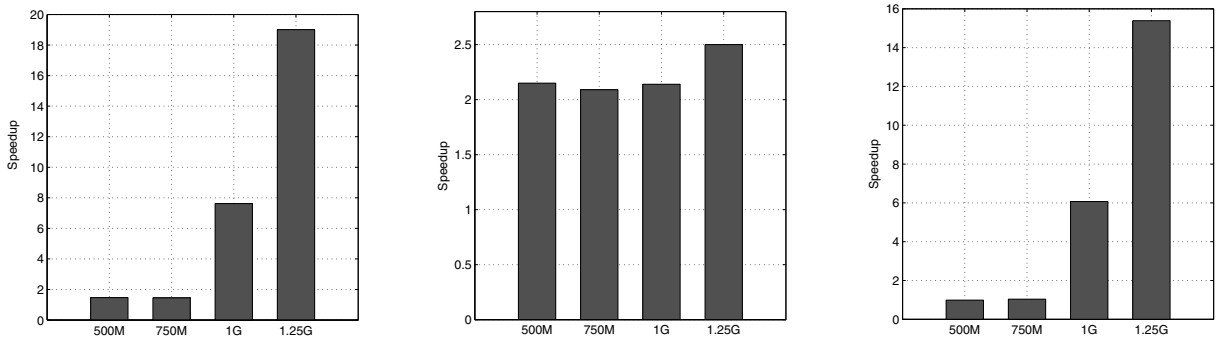
B. Single-Application Performance

Fig. 8 shows the speedup achieved by using the Partition-enabled programming model, relative to the no-partition version and sequential implementation, respectively. In terms of single application benchmarks, we observed that the traditional Phoenix cannot support the Word-count and the String-match for data size larger than 1.5G, because of the memory overflow. From the Fig. 8, when the data size is in a reasonable interval (say, less than half of the memory size), the traditional parallel approach provides almost the same performance. However, in terms of the Word-count, when the data size is huge (compared with the memory size), the elapsed time of Partition-enabled approach is only 1/6 of the traditional one. When comparing with the sequential approach, both the benchmarks can achieve a 2X speedup, which proves the fully utilization of duo-core processor. Fig. 8(b) and Fig. 8(c) show the plots of the execution time versus the size of the input data file on the two SD platforms. From the figure, since we can observe that the performance curve has linear-like growth, our methodology provides scalability performance for its audience objective. We can summarize that: (1) for the very data-size sensitive applications, such as Word Count, the Partition procedure can not only support data size which cannot fit in the physical memory but also improve the performance; (2) for the applications that are not very data-intensive, the Partition model can only enhance their supportability of data-size range. Of course, all those observations are based on the assumption that the applications are partition-able; (3) the last but not the least, the use of our Partition-enabled approach can fully utilize the multicore processor in almost all subjects in this test.

C. Multiple-Application Performance

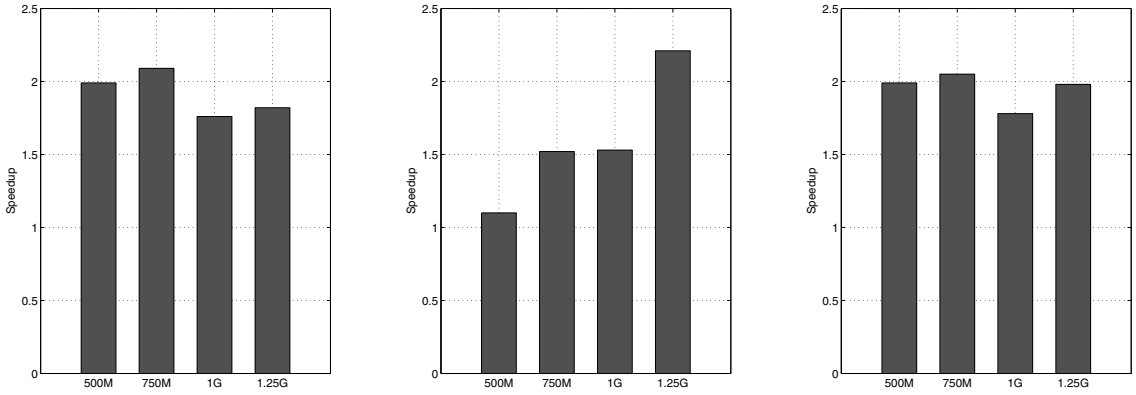
When multiple applications are running concurrently– following the McSD framework, the system should exhibit the basic properties: (1) the system overall throughput should be increased, and (2) the overall performance of the application set should be improved. In order to evaluate our McSD execution framework, we create two multiple-application benchmarks, each of which contains : a computation-intensive function and a data-intensive one. To explore how well our system meets the performance expectations, we report two pairs of application benchmarks: Matrix-multiplicity/Word-count and Matrix-multiplicity/String-match. The first pair is very data-intensive, or memory-consuming, since the memory footprint of Word-Count is around three times of the input data size. On the other hand, the memory footprint of String-Match is around two times of the input data size. Thus, those two are representatives of two levels of data-intensive applications.

For each pair of applications, we set up four scenarios to execute the program: (1) the benchmarks running on the traditional single-core SD mode (a combination of host and single-core SD node), (2) the benchmarks running on the duo-core embedded SD mode without Partition function, (3) the



(a) Host Node Only (b) Tradtional SD (c) McSD without Partition

Fig. 9. **Speedups of Matrix Multiplicity and Word-count.** Trad_SD - traditional smart storage (SD) with single-core processor embedded. DuoC_SD-nopar - duo-core processor embedded smart storage operating in a parallel way without the partitioning function. The benchmarks are running on the multicore host node only in the Host-only scenario. The last one, Host-part, is partitioning-enabled on the Host node. Compared with the traditional smart storage (running sequentially), our McSD improves the overall performance by 2x. With the data size increasing, the elapsed time of non-partitioned approaches (the DuoC-SD and Host-only) can cost 16 to 18 times more than that of the McSD approach.



(a) Host Node Only (b) Tradtitional SD (c) McSD without Partition

Fig. 10. **Speedups of Matrix-multiplicy and String-match.** Compared with the traditional smart storage (SD) running sequentially, our McSD improves the overall performance by 1.5x. When data size is increasing, McSD improves the performance of the non-partitioning approaches (the DuoC-SD and Host-only) by 2x.

programs running on the host node only, and (4) the programs follow the McSD execution framework; the host machine handles the computation-intensive part and the SD machine processes the on-node data-intensive function. Each of the solutions performs three tests: parallel processing without partition, parallel processing with partition and the sequential solution.

Fig. 9 and Fig. 10 illustrate the performance improvement of using the optimized approach, the parallel-enabled one with 600MB partition, against the other scenarios. Fig. 9 and Fig. 10 show speedups on the pair of MM/WC and MM/SM, respectively. We defined the performance speedup to be the ratio of the elapsed time without the optimization technique to that with the McSD technique. From both of the figures, we observe a common point: compared with the traditional (single-core processor equipped) SD, the McSD (duo-core processor embedded) averagely improves the overall performance by 2X for both two pairs of applications. Thus it illustrates the utilization of the duo-core processor. Also, the difference between those two sets of figures is obvious. In terms of the MM/WC, the elapsed time of non-partitioned parallel approaches, host node only and McSD without Partition, increase nonlinearity. When the data size exceeds a threshold, the speedups averagely achieve 6.8X and 17.4X. On the other hand, the McSD can only make slightly improvement when the data size are 500MB and 750MB (below the threshold). In contrary, the speedups of the MM/SM, which represents less data-intensive applications, are both averagely 2X speedup.

As we can see, using our methodology gives better speedups compared with the traditional SD (averagely 2X) and parallel

processing without Partition (maximum to 17X). While the SD being widely considered to be one of the heterogeneous computing platforms, the frameworks like ours will be considered to manage the system and improve the performance.

VI. Conclusion

Processor-enabled smart storage can improve I/O performance of data-intensive applications by processing data directly using storage nodes, because smart nodes avoid moving data back and forth between storage and host computing nodes. Thanks to the escalating manufacturing technology, it is possible to integrate multi-core processors into smart storage nodes. In this study, we implemented a prototype called McSD for multicore-enabled smart storage that can improve performance of data-intensive applications by offloading data processing to multicore processors employed in storage nodes of computing clusters.

Our McSD system differs from conventional smart/active storage in two ways. First, McSD is a smart storage nodes rather than a smart disk. Second, McSD can leverage multicore processors in storage nodes to improve performance of data-intensive applications running on clusters.

McSD along with its programming framework enables programmers to write MapReduce-like code that can be automatically offload data-intensive computation to multicore processors residing in smart storage nodes. The McSD programming framework allows smart storage nodes to take full advantages from embedded multi-core processors. The APIs and a runtime environment in this programming framework automatically handles computation offload, data partitioning, and load balancing. The McSD prototype was implemented in a testbed–a 5-node cluster containing both host computing nodes and McSD smart-storage nodes. Our experimental results were taken by running three real-world applications on the testbed. The tested data-intensive applications include Word Count, String Matching, and Matrix Multiplication. Our multicore-enabled smart storage system - McSD - significantly reduces the execution time of the three applications. Overall, we conclude that McSD is a promising approach to improving I/O performance of data-intensive applications.

Our prototype for multcore-enabled smart storage was built in a MapReduce cluster. The performance of the benchmark applications largely depends on the testbed. Therefore, we will upgrade our testbed (e.g., replace Ethernet with Infiniband) to evaluate the impact of fast network interconnects on McSD. Perhaps the most exciting future work lies in exploring (1) the extensibility of data-processing modules and operations (*i.e. data-intensive applications and database operations*) that are preloaded into McSD smart-disk nodes, (2) the parallelisms among multiple McSD smart disks, and (3) a mechanism in McSD to support fault tolerance and improve reliability.

Acknowledgment

The work reported in this paper was supported by the US National Science Foundation under Grants CCF-0845257 (CAREER), CNS-0757778 (CSR), CCF-0742187 (CPA), CNS-0917137 (CSR), CNS-0831502 (CyberTrust), CNS-0855251 (CRI), OCI-0753305 (CI-TEAM), DUE-0837341 (CCLI), and DUE-0830831 (SFS).

References

[1] Apache hadoop, 2006. `http://lucene.apache.org/hadoop/`.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[3] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[4] Steve C. Chiu, Wei-keng Liao, Alok N. Choudhary, and Mahmut T. Kandemir. Processor-embedded distributed smart disks for i/o-intensive workloads: architectures, performance models and evaluation. *J. Parallel Distrib. Comput.*, 65(4):532–551, 2005.

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

[6] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, 2000.

[7] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[8] Windsor W. Hsu, Honesty C. Young, and Alan Jay Smith. Projecting the performance of decision support workloads on systems with smart storage (smartstor). In *ICPADS '00: Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, page 417, Washington, DC, USA, 2000. IEEE Computer Society.

[9] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, 1998.

[10] Gokhan Memik, Alok Choudhary, and Mahmut T. Kandemir. Design and evaluation of smart disk architecture for dss commercial workloads. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, page 335, Washington, DC, USA, 2000. IEEE Computer Society.

[11] Anurag Acharya Mustafa, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation, 1998.

[12] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[13] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[16] Anand Lal Shimpi. Anandtech: Intel's 90nm pentium m 755: Dothan investigated, Dec 2007. `http://www.anandtech.com/cpuchipsets/`.

[17] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 73–88, Berkeley, CA, USA, 2003. USENIX Association.

[18] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Evaluation of active disks for large decision support databases. Technical report, Santa Barbara, CA, USA, 1999.