COMPUTER SCIENCE
AND
SOFTWARE ENGINEERING

AUBURN UNIVERSITY

SAMUEL GINN COLLEGE
OF ENGINEERING

April 28, 2011

Dr. Ahmed Louri
Program Director
Division of Computer and Network Systems
Directorate for Computer & Information Science & Engineering
National Science Foundation

Dear Dr. Louri:

Thank you very much for your time and efforts on Dr. Xiao Qin's NSF CAREER award. I am delighted to write this letter to show my support for Dr. Xiao Qin's CAREER project that is currently being supported by your Software and Hardware Foundations (NSF-SHF) program.

In the past two years, Dr. Qin has completed the following four research tasks described in his NSF CAREER proposal:

- Developing Multicore-Embedded Smart Disks
- Improving MapReduce Performance through Data Placement}
- An Offloading Framework for I/O Intensive Applications on clusters}
- Using Active Storage to Improve the Bioinformatics Application Performance

In the first research task, Dr. Qin and his doctoral students developed a multicore-embedded smart disk system that can improve performance of dataintensive applications by offloading data processing to multicore processors embedded in disk drives.

In the second research task, Dr. Qin's research group shows that ignoring the data locality issue in heterogeneous environments can noticeably reduce the MapReduce performance. They addressed the problem of how to place data across nodes in a way that each node has a balanced data processing load.

In the third task, Dr. Qin's team developed an offloading framework that is able to be easily applied in either an existing or a completely newly developed I/O intensive application.

In the last task, Dr. Qin and his graduate students implemented a pipelining mechanism that leverages active storage to maximize throughput of data-intensive applications on a high-performance cluster.

Overall, I am very satisfied with Dr. Qin's progress on his NSF CAREER project. I will continue my strongest support for his NSF CAREER project.

Sincerely,

Kai H. Chang
Professor and Chair
changka@auburn.edu
334-844-6310

Years 1 and 2 (2009-2011) Annual Report for NSF Award CCF-0845257

CAREER: Multicore-Based Parallel Disk Systems for Large-Scale Data-Intensive Computing

Xiao Qin *

Department of Computer Science and Software Engineering

Auburn University, Auburn, AL 36849

April 29, 2011

1 Research and Education Activities

1.1 Developing Multicore-Embedded Smart Disks

In this study, we developed a multicore-embedded smart disk system that can improve performance of dataintensive applications by offloading data processing to multicore processors embedded in disk drives. 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 disks are promising devices to perform complex on-disk operations. Smart disks can avoid moving a huge amount of data back and forth between storage systems and host processors. To enhance the performance of data-intensive applications, we have designed a smart disk called McSD, in which a multicore processor is embedded. 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 host CPUs and multicore embedded smart disks. To fully utilize multicore processors in smart disks, we have implemented the MapReduce model for McSDs to handle parallel computing. A prototype of McSD has been implemented in a PC cluster connected by Gigabit Ethernet. McSD significantly reduces the execution time of word count, string matching, and matrix multiplication. Overall, we conclude that, integrated with MapReduce, multicore-embedded smart disk systems are a promising approach for improving I/O performance of data-intensive applications.

*xqin@auburn.edu

1.2 Improving MapReduce Performance through Data Placement

MapReduce has become an important distributed processing model for large-scale data-intensive applications like data mining and web indexing. HadoopCan open-source implementation of MapReduce is widely used for short jobs requiring low response time. The current Hadoop implementation assumes that computing nodes in a cluster are homogeneous in nature. Data locality has not been taken into account for launching speculative map tasks, because it is assumed that most maps are data-local. Unfortunately, both the homogeneity and data locality assumptions are not satisfied in virtualized data centers. We show that ignoring the data locality issue in heterogeneous environments can noticeably reduce the MapReduce performance. In this research task, we address the problem of how to place data across nodes in a way that each node has a balanced data processing load. Given a data intensive application running on a Hadoop MapReduce cluster, our data placement scheme adaptively balances the amount of data stored in each node to achieve improved data-processing performance. Experimental results on two real data-intensive applications show that our data placement strategy can always improve the MapReduce performance by rebalancing data across nodes before performing a data-intensive application in a heterogeneous Hadoop cluster.

1.3 An Offloading Framework for I/O Intensive Applications on clusters

In this study, we propose an offloading framework that is able to be easily applied in either an existing or a completely newly developed I/O intensive application with minor efforts. In particular, we not only illustrate core theory of designing an offloading program, such as structures and methods of offloading programs and controlling execution paths, but also discuss several essential issues which are required to be carefully considered in implementation, including configuration, offloading work flow, programming interfaces and data sharing. In order to compare performance of offloading applications with corresponding original versions, we have applied offloading to five programs and measured them in a typical cluster. The experimental results show that offloading applications run much faster than original ones and systems on which offloading applications execute have remarkably lower network burden than ones original applications run on.

1.4 Using Active Storage to Improve the Bioinformatics Application Performance

Active storage is an effective technique to improve applications' end-to-end performance by offloading data processing to storage nodes. In this research task, we present a pipelining mechanism that leverages active storage to maximize throughput of data-intensive applications on a high-performance cluster. The mechanism overlaps data processing in active storage with parallel computations on the cluster, thereby allowing clusters and their active storage nodes to perform computations in parallel. To demonstrate the effectiveness of

the mechanism designed for active storage, we implemented a parallel pipelined application called pp-mpiBLAST, which extends mpiBLAST that is an open-source parallel BLAST tool. Our pp-mpiBLAST relies on active storage to filter unnecessary data and format databases, which are then forwarded to the cluster running mpiBLAST. We develop an analytic model to study the scalability of pp-mpiBLAST on large-scale clusters. Measurements made from a working implementation suggest that this method reduces mpiBLAST's overall execution time by up to 50%.

1.5 Mini Conference in the Advanced Operating Systems Class

A mini-conference model was used to motivate and educate graduate students to conduct research projects in the discipline of storage systems, energy-efficient computing, and prefetching/caching for file systems. By the end of the Spring 2010 semester, when the Comp7500 C Advanced Operating Systems Class is taught, each graduate student is required to write a research paper and submit to a mini-conference. All the student papers were reviewed and each student gave a presentaion of 20 minutes. After each presentation, each student had a question-answer session of 5 minutes. The PI also gave constructive comments and suggestions on each students research project. In this mini-conference model, the graduate students who are taking the Comp7500 class improved their presentation and communication skills. After we receive feedbacks from the graduate students, we will formally evaluate the this class next semester.

2 Findings

A focus of the research activities carried out in the last year is (1) the development of multicore-embedded smart disks and (2) a data placement module in heterogeneous hadoop clusters.

2.1 Multicore-Embedded Smart Disks [7]

2.1.1 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 [12]. To improve the performance of large data-intensive applications, we designed McSD - a prototype of multicore-embedded smart disks. Different from the existing smart-disk solutions, McSD addresses the performance needs of data-intensive applications using multi-core processors embedded in hard disks.

Fig. 1 depicts the McSD prototype, where each smart disk contains a multicore-processor, memory, and a SATA disk drive. In what follows, let us address the following design issues.

- How to build a testbed where a McSD smart disk is connected to a host computing node?

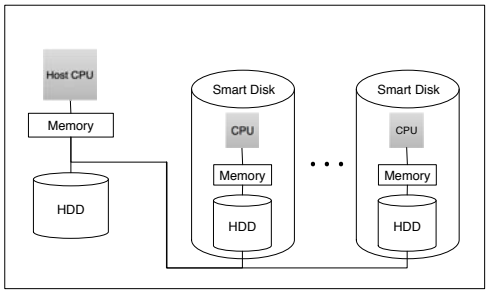


Figure 1: McSD - The prototype of multicore-embedded smart disks. Each smart disk in the prototype contains memory, a SATA disk drive, and a multicore-processor.

- How to evaluate the performance of McSD in the testbed?
- How to fully utilize a multi-core processor embedded in McSD?
- What is the programming framework for McSD?
- How to pass input parameters from a host to its McSD smart disk?

2.1.2 Design of the McSD Prototype

A traditional smart disk consists of an embedded processor, a disk controller, on-disk memory, a local disk drive, and a network interface controller (NIC). In our McSD prototype, we integrate multi-core processors into smart disks. Storage-interfaces of in the existing smart disk prototypes were not well implemented, 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 a smart disk, we designed a communication mechanism similar to the The file alteration monitor. In our 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 in McSD. Without using NICs, the McSD prototype becomes closer to actual smart disk systems. The design details will be described in the two subsections below.

2.1.3 A Testbed for McSD

Although a few smark disk prototypes have been developed, there is no off-the-shelf commodity smart disks. As such, we built a testbed for the McSD prototype. Fig. 2 briefly outlines the testbed, where two PCs are connected through the fast Ethernet. The first PC in the testbed plays the role of host computing node, whereas the second one performs as the McSD smart-disk 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

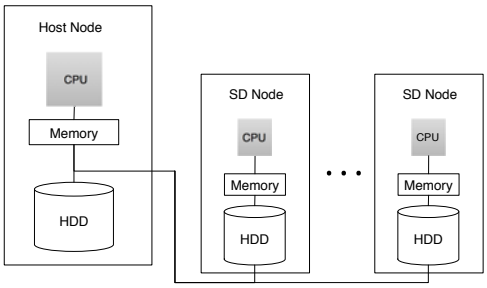


Figure 2: A testbed for the McSD prototype. A host computing node and an McSD storage node are connected via a fast Ethernet switch. The host node can access the disk drives in McSD through the networked file system or NFS.

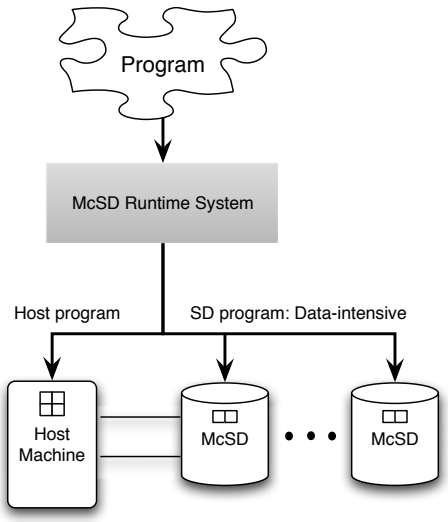


Figure 3: The programming framework for a host computing node supported by an McSD smart disk.

client computer; the McSD node is configured as an NFS server. We chose to use NSF as an efficient means of connecting the host node and the smart-disk node, because data transfers between the host and smart-disk nodes are handled by NSF.

We run three state-of-the-art benchmarks on this testbed to evaluate the performance of the McSD prototype. The benchmarks considered in our experiments include word count, string matching, and matrix-multiplication.

2.1.4 A Programming Framework

Fig 3 shows a programming framework for a host computing node supported by an McSD smart disk. The framework generates an optimized operation plan for data-intensive programs running on the McSD testbed for the description of the testbed), where there is a host

computing node and a McSD smart-disk node. The framework automatically assigns general purpose operations to the host computing nodes and offload data-intensive operations to the McSD node, in which Phoenix for the description of Phoenix) handles parallel data processing. Although applying Phoenix in the McSD node can not increase performance for all applications running in our testbed, Phoenix can substantially boost performance of data-intensive applications. Because this programming framework provides a relatively flexible autonomy, data processing modules (e.g., word-count, sort, and other primitive operations) can be readily added into a McSD smart disk.

To seamlessly integrate Phoenix into a McSD smart disk, we addressed the issue of limited embedded memory in McSD by implementing new functions like data partitioning, which split input data files whose memory footprints exceed the memory capacity of the McSD smart disk.

2.1.5 System Workflow and Configuration

Unlike the previous network-attached smart disks, McSD uses the SATA interface to transfer data. We implemented the McSD prototype using a host computing node and a multicore-embedded storage node and 2.1.3 for the design issues of the prototype). In the prototype, the multicore storage node has no keyboard, mouse, and display unit. Note that storage nodes in other existing smart-disk prototypes have keyboard and mouse activities. Compared with the earlier prototypes, our McSD prototype better resembles next-generation multicore-embedded smart disks. An actual smart disk only needs to process on-disk data-intensive operations. In other words, smart disks only provides some primitive functions termed as data-intensive processing modules (or processing modules for short) in the McSD prototype.

Fig. 2 shows the hardware configuration of the McSD prototype where a host node is connected to a McSD storage node through the SATA bus interface. One of the most important implementation issues is to allow a host computing node to offload data-intensive computations to McSD. There are two general approaches to implementing computation offloading. First, each offloaded data-intensive operation or module are delivered from a host node to a McSD storage node (hereinafter refered to as McSD or McSD node) when the operation or module needs to be processed by McSD. Second, all data-intensive operations and modules are preloaded and stored in the McSD node. Although the first approach can handle the dynamic environment problem where data-intensive operations/modules are not predictable, the downside of the first approach lies in high communication overhead between host nodes and McSD nodes. The second approach reduces the communication overhead caused by moving data-intensive operations/modules, because the operations/modules are residing in McSD prior to the execution of the data-intensive programs.

In the process of implementing the McSD prototype, we took the second approach - preloading data-intensive modules. We believe that the preloading approach is practical for a vast variety of programs, where data-intensive processing modules can be determined before the programs are executed in a host computing node accompanied by a McSD smart disk. In our preloading approach, the program running on the computing node has to invoke the processing modules preloaded to the McSD node. An invocation mechanism, called

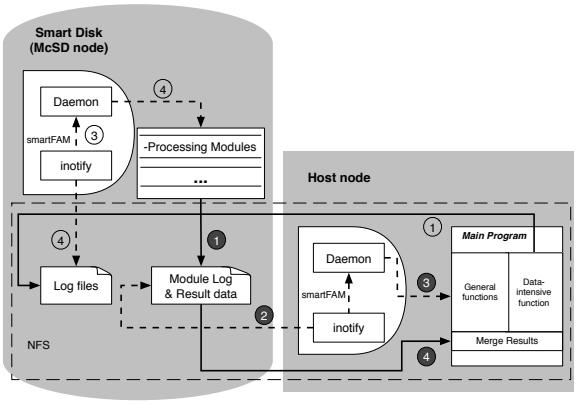


Figure 4: The implementation of smartFAM - an invocation mechanism that enables a host computing node to trigger data-intensive processing modules in a McSD storage node (smart-disk node).

smart-file-alternation monitor (smartFAM), was implemented to enable the host node to readily trigger the processing modules in the McSD node. The implementation issues of smartFAM are addressed in the next subsection.

2.1.6 Implementation of smartFAM

Fig. 4 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-disk data-intensive operations or modules.

To make our McSD prototype closely resemble future multicore-embedded smart disks, we connected the host node with the McSD smart-disk node using the Linux network 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-disk 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-disk 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 node?

- (2) how to return results from a McSD node to a host.

Passing input parameters from a host node to a McSD smart-disk 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-disk node via the module's log file (see Fig. 4):
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-disk 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. 4):
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.

2.1.7 Partitioning and Merging

A second implementation issue that has not been investigated in the existing smart-disk prototypes is how to process large data sets that are too large to fit in on-disk 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 disks. On-disk memory space

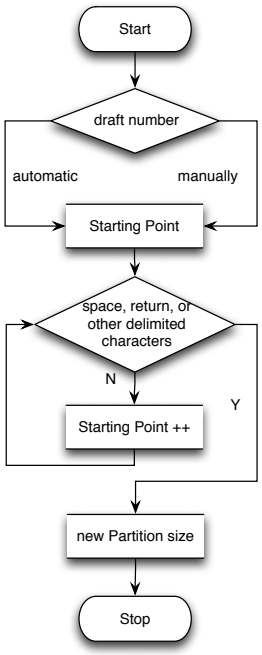


Figure 5: The workflow diagram of integrity checking.

in smart disks is typically small due to smart disks' constraints on size, power consumption, and manufacturing cost. 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-disk 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 on-disk memory capacity.
- Boosting performance of data-intensive applications (e.g., word-count) by improving on-disk memory usage (see Fig. 7 in Section 2.4.5).

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. 7 in Section 2.4.5) shows that data partitioning can improve performance of certain data-intensive applications.

Fig. 26 describes the procedure inside a partitioning function. Fragment sizes of new partitions are determined by (1) the draft number provided by programmers or systems and

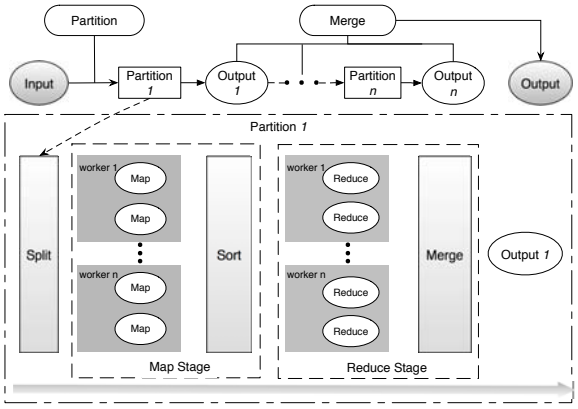


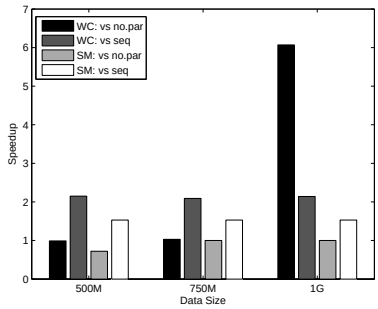
Figure 6: Workflow of the extended Phoenix model with partitioning and merging

(2) the extra displacements from the integrity-check function in order to make sure the new partition is ended correctly. The draft number can be manually filled in by the programmer or automatically determined by a runtime system. Empirical data or operator details may be required for the automatic scheme to improve performance. The integrity-checking function can automatically return the extra displacements by scanning from the starting point of the draft number till the first space, return or the symbol defined by the programmer. Fig. 6 depicts the workflow of the extended Phoenix in which the partitioning and merging procedures are incoporated. Conceptually, the entire partitioning/merging process can be envisioned as a two-stage MapReduce process. The partitioning function is provided by the runtime system, while the application-dependent Merging function needs to be programmed by developers to support different applications.

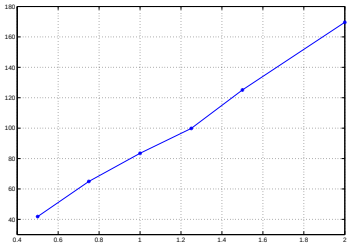
2.1.8 Experimental Testbed

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 node, one of Smart Disk nodes, and three other general purpose 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 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-disk 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.

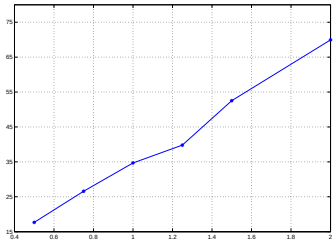
- **Word Count (WC)**: It counts the frequency of occurrence for each word in a set of files. The Map tasks process different sections of the input files and return intermediate data ⟨key, value⟩ that consist of a word and a value of 1. Then the Reduce tasks add



(a) Performance Speedup of scaling input data size.



(b) Growth curve of Word Count



(c) Growth curve of String Match

Figure 7: Speedups of Word Count and String Matching on partition-enabled Phoenix vs. original Phoenix and the sequential approach. In Fig. 7(a), the first two bars of each set are speedups for Word Count using our approach, original Phoenix, and the sequential method, respectively. The other two bars are speedups for the String-Matching benchmark.

up the 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.

Table 1: The Configuration of the 5-Node Cluster

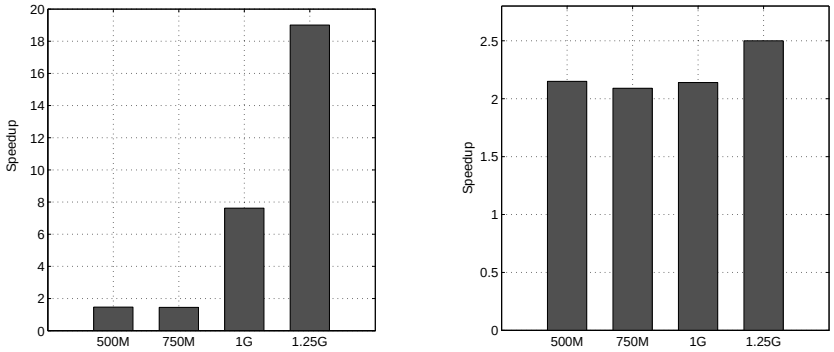
	Host	SD	Nodes ×3
CPU	Intel Core2 Quad Q9400	Intel Core2 Duo E4400	Intel Celeron 450
Memory	2GB		
OS	Ubuntu 9.04 Jaunty Jackalope 64bit version		
Kernel version	2.6.28-15-generic		
Network	1000Mbps		

2.1.9 Single-Application Performance

Fig. 7 shows the speedup achieved by using the Partition-enabled programming model, relative to the no-partition version and sequence 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. 7, 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. 7(b) and Fig. 7(c) show the plots of the execution time versus the size of the input data file on the SD platform. 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.

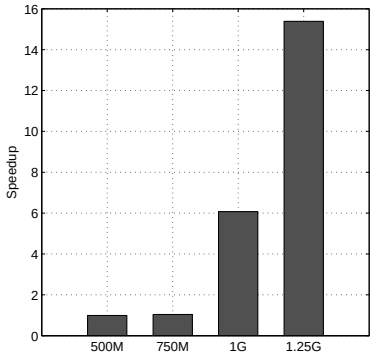
2.1.10 Multiple-application Performance

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 majorly 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 foot-



(a) Host Node Only

(b) Tradtional SD

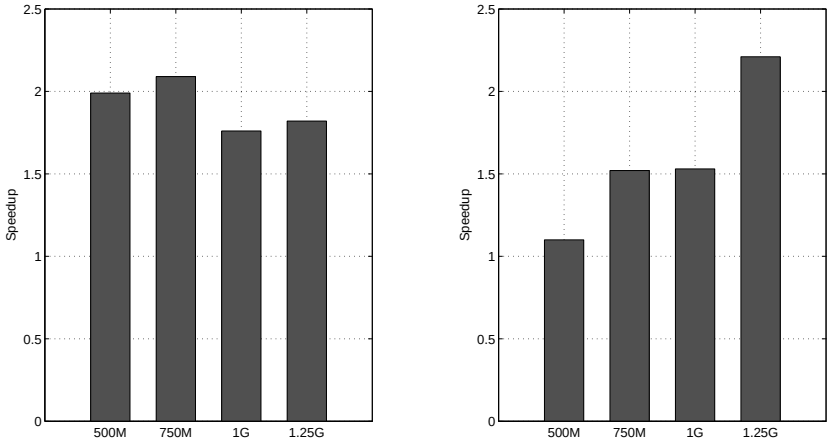


(c) McSD without Partition

Figure 8: **Speedups of Matrix Multiplicity and Word-count.** Trad_SD - traditional smart disk (SD) with single-core processor embedded. DuoC_SD-nopar - duo-core processor embedded smart disk 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 disk (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.

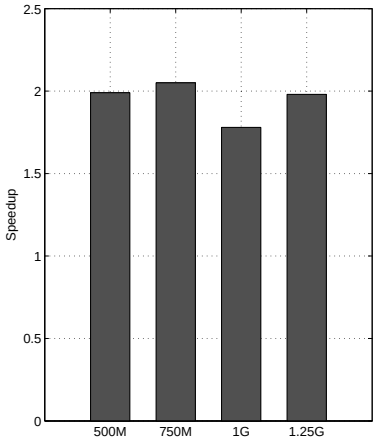
print of Word-Count is around three times of the input data size. On the other hand, the memory footprint of String-Match is around two times of the input data size. Thus, those two are representatives of two levels of data-intensive applications.

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 programs running on the host node only, and (4) the they follow the McSD execution framework. In the host machine handles the computation-



(a) Host Node Only

(b) Tradtitional SD



(c) McSD without Partition

Figure 9: **Speedups of Matrix-multiplicy and String-match.** See Fig. 8 for legend details. Compared with the traditional smart disk (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.

intensive part and the SD machine processes the on-disk data-intensive function. From the data-intensive function perspective, each of the solutions involves three tests: parallel processing without partition, parallel processing with partition and the sequential solution.

Fig. 8 and Fig. 9 illustrate the performance improvement of using the optimized approach, the parallel-enabled one with 600MB partition, against the other scenarios. Fig. 8 and Fig. 9 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 tech-

nique to that with the McSD technique. From both of the figures, we observe a common point: compared with the traditional (single-core processor equiped) SD, the McSD (duo-core processor embedded) avergely improves the overall performance by 2 for both two pairs of applications. Thus it illustrates that our McSD can fully utilize the multicore processor by using of MapReduce parallel programming model. The difference between those two sets of figures 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. However, the McSD can only make slightly improvement when the data size are 500MB and 750MB (less than half of the memory size). In contrary, the speedups of the MM/SM ones, which are less data-intensive than the first pair, are remain almost in the same stage.

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 needed to help manage the system and improve the performance.

2.2 Data Placement in Heterogeneous Hadoop Clusters [15]

2.2.1 Data Placement in Heterogeneous Clusters

In a cluster where each node has a local disk, it is efficient to move data processing operations to nodes where application data are located. If data are not locally available in a processing node, data have to be migrated via network interconnects to the node that performs the data processing operations. Migrating huge amount of data leads to excessive network congestion, which in turn can deteriorate system performance. HDFS enables Hadoop MapReduce applications to transfer processing operations toward nodes storing application data to be processed by the operations.

In a heterogeneous cluster, the computing capacities of nodes may vary significantly. A high-speed node can finish processing data stored in a local disk of the node faster than low-speed counterparts. After a fast node complete the processing of its local input data, the node must support load sharing by handling unprocessed data located in one or more remote slow nodes. When the amount of transferred data due to load sharing is very large, the overhead of moving unprocessed data from slow nodes to fast nodes becomes a critical issue affecting Hadoop's performance. To boost the performance of Hadoop in heterogeneous clusters, we aim to minimize data movement between slow and fast nodes. This goal can be achieved by a data placement scheme that distribute and store data across multiple heterogeneous nodes based on their computing capacities. Data movement can be reduced if the number of file fragments placed on the disk of each node is proportional to the node's data processing speed.

To achieve the best I/O performance, one may make replicas of an input data file of a Hadoop application in a way that each node in a Hadoop cluster has a local copy of the input data. Such a data replication scheme can, of course, minimize data transfer among slow and

fast nodes in the cluster during the execution of the Hadoop application. The data-replication approach has several limitations. First, it is very expensive to create replicas in a large-scale cluster. Second, distributing a large number of replicas can wasterfully consume scarce network bandwidth in Hadoop clusters. Third, storing replicas requires an unreasonably large amount of disk capacity, which in turn increases the cost of Hadoop clusters.

Although all replicas can be produced before the execution of Hadoop applications, significant efforts must be make to reduce the overhead of generating replicas. If the data-replication approach is employed in Hadoop, one has to address the problem of high overhead for creating file replicas by implementing a low-overhead file-replication mechanism. For example, Shen and Zhu developed a proactive low-overhead file replication scheme for structured peer-to-peer networks [14]. Shen and Zhu's scheme may be incorporated to overcome this limitation.

To address the above limitations of the data-replication approach, we are focusing on data-placement strategies where files are partitioned and distributed across multiple nodes in a Hadoop cluster without being duplicated. Our data placement approach does not require any comprehensive scheme to deal with data replicas.

In our data placement management mechanism, two algorithms are implemented and incorporated into Hadoop's HDFS. The first algorithm is to initially distribute file fragments to heterogeneous nodes in a cluster (see Section 2.2.2). When all file fragments of an input file required by computing nodes are available in a node, these file fragments are distributed to the computing nodes. The second data-placement algorithm is used to reorganize file fragments to solve the data skew problem (see Section 2.2.3). There two cases in which file fragments must be reorganized. First, new computing nodes are added to an existing cluster to have the cluster expanded. Second, new data is appended to an existing input file. In both cases, file fragments distributed by the initial data placement algorithm can be disrupted.

2.2.2 Initial Data Placement

The initial data placement algorithm begins by first dividing a large input file into a number of even-sized fragments. Then, the data placement algorithm assigns fragments to nodes in a cluster in accordance to the nodes' data processing speed. Compared with low-performance nodes, high-performance nodes are excepted to store and process more file fragments. Let us consider a MapReduce application and its input file in a heterogeneous Hadoop cluster. Regardless of the heterogeneity in node processing power, the intial data placement scheme has to distribute the fragments of the input file so that all the nodes can complete processing their local data within almost the same time.

In our experiments we observed that the computing capability of each node is quite stable for certain tested Hadoop applications, because the response time of these Hadoop applications on each node is linearly proportional to input data size. As such, we can quantify each node's processing speed in a heterogeneous cluster using a new term called computing ratio. The computing ratio of a computing node with respect to a Hadoop application can be calculated by profiling the application (see Section 2.2.4 for details on how to determine

computing ratios). It is worth noting that the computing ratio of a node may vary from application to application.

2.2.3 Data Redistribution

Input file fragments distributed by the initial data placement algorithm might be disrupted due to the following reasons: (1) new data is appended to an existing input file; (2) data blocks are deleted from the existing input file; and (3) new data computing nodes are added into an existing cluster. To address this dynamic data load-balancing problem, we implemented a data redistribution algorithm to reorganize file fragments based on computing ratios.

The data redistribution procedure is described as the following steps. First, like initial data placement, information regarding the network topology and disk space utilization of a cluster is collected by the data distribution server. Second, the server creates two node lists: a list of nodes in which the number of local fragments in each node exceeds its computing capacity and a list of nodes that can handle more local fragments because of their high performance. The first list is called over-utilized node list; the second list is termed as under-utilized node list. Third, the data distribution server repeatedly moves file fragments from an over-utilized node to an underutilized node until the data load are evenly distributed. In a process of migrating data between a pair of an over-utilized and an underutilized nodes, the server moves file fragments from a source node in the over-utilized node list to a destination node in the underutilized node list. Note that the server decides the number of bytes rather than fragments and moves fragments from the source to the destination node. The above data migration process is repeated until the number of local fragments in each node matches its speed measured by computing ratio.

2.2.4 Measuring Heterogeneity

Before implementing the initial data placement algorithm, we need to quantify the heterogeneity of a Hadoop cluster in terms of data processing speed. Such processing speed highly depends on data-intensive applications. Thus, heterogeneity measurements in the cluster may change while executing different MapReduce applications. We introduce a metric - called computing ratio - to measure each node's processing speed in a heterogeneous cluster. Computing ratios are determined by a profiling procedure carried out in the following steps. First, the data processing operations of a given MapReduce application are separately performing in each node. To fairly compare processing speed, we ensure that all the nodes process the same amount of data. For example, in one of our experiments the input file size is set to 1GB. Second, we record the response time of each node performing the data processing operations. Third, the shortest response time is used as a reference to normalize the response time measurements. Last, the normalized values, called computing ratios, are employed by the data placement algorithm to allocate input file fragments for the given MapReduce application.

Now let us consider an example to demonstrate how to calculate computing ratios used to guide the data distribution process. Suppose there are three heterogeneous nodes (i.e., Node

A, B and C) in a Hadoop cluster. After running a Hadoop application on each node, one collects the response time of the application on node A, B and C is 10, 20 and 30 seconds, respectively. The response time of the application on node C is the shortest. Therefore, the computing ratio of node A with respect to this application is set to 1, which becomes a reference used to determine computing ratios of node B and C. Thus, the computing ratios of node B and C are 2 and 3, respectively. Recall that the computing capacity of each node is quite stable with respect to a Hadoop application. Hence, the computing ratios are independent of input file sizes. Table 2 shows the response times and computing ratios for each node in a Hadoop cluster. Table 2 also shows the number of file fragments to be distributed to each node in the cluster. Intuitively, the fast computing node (i.e., node A) has to handle 30 file fragments whereas the slow node (i.e., 3) only needs to process 10 fragments.

Table 2: Computing ratios, response times and number of file fragments for three nodes in a Hadoop cluster

Node	Responce time	Ratio	File fragments	Speed
Node A	10	1	30	Fastest
Node B	20	2	20	Average
Node C	30	3	10	Slowest

2.2.5 Sharing Files among Multiple Applications

The heterogeneity measurement of a cluster depends on data-intensive applications. If multiple MapReduce applications must process the same input file, the data placement mechanism may need to distribute the input file's fragments in several ways - one for each MapReduce application. In the case where multiple applications are similar in terms of data processing speed, one data placement decision may fit the needs of all the applications.

2.2.6 Data Distribution.

File fragment distribution is governed by a data distribution server, which constructs a network topology and calculates disk space utilization. For each MapReduce application, the server generates and maintains a node list containing computing-ratio information. The data distribution server applies the round-robin algorithm to assign input file fragments to heterogeneous nodes based on their computing ratios.

A small value of computing ratio of a node indicates a high speed of the node, meaning that the fast node must process a large number of file fragments. For example, let us consider a file comprised of 60 file fragments to be distributed to node A, B, and C. We assume the computing ratios of these three nodes are 1, 2 and 3, respectively (see Table 2). Given the computing ratios, we can conclude that among the three computing nodes, node A is the fastest one whereas node B is the slowest node. As such, the number of file fragments assigned to each node is proportional to the node's processing speed. In this example, the

data distribution server assigns 30 fragments to node A, 20 fragments to node B, and 10 fragments to node C (see Table 2).

2.2.7 Evaluation

We used two data-intensive applications - Grep and WordCount - to evaluate the performance of our data placement mechanism in a heterogeneous Hadoop cluster. The tested cluster consists of five heterogeneous nodes, whose parameters are summarized in Table 3. Both Grep and WordCount are two MapReduce applications running on Hadoop clusters. Grep is a tool searching for a regular expression in a text file; whereas WordCount is a program used to count words in text files.

Table 3: Five Nodes in a Hadoop Heterogeneous Cluster

Node	CPU Model	CPU(hz)	L1 Cache(KB)
Node A	Intel Core 2 Duo	2 ×1G=2G	204
Node B	Intel Celeron	2.8G	256
Node C	Intel Pentium 3	1.2G	256
Node D	Intel Pentium 3	1.2G	256
Node E	Intel Pentium 3	1.2G	256

We followed the approach described in Section 2.2.4 to obtain computing ratios of the five computing nodes with respect of the Grep and WordCount applications (see Table 4). The computing ratios shown in Table 4 represent the heterogeneity of the Hadoop cluster with respect to Grep and WordCount. We conclude from the results given in Table 4) that computing ratios of a Hadoop cluster are application dependent. For example, node A is 3.3 times faster than nodes C-E with respect to the Grep application; node A is 5 (rather than 3.3) times faster than nodes C-E when it comes to the WordCount application. The implication of the results is that given a heterogeneous cluster, one has to determine computing ratios for each Hadoop application. Note that computing ratios of each application only needs to be calculated once for each cluster. If the configuration of a cluster is updated, computing ratios must be determined again.

Table 4: Computing Ratios of the Five Nodes with Respective of the Grep and WordCount Applications

Computer Node	Ratios for Grep	Ratios for WordCount
Node A	1	1
Node B	2	2
Node C	3.3	5
Node D	3.3	5
Node E	3.3	5

Figs. 10 and 11 show the response times of the Grep and WordCount application running on each node of the Hadoop cluster when the input file size is 1.3 GB and 2.6 GB, respectively.

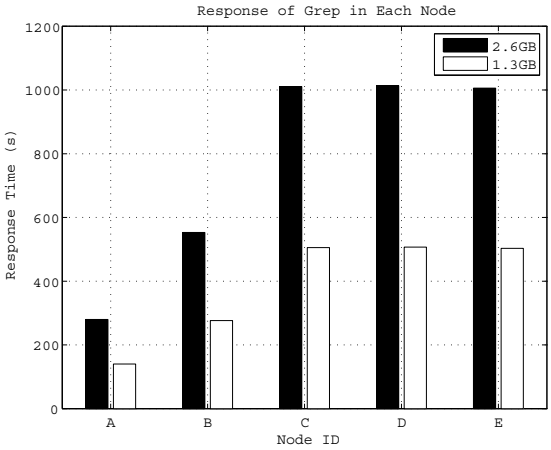


Figure 10: Response time of Grep running on the 5-node Hadoop heterogeneous cluster.

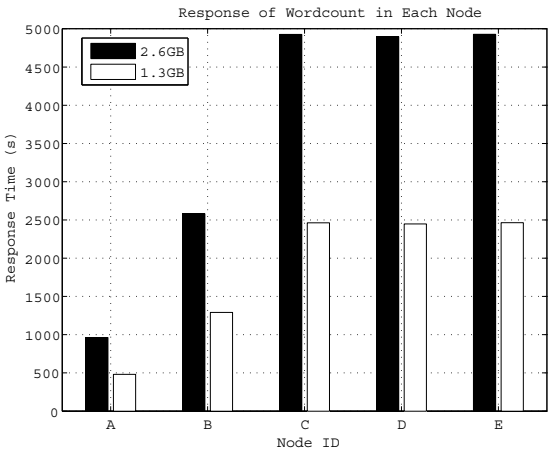


Figure 11: Response time of WordCount running on the 5-node Hadoop heterogeneous cluster.

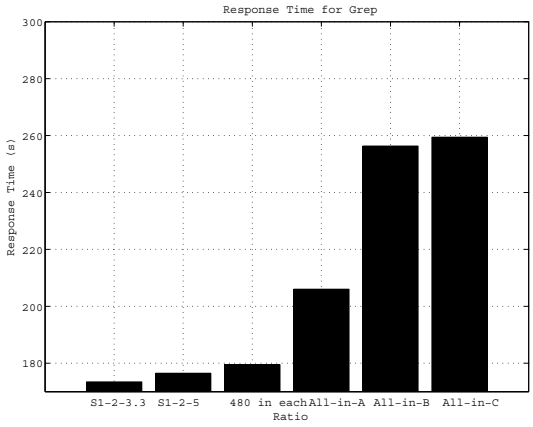


Figure 12: Impact of data placement on performance of Grep.

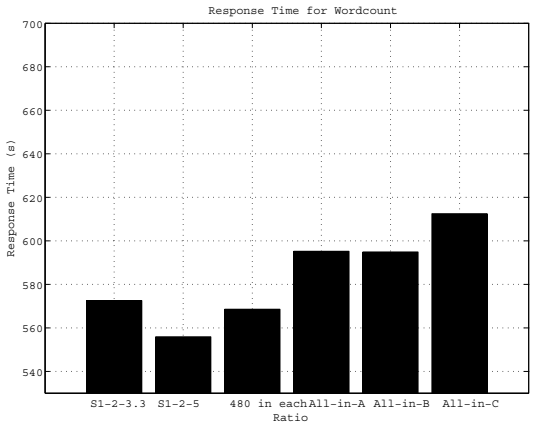


Figure 13: Impact of data placement on performance of WordCount.

The results plotted in Figs. 10 and 11 suggest that computing ratios are independent of input file size, because the response times of Grep and WordCount are proportional to the file size. Regardless of input file size, the computation ratios for Grep and WordCount on the 5-node Hadoop clusters remain unchanged as listed in Table 4.

Given the same input file size, Grep's response times are shorter than those of WordCount (see Figs. 10 and 11). As a result, the computing ratios of Grep are different from those of WordCount (see Table 4).

Table 5: Six Data Placement Decisions

Notation	Data Placement Decisions
S1-2-3.3	Dirstribuating files under the computing ratios of the grep. (This is an optimal data placement for Grep)
S1-2-5	Dirstribuating files under the computing ratios of the wordcount. (This is an optimal data placement for WordCount)
480 in each	Average distribution of files to each node.
All-in-A	Allocating all the files to node A.
All-in-B	Allocating all the files to node B.
All-in-C	Allocating all the files to node C.

Now we are positioned to evaluate the impacts of data placement decisions on the response times of Grep and WordCount (see Figs. 12 and 13). Table 5 shows six representative data placement decisions, including two optimal data-placement decisions (see S1-2-3.3 and S1-2-5 in Table 5) for the Grep and WordCount applications. The file fragments of input data are distributed and placed on the five heterogeneous nodes based on six different data placement decisions, among which two optimal decisions (i.e., S1-2-3.3 and S1-2-5 in Table 5) are made based on the computing ratios given Table 4.

Let us use an example to show how the data distribution server relies on the S1-2-3.3 decision - optimal decision for Grep - in Table 5 to distribute data to the five nodes of the tested cluster. Recall that the computing ratios of Grep on the 5-node Hadoop cluster are 1, 2, 3.3, 3.3, and 3.3 for nodes A-E (see Table 4). We suppose there are 24 fragments of the input file for Grep. Thus, the data distribution server allocates 10 fragments to node A, 5 fragments to node B, and 3 fragments to nodes C-E.

Fig. 12 reveals the impacts of data placement on the response times of the Grep application. The first (leftmost) bar in Fig. 12 shows the response time of the Grep application by distributing file fragments based on Grep's computing ratios. For comparison purpose, the other bars in Fig. 12 show the response time of Grep on the 5-node cluster with the other five data-placement decisions. For example, the third bar in Fig. 12 is the response time of Grep when all the input file fragments are evenly distributed across the five nodes in the

cluster. We observe from Fig. 12 that the first data placement decision (denoted as S1-2-3.3) leads to the best performance of Grep, because the input file fragments are distributed strictly according to the nodes' computing ratios. If the file fragments are placed using the "All-in-C" data-placement decision, Grep performs extremely poorly. Grep's response time is unacceptably long under the "All-in-C" decision, because all the input file fragments are placed on node C - one of the slowest node in the cluster. Under the "All-in-C" data placement decision, the fast nodes (i.e., nodes A and B) have to pay extra overhead to copy a significant amount of data from node C before processing the input data locally. Compared with the "All-in-C" decision, the optimal data placement decision reduces the response time of Grep by more than 33.1%.

Fig. 13 depicts the impacts of data placement decisions on the response times of WordCount. The second bar in Fig. 13 demonstrates the response time of the WordCount application on the cluster under an optimal data placement decision. In this optimal data placement case, the input file fragments are distributed based on the computing ratios listed in Table 4. To illustrate performance improvement achieved by our new data placement strategy, we plotted the other five bars in Fig. 13 to show the response time of WordCount when the other five data-placement decisions are made and applied. Results plotted in Fig. 13 indicate that the response time of WordCount under the optimal "S1-2-5" data placement decision is the shortest compared with all the other five data placement decisions. For example, compared with the "All-in-C" decision, the optimal decision made by our strategy reduces the response time of WordCount by 10.2%. The "S1-2-5" data placement decision is proved to be the best, because this data placement decision is made based on the heterogeneity measurements - computing ratios in Table 4. Again, the "All-in-C" data placement decision leads to the worst performance of WordCount, because under the "All-in-C" decision the fast nodes have copy a significant amount of data from node C. Moving data from node C to other fast nodes introduces extra overhead.

In summary, the results reported in Figs. 12 and 13 show that our data placement scheme can improve the performance of Grep and Wordcount by up to 33.1% and 10.2% with averages of 17.3% and 7.1%.

2.3 An Offloading Framework for I/O Intensive Applications on clusters [16]

2.3.1 Motivations

Offloading techniques had been applied in a wide range of applications, however there is few research working on details of its design and implementation. In this paper, we proposed an offloading framework which is able to be easily applied to either an existing or a completely newly developed I/O intensive application with slight efforts. We also illustrate in details its design and implementation, including theory and key issues of developing an offloading application. The primary goal of our approach is to not only increase I/O performance, but remarkably reduce internal network traffic in clusters.

Two factors make our offloading framework desirable and practical:

- I/O inefficiency in data intensive applications, and
- heavy burden of data transmission on internal network of clusters

The inefficiency of I/O performance has gradually become a major bottleneck that although computing power of processors has rapidly increased, the speed of accessing data from storage systems, including both magnetic and optical media, does not grow as fast as expected. This problem would be even worse in high performance computing, especially for running a data intensive application. In addition, it is also a potential limitation on availability and scalability of entire systems. Thus, approaches of improving I/O performance play an essential role in large scale clusters.

Another factor by which we are motivated is network burden of data transmission. In a typical cluster, all data needed by applications running on computation nodes should be loaded from and stored to storage nodes through internal networks. Facing a similar condition with disks, the growth rate of network bandwidth is significantly less than speed of data explosion. Large amount of data transmission would remarkable increase data latency and decrease performance of entire systems. It would be even worse in Ethernet network, widely used in community clusters, that tens or hundreds of nodes have to compete with each other in local networks for a change of sending or receiving a small piece of data. Therefore, scarcity and sharing of network resource might be another possible thread to system availability and scalability.

2.3.2 Offloading Framework

In this section, we illustrate an offloading framework at first, and then discuss several essential issues.

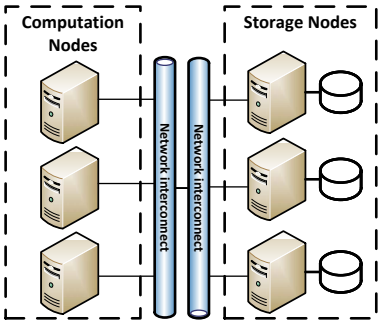


Figure 14: A typical architecture of cluster

Fig. 14 illustrates a cluster architecture which is accepted as a typical environment for cluster computing [13] [10]. The cluster comprises of a number of nodes which connect with each other by internal networks. These nodes are divided into two groups, storage nodes and computation nodes. The primary responsibility of storage nodes that are attached disks are to store massive data, while computation nodes mainly focus on computing tasks, including both CPU intensive and I/O intensive ones.

The storage system in this cluster is a Client/Server model that applications execute on computation nodes as clients and storage nodes run as servers. All data needed by applications are required to be transmitted back and forth on internal networks. Accessing data in this way would become an obviously serious bottleneck when amount of data transmission grows. Thus, it would achieve much better performance if it is able to efficiently reduce data transmission on network.

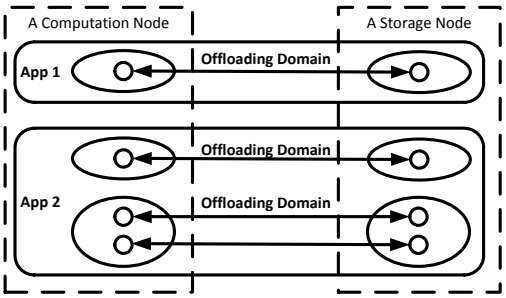


Figure 15: A framework of an offloading process, circles represent computation or offloading parts.

Our design is to assign parts of I/O intensive applications to storage nodes in order to minimize transmission requirement. As shown in Fig. 15, according to places where programs are executing, an application can be logically divided into two parts, the one running on computation nodes entitled computation part and the other running on storage nodes called offloading part.

An application is also able to be divided by offloading domains (logic processing units). An offloading domain describes a fully closed relationship between, or a pair of, computation and offloading parts. An application may have either only one offloading domain or multiple ones. How many offloading domains it has heavily depends on its design, more exactly the number of offloading modules. The offloading domains are independent to each other that one offloading domain can not be interfered by others. In addition, both computation and offloading parts in an offloading domain are strictly serially processed. In other word, while the computation part is running, the corresponding offloading part is suspended and vice versa.

Fig. 16 shows an offloading version of Parallel Word Counter (PWC) which calculates the number of words in a group of files in parallel. Function *PWC* and *domain_entry* are running as a computation part and *word_count* is running as an offloading part. We assume that each domain of PWC processes 2 text files.

In a case of processing 4 text files, PWC will create two threads in function *PWC*, each of which will create two offloading domains by serially invoking remote *word_count* twice.

2.3.3 Design Issues

Before developing an offloading application, there are three essential issues that should be considered carefully.

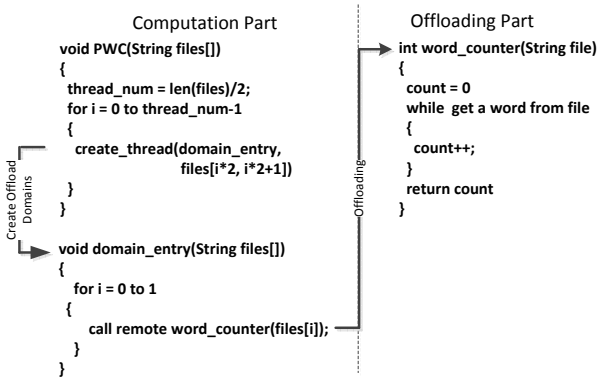


Figure 16: Pseudo code of offloading parallel word count

- How to offload a program to a specified node?
- How to transfer an execution to an offloading part?
- How to share data between computation and offloading parts?

Offloading a program The first issue needed to be taken into account is how to offload an executable file. Dynamic distribution and pre-configuration are the most widely used methods today. The main idea of dynamic distribution is to automatically transmit an executable file and configuration files to specified places and then load them into memory while an application is running. In this method, details of platform implementation cannot be ignored if applications are implemented by platform dependent languages, such as assemble language, C language and etc. While the ones using platform independent language, such as scripts or java, do not have any necessity to consider what platform they run on. Thus, how much efforts should be paid on dynamic distribution highly relies on the nature of applications.

In another method, called pre-configuration, all applications have been configured in advance. The entire procedure of configuration includes manually compiling applications for different environments, writing specific configuration files and deploying them onto target systems. Although it seems a complicated process that we have several steps to do, these tasks can be automatically completed by a small tool in a short period of time. Moreover, it greatly simplifies our design that we do not consider platform dependent issues at all. When an application starts, the proper offloading parts are already on storage nodes. That is the main reason we choose pre-configuration as the method of offloading a program.

Controlling an execution path The second issue is how to transmit executions back and forth between computation and offloading parts. we also have couples of language-independent candidates. CORBA [5] is a distributed programming model which is able to accommodate a number of components implemented by different languages. These components usually execute on different machines and communicate with each other through networks. However, its extremely complexity often prevents beginners from further using and learning. It normally requires at least several months for novices to get familiar with its

fundamentals [9]. Another factor we have to consider is that storage nodes might be required to be equipped with powerful processors in order to host incredibly complicated CORBA framework. Otherwise, processors would be always occupied by workload of CORBA routine jobs.

Another feasible option is Remote Procedure Call (RPC) technique, which is also a broadly accepted method of invoking a function to execute in another machine as if calling a local one. The main feature of RPC is that it is quite easy to learn and use. So far there are a lot of existing RPC libraries implemented by various general-purpose programming languages and freely opened to developers. Due to its simplicity, since RPC was applied to the first version of Network File System(NFS) [11], various well-known applications, such as MapReduce [6] and Hadoop [4], adopted it as a basis service provided by system as well. Therefore, RPC is a better choice for our framework.

Data sharing between both parts The third issue is how to share data, which includes global variables and code segments. The major difference between offloading applications from regular ones is that in offloading applications, global resources, such as global variables, can not be shared by computation and offloading parts. For example, any changes on global variables in one part is not visible to the other part.

An intuitive solution is to establish a synchronization mechanism of notifying the other end that global modifications occur. If computation parts modified shared data, they will immediately send a notification message to offloading parts and wait until receiving a reply. Although this solution is quite straightforward, complexity of applications would be largely increased and it may have much message exchanges when global data changes frequently.

Another solution is based on observation that offloading parts are in waiting state when computation part is running. If global data changes at computation parts in this period of time, offloading parts would not access it until it receives execution control. Therefore, it is not necessary to synchronize global modifications in real time. Instead, offloading parts can be notified later by appending modifications to offloading requests. It at first updates changes and then processes offloading requests. On the other hand, the changes that occur at offloading parts can be treated as results in response message as well.

Code segments are another special kind of data needed to be carefully considered. In applications implemented by compiled languages, due to that addresses of an function in these two parts may be different after loading into memory, function objects can not be shared directly. However, in interpreted applications, functions are parsed by names rather than addresses. So, both two parts are able to obtain identical functions by their names.

In this section, we only discuss reasons why data sharing is important to offloading applications. And our method of sharing data will be provided in 2.3.4.

2.3.4 Implementation Details

In this section, we describe details of implementing an offloading application and an entire process of running an offloading application on clusters.

Configuration As mentioned before, we adopt pre-configuration method to offload an application. We have a number of jobs to do before starting an offloading application.

The following steps describe an entire procedure of implementing and running an offloading program.

1 Design an offloading application and decide which one or more parts required to be offloaded.

2 Convert an original application to offloading version by using offloading programming interfaces discussed in 2.3.4. Developers may need to write configuration files if the application has.

3 Create executable files for different target nodes if they are implemented by compiled languages. While applications are developed by interpreted languages, the source files themselves are executable.

4 Copy proper files manually to specified directories on either computation and storage nodes.

5 Start offloading parts at first and then computation parts. The main reason of keeping in such order is that offloading parts have to be ready to provide offloading services before computation parts start.

Workflow of an offloading application In this section, we illustrate workflow of an offloading application. Normally, offloading parts can be distributed across multiple storage nodes. The places where they are depend on distribution policies. For example, a typical policy is to distribute offloading parts to the nodes where data is [6] [4]. Another policy of considering load balancing is to equally distribute offloading parts across storage nodes. Thus, computation parts have to decide which offloading part is about to be invoked according to a specific distribution policy. After an offloading part completes, it returns executions to the corresponding computation part.

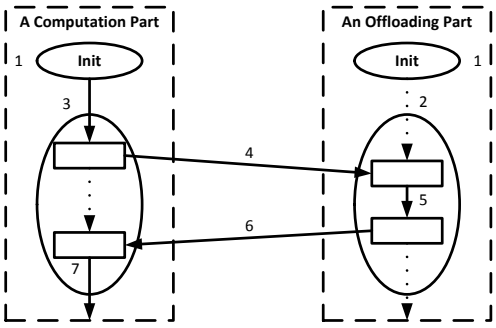


Figure 17: An execution flow of an offloading application

Fig 17 shows a workflow of an offloading application with a single offloading invocation. When an offloading application starts, the following 7 actions occur:

1 Firstly both computation and offloading parts initialize and prepare for their execution. The main tasks of this action is to distinguish which role they play in applications, offloading or computation.

2 After initialization, offloading parts will be suspended immediately and wait for offloading requests sent from computation parts.

3 After offloading parts are ready, computation parts start processing.

4 When running to the place where an offloading invocation is required, the computation part sends a request to an offloading part and wait for its reply. The request includes a network address of target node, a name of offloading entry and input parameters. Network addresses of storage nodes can be recorded in a configure file in order that they can be easily obtained. Names of offloading entries can be hard-coded in applications, just like calling a function in source files. All input parameters need to be transformed to a data stream in order to be transmitted on network.

5 After receiving a request, an offloading part will be activated to parse the request and start processing.

6 After completion, the offloading part sends a response back to the computation part. The response comprises of an network address of an computation node and results. As creating the request, network address of the computation part can be obtained from a configuration file and results require to be transformed to a data stream.

7 After receiving a response, the computation part continues processing.

Programming Interface The current implementation of offloading framework provides a group of programming interfaces for C and C++ languages. And it is also quite easy to define identical interfaces for other languages like java or python. It provides four sets of interfaces summarized in Table 6.

The second set of interfaces is used to register offloading entries. In C/C++ applications, offloading entries are addresses of functions in offloading parts. After compilation, all functions are converted into addresses that an identical function may have different values in computation and offloading parts. In order to exchange offloading entries between both parts, we provide a solution of assuming that applications would call `register_function` to register functions at first and then exchange function names instead of addresses. Addresses are automatically converted to names in computation parts and reverse in offloading parts by calling `find_name_by_func_addr` and `find_func_by_name`.

The third set is used to send and receive parameters and results from a data stream. Both `MARSHAL` and `UNMARSHAL` accept input parameter *object* in type of *void ** in order to adapt all types of objects. The following two parameters specify buffer of data stream and its length. All data exchanging between both parts must implement corresponding `MARSHAL` and `UNMARSHAL` functions which would be automatically called by system. If a function pointer need to be serialized or un-serialized, it has to be processed as a string after converting to its name by second set of interfaces.

sharing data In sec. 2.3.3, we have discussed complexity of offloading programs heavily depends on how to share data. we choose the easiest way of passing data as input and output parameters because we want to keep offloading programming simple. Two key aspects should

Table 6 Offloading Programming Interface

Interface & Description
`void init ()` Initialize the system.

`void register_function (func_addr)`
Register a function and build a map from its address to name.
`func_name find_name_by_func_addr (func_addr)`
Get a function name by a given address.
`func_addr find_func_by_name (func_name)`
Get a function address by a given name.

`void MARSHAL (void* obj, char**buf, int* len)`
Serialize an object pointed by obj into a data stream. The address and size of buffer are specified by buf and len.
`void UNMARSHAL (void* obj, char*buf, int len)`
Un-serialize an object pointed by obj from a data stream. The address and size of buffer are specified by buf and len.

`void offload_call (addr, func_name, ins, outs)` Invoke an offloading procedure named by func_name. The input parameter and result are specified by ins and outs.

be considered about sharing data. The first one is how to share global data between two parts. As mentioned before, all data needed by both parts should be passed by input parameters and result, which are required to be deeply copied in `MARSHAL` and `UNMARSHAL` instead of merely copying object points, because objects created in address spaces are totally different in two parts.

The second one is how to share code segments. Function entries or executable objects are a special kind of data in programs. We can not simply copy binary codes and transmit them to the other part, since they might be not runnable at all. So in our design, we link all object codes into each part, no matter whether codes are used or not. In order to transmit a function entry, we build a map between function names addresses and put function names in offloading requests or responses. Both parts can resolve function names and addresses by using programming interfaces.

2.3.5 Evaluations

In this section, we evaluate data intensive offloading applications, comparing them with original versions, on our cluster.

Testbed We set up a 2-node, one computation and storage node, cluster serving as a testbed to evaluate performance of offloading applications implemented by our offloading framework. Two nodes are connected by internal Ethernet network. Both two nodes have the same configuration as shown in Table 7.

Table 7 Configuration of Testbed

Configuration	Details
Hardware	1 × Intel Xeon X3430 2.4 GHz processor
	1 × 2GBytes of RAM
	1 × 1G Ethernet network card
	1 × 160 GBytes Sata disk
Software	Ubuntu 10.04
	Linux kernel 2.6.23

Benchmark Applications

2.3.6 Applications

We set up 5 benchmarks, shown in Table 8, which are well-known I/O intensive applications. PostgreSQL, Word Count(WC), Sort and Grep are obtained from their official website, while Inverted Index application is created by ourself. In our experiments, these original applications execute on computation nodes and load data from storage nodes through Network File System (NFS) service [11].

We also applied offloading techniques to these applications which has an offloading module assigned to be running on storage nodes. Details of their implementation are described in Table 8.

Data Preparation In order to measure PostgreSQL, we create five databases whose sizes are 400MB, 600MB, 800MB, 2GB and 4GB. We do not generate any index in these databases so that PostgreSQL will read real data in tables instead of merely checking index structure during query processing. Each database is comprised of 1, 000 tables, each of which has 100 integer attributes. Tuples are equally distributed across these tables that a larger database has more tuples in each table. Moreover, we also generate 1000 query statements, each of which scans only one table. Therefore, these 1000 query statements will cover all tables in a database.

We create five text files, in the same size of ones used in PostgreSQL, for other four applications as well. Each text file contains a number of words which are randomly generated. Due to the limitation of physical memory, we only test inverted index application on first

three text files because it will frequently cause page faults which makes a lot of noise in experiments when the size of input data is larger than memory.

Take PostgreSQL as an example We would like to briefly describe how official and offloading PostgreSQL work in our experiments. The reason of choosing PostgreSQL as an example is that it is a relatively more complicated application which has a number of independent modules. Moreover, boundaries of I/O intensive modules are highly distinguishable. It makes us easily partition PostgreSQL into computation and offloading parts.

2.3.7 Official PostgreSQL

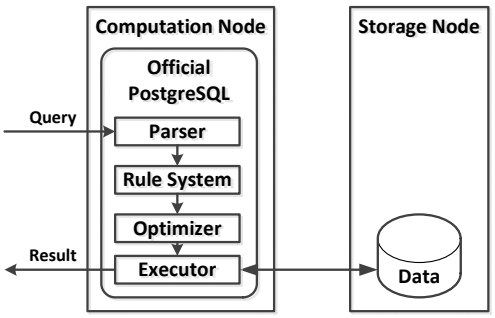


Figure 18: The execution flow of official postgresql

PostgreSQL is an well-known open source relational database management system which can be freely used and modified for research purpose. We choose the newest stable release, PostgreSQL 9.0, as a target application in these experiments.

As shown in Fig 18, PostgreSQL backend program, which mainly support SQL queries in background, has four components. The parser checks a query string for valid syntax and creates a parse tree after validation process. The rule system applies a group of rules to rewrite the parse tree to an execution plan. The optimizer tries to create an optimal execution plan and the executor runs the entire query [8].

Offloading PostgreSQL

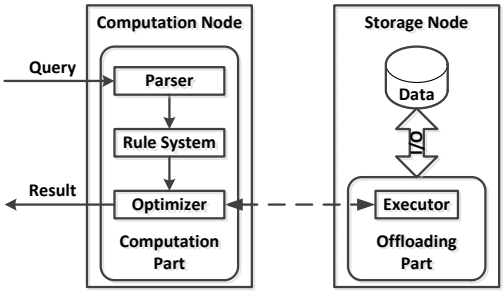


Figure 19: The execution flow of offloading postgresql

In a query procedure, the executor is a typical I/O intensive program. It may read or write large amount of data from storage system during processing expensive operations, such

as scanning or joining tables. As shown in Fig 19, we modified official PostgreSQL 9.0 by assigning the executor to be executed on storage nodes. We do not change modules related to storage system, such as access methods and disk space manager so that an offloading PostgreSQL is able to use the same data files. The only difference from official versions is that the executor receives the execution plan from a remote optimizer and sends results back to the backend program.

As discussion in Sec. 2.3.4, we link computation and offloading parts together as an executable program. And we copy it with meta-data files to both computation and storage nodes. Meta-data files, much smaller than real data, record information about databases and tables, such as schema and relationship between tables, which are required by both parts. We provide an additional command-line argument to distinguish whether it is an offloading program or not. These details will be handled by `init` interface.

Shared memory is the place where PostgreSQL shares global resources, such as locks and buffers. It is created by a Postmaster daemon and used by a number of backend programs. In our offloading PostgreSQL, the computation and offloading parts will create shared memory on their nodes separately and they do not shared any global resources for following two reasons. The first one is that we use only a client to do queries, so there is only one backend program existing in test environment. No other backend programs use shared memory at the same time. In addition, global resources, such as meta-data, shared by computation and offloading parts are read-only in our test, so it is not a problem that it has two copies, each of which exists in either part. For example, we do not change schema information which is loaded by both parts from their meta-data files.

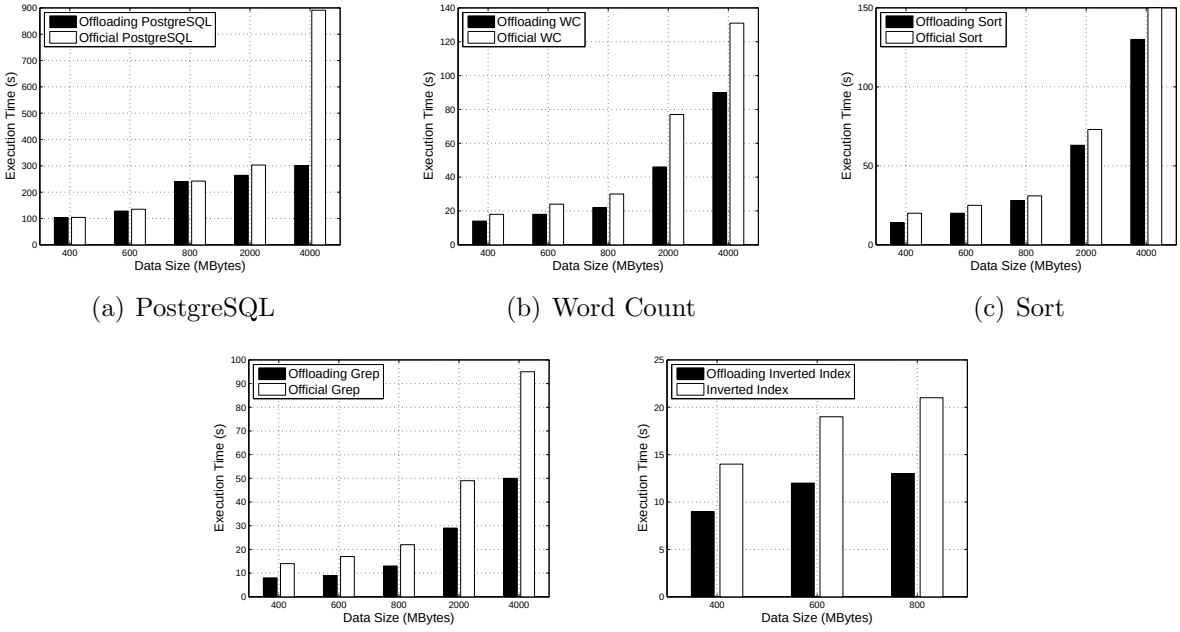
2.3.8 Results

In this section, we present experimental results of comparing offloading applications to their original versions.

Overall performance evaluation

Fig. 20 illustrates execution time comparison of offloading and original applications listed in Table 8. In these five groups of experiments, offloading technique provides speedup which becomes more obvious when data size grows. Due to network latency in each time of communication between nodes, official applications suffer from a large number of such latency when accessing remote data. However, offloading applications only experience once during processing in our experiments. Another reason that makes execution time reduce is that official applications need to transfer entire data which naturally increases as data set grows. As shown in Fig. 22, we will give more details in Sec. 2.3.8. On the other hand, such requirement for offloading applications keep relatively invariable because only input and output parameters of offloading modules are needed to be transmitted on network. Therefore, such difference becomes larger as data sets grow.

Another observation we obtain is that difference of time consumption between two PostgreSQLs on 4GB data set is much greater than others. The pattern of accessing data in PostgreSQL is totally different from other applications. WC, Sort, Grep and Inverted Index enjoy contiguous I/O operations in order that I/O costs can be optimized by NFS, such



(a) PostgreSQL (b) Word Count (c) Sort

(d) Grep (e) Inverted Index

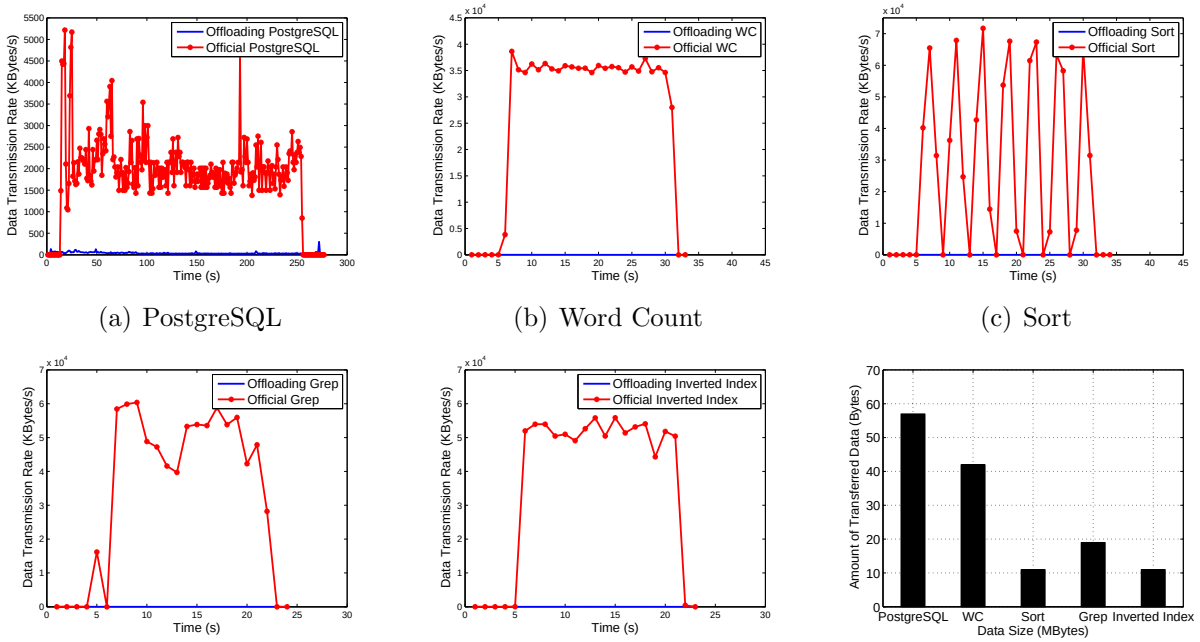
Figure 20: Execution Time Comparison of I/O Intensive Applications

as using prefetching technique. However, PostgreSQL reads data in relatively random way so that these optimization techniques do not work well. In addition, this difference is also much higher than its own measurement on 2GB data set. Although PostgreSQL uses shared memory to buffer data recently used, the size of buffers is controlled by itself. However, NFS service also caches remote data at local memory whose size can expand to more than 1GB. Therefore, when data size exceeds physical memory capacity, official PostgreSQL will suffer from much more frequent page faults that would dramatically decrease system performance.

Network Traffic Evaluation

Fig 21 shows network traffic comparison in our experiments. The network burdens of official applications are much heavier than offloading ones. When official applications are running, network traffic keeps consistently high which must become a major bottleneck of entire systems. On the other hand, network resources used by offloading applications can be nearly overlooked. In particular, all official applications have to retrieve entire data (eg., 800 MBytes) from storage nodes. But offloading applications only transmit not more than 100 bytes.

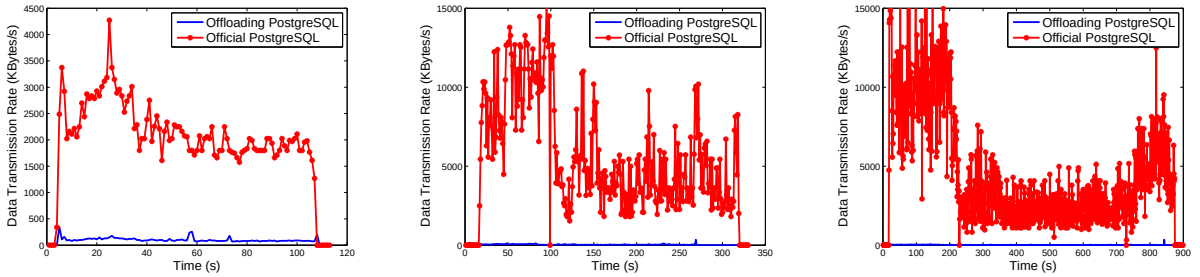
In fact, how much network resources they use heavily depends on application designs, or more exactly the size of input and output parameters of offloading modules. Fig 21(f) shows different requirements for these five offloading applications. Offloading PostgreSQL requires transferring 57 bytes of data that contains an internal query plan while Sort and Inverted Index only transfer 12 bytes, a string of a file name. In an extreme case, offloading PostgreSQL also has to transmit 800 MBytes data on network if a query is to retrieve an entire database.



(a) PostgreSQL (b) Word Count (c) Sort

(d) Grep (e) Inverted Index (f) Amount of Transferred Data Comparison

Figure 21: Network Traffic Comparison of I/O Intensive Applications on 800 MBytes Data sets



(a) 400 MBytes (b) 2 GBytes (c) 4 GBytes

Figure 22: Network Traffic Comparison of PostgreSQL on different databases

The pattern of network traffic in Sort application is different from others. As shown in Fig 21(c), the rate of data transmission in Sort keeps high in very short period of time, while it keeps consistently high in others during processing. The main reason is that after reading a certain amount of data, Sort is required to keep texts in order which takes processors a short period to complete. At these short intervals, network devices are waiting for next I/O requests. However, other four applications do not have such complicated tasks so that network devices are very busy.

Dependency on Data Size

We also evaluate network burden of PostgreSQLs on different data sets. Fig. 22 and

Fig. 21(a) respectively displays network measurement in 400 MBytes, 800 MBytes, 2 GBytes and 4 GBytes. We observe that increasing data size leads to a higher and longer lasting network burden in official applications. The main reason is that, as data sets grow, official PostgreSQL needs retrieve more data from storage nodes which spend more time and resources on data transmission. On the other hand, we do not observe much network traffic occurred when running offloading PostgreSQL. It offers an evidence that amount of data transmission in offloading PostgreSQL is independent of data sizes.

2.4 Using Active Storage to Improve the Bioinformatics Application Performance: A Case Study

2.4.1 Motivations

Processing massive amounts of data has resulted in a mushrooming of data-intensive applications like bioinformatics data processing. Evidence shows that the collective amount of genomic information doubles every 12 months [**?**]. Most bioinformatics applications have to deal with the I/O bottleneck issue. Processing huge datasets in a high-performance cluster normally requires copying data from storage nodes to computing nodes, thereby leading to a large number of I/O operations. Active storage is an effective technique to improve applications' end-to-end performance by offloading data processing from computing nodes to storage nodes.

Active storage brings three key advantages. First, the amount of data moved back and forth between computing nodes and storage nodes in clusters can be significantly reduced, since large datasets can be locally processed by storage nodes before being forwarded to computing nodes. Second, data-intensive applications run faster, because active storage nodes accelerate data processing operations. If computing nodes and active storage nodes efficiently coordinate, both computing nodes and storage nodes can perform data processing in parallel. Third, network performance in clusters can be improved due to reduced amounts of data moved into and out of storage nodes.

Challenges: There are two main challenges in applying active storage to support data-intensive applications on clusters. The first challenge is to partition a parallel application into computation-intensive and data-intensive tasks. If such a partition is successfully created, computing nodes will handle computation-intensive tasks whereas active storage nodes will run data-intensive tasks.

The second challenge lies in the coordination between computing nodes and active storage nodes. When it comes to applications where computation-intensive tasks are independent of data-intensive tasks, computing nodes and active storage nodes are non-blocking to each other, meaning that computing and storage nodes can easily operate in parallel. However, if computing nodes have to wait for storage nodes to catch up, the blocked computing nodes could slow down data-intensive applications.

Contributions: In this task, we address the partitioning and synchronization issues in the context of active storage supporting bioinformatic applications. To solve the blocking problem incurred by synchronized computing and storage nodes, we developed a pipelining

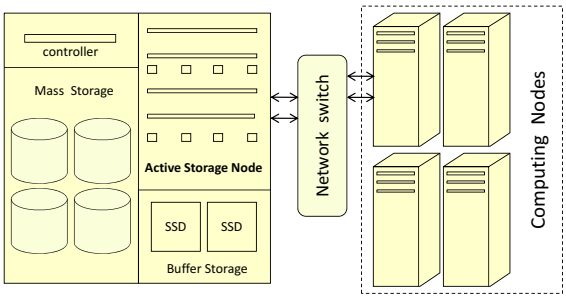


Figure 23: A cluster involves a collection of computing nodes and active storage nodes.

mechanism that exploits parallelism among data processing transactions in a sequential transaction stream. We report the effectiveness of the pipelining mechanism that leverage active storage to maximize throughput of data-intensive applications on a high-performance cluster.

To demonstrate the effectiveness of the pipelining mechanism designed for active storage, we implemented a pipelined application called pp-mpiBLAST, which extends mpiBLAST, which is an open-source parallel BLAST tool. pp-mpiBLAST deals with a sequential data processing transactions, each of which contains a filtering/formatting task and a mpiBLAST task. The mechanism overlaps data filtering/formatting in active storage with parallel BLAST computations in computing nodes, thereby allowing clusters and their active storage nodes to perform data processing in parallel. The pp-mpiBLAST application relies on active storage to filter unnecessary data and to format databases, which are then forwarded to the cluster running mpiBLAST.

We develop an analytic model to study the scalability of pp-mpiBLAST on large-scale clusters. This model allows us to study the performance of pp-mpiBLAST on a cluster using active storage. This performance model is ideal for application developers who have limited computing resources to test the scalability of their parallel applications using active storage. Furthermore, programmers can use the analytic model to explore the design space related to the number of computing nodes, active storage speed, data processing capacity, and input data size. The model shows the behavior of pp-mpiBLAST under different configurations of the cluster coupled with active storage.

Measurements made from a working implementation and a modeling study suggest that this method not only improves mpiBLAST's overall performance by up to 75%, but also achieves high scalability on clusters coupled with active storage.

2.4.2 Design and Implementation

In this section, we describe the system from the top down: an overview of the system, a hybrid mix of storage devices, and the parallel pipelined processing. Hereinafter, the active storage node is refereed to as ASN for short.

Active Storage for Clusters: A typical high-performance cluster consists of computing nodes and storage nodes. Data-intensive applications on clusters can cause heavy I/O traffic between the computing and storage nodes in the cluster. To achieve high performance for data-intensive applications, we aim to reduce network traffic caused by moving massive amounts of data from/to storage systems in clusters. This goal can be accomplished by offloading data-intensive computations from computing nodes to active storage nodes.

Fig. 23 illustrates a cluster that involves a collection of computing nodes coupled with active storage nodes. Storage nodes become active if they can handle application-level data processing offloaded from computing nodes.

The storage devices can be further divided into two categories: the mass storage and the buffer storage. Benefiting from the non-volatile memory store, Solid state disk (SSD) is a new option to fill the latency gap, which is around 5-order-of-magnitude, between main memory and spinning disks [**?**]. Thus in our system, SSDs are used as the buffer disk drives: large-scale data is moved from mass storage to buffer drives before processing. The results of experiments in Section 2.4.5 show that SSDs not only speed up the I/O but also provide a better scalability performance. The advantage of using the hybrid mix of both the solid state disk and the magnetic hard disk is mutual complementarity: the fast and expensive cooperates with the large-capacity and cost-efficiency.

In this paper, we use a commodity computer as the computing end. The "channel" formed by the computing nodes of cluster and the ASN is considered as a pipeline, or assembly line. In other words, by applying the active storage, applications containing multiple stages are capable to extend to a parallel pipelined implementation. The exploration of parallelism improve the performance data-intensive applications. As a case study, we extend the mpiBlast, a well-know parallel BLAST application, to a parallel pipelined implementation, called pp-mpiBlast.

Parallel Pipelined System:

Native mpiBlast application can be easily considered as two steps: format the raw database file (corresponding to the query request) and run the parallel BLAST functions to do the comparison. Thus, the pre-cook (*i.e. format*) phase and the parallel computation phase are handled by the ASN and computing nodes, respectively.

How to utilize the active storage device has always been another critical issue. In general, all cases can fall into two scenarios. The first is that tasks are independent (*i.e.* computing nodes and active storage nodes are non-blocking to each other), meaning that computing and storage nodes can easily can operate in parallel. However, if computing nodes have to wait for storage nodes to catch up, the blocked computing nodes can slow down data-intensive applications. For instance, in terms of mpiBlast, the parallel comparison step requires the formated database file from the previous step. The assembly line pattern is a one of the solutions for the second scenario.

As a case study, we extend the mpiBlast to a parallel pipeline implementation (hereinafter refereed to as pp-mpiBlast). The pp-mpiBlast system consists two tasks: 1) raw database formating, and 2) genome or protein sequences comparison. Further subdividing the pipeline patterns, there are inter- and intra-application pipeline processing. The pp-mpiBlast is intra-application parallel processing, which means that, as the name - 'intra-' - suggests, one native

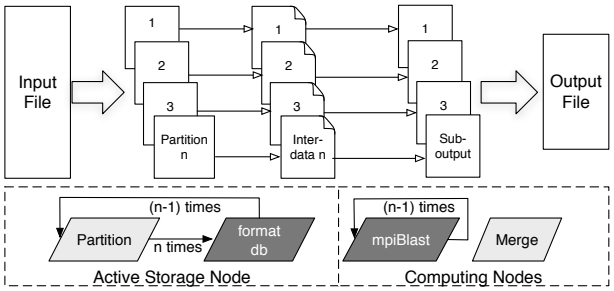


Figure 24: pp-mpiBlast Workflow

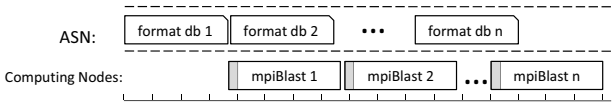


Figure 25: Pipeline Tasks Scheduling

sequential transaction is partitioned into multiple parallel pipelined transactions. The system performance is improved by fully exploiting the parallelism. The workflow of pp-mpiBlast is depicted in Fig. 24.

Intra-application Pipeline Processing: As we mentioned in the previous section, in order to extend a sequential transaction to multiple pipelined parallel transactions, both partition and merge functions are introduced in the pp-mpiBlast system. Fig. 25 illustrates the two-task two-stage pipeline processing workflow. The pipeline pattern no only improves the performance by exploiting the parallelism, but also can solve the out-of-core processing issue, which means required amount of data are too large to fit in the ASN's main memory. In pp-mpiBlast, partition function is implemented within *mpiformatdb* fucntion running on ASN. And the merge function is a separate one running on the front node of the cluster.

When partitioning the source data, an assistant function - the integrity-check - automatically returns the extra displacements by scanning the return or the symbol defined by the programmer. The reason we involved the integrity-check procedure to the partition function is that there exists the consistency issue of partitioned data files; the content of the source data file could be broken in shatters (e.g. a sequence could be cut and placed into two slitted fragments not on purpose). Fig. 26 describes the integrity-check work flow.

Pipeline parallelism is an important processing pattern and we are interested in providing models and guidance for tuning the scalability and the performance using this pattern. In Section 2.4.4, we develop a mathematics model for analysis.

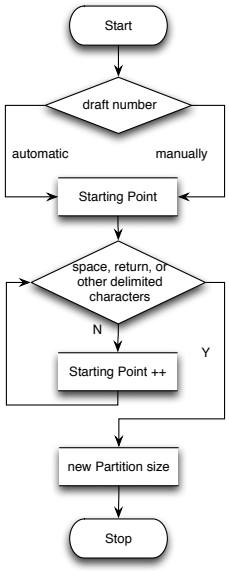


Figure 26: The workflow diagram of sequence-integrity checking.

2.4.3 Preliminary Result

In order to prove the feasibility of the partition-based intra-application pipeline design, a preliminary test on a single-node with 2GB memory environment is performed. We extended Word-Count and String-Match benchmarks of Phoenix system [**?**], which is a shared-memory implementation of MapReduce, to intra-application pipeline editions using partition and merge runtime functions we developed under Phoenix system. Fig. 27 depicts the work flow of the extended approach. After the input data is partitioned into fragments in size of 600 MB, each of them is processed sequentially using native word-count or string-match applications. And then, the generated sub-results are merged at the end. The control test is that we run the native applications to process the input data without the partition and merge functions.

Table 9 shows the results. We can observe that in terms of data-intensive, especially memory-intensive, applications, partitioning can significantly reduce the running time. For example, in terms of word-count application results, an average 2.4X speedup of time consumption can be achieved. To the contrary, partitioning does not benefit the speed of the string-match application. But, it can make the large-scale data-intensive applications running on limited memory machines. For example, when executing the string-match application without the help of partition function, the native system does not support the case that the input data size is more than two times of the local memory size. Thus, the preliminary results show that the partition-enabled design can (1) improve data-intensive applications' performance, (2) adapt the data-intensive applications to limited memory machines, or both.

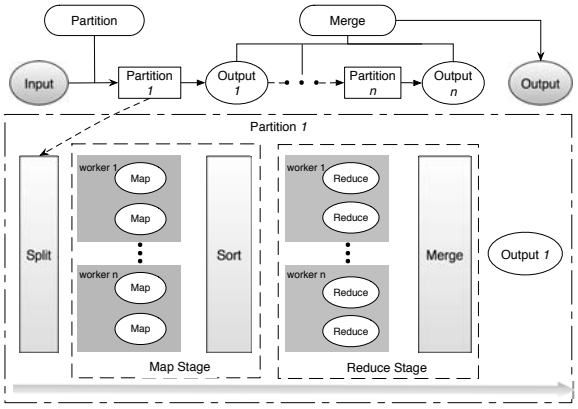


Figure 27: Workflow of the extended Phoenix model - intra-application pipeline

2.4.4 Modeling and Analysis

We develop in this section an analytic model to study performance and scalability of the parallel pipelined BLAST on large-scale cluster computing platforms. The analytic model has the following three key advantages:

- **Performance Evaluation**: The model allows us to study the performance (i.e., speedup and throughput) of parallel pipelined BLAST system with active storage.
- **Scalability Analysis**: The performance model is ideal and suitable for bioinformatics application developers who have limited local computing resources to test the scalability of their parallel applications using active storage.
- **Design-Space Exploration**: Application programmers can use the model to explore the design space related to the number of computing nodes, active storage speed, data processing capacity, and input data size. The model shows the behavior of the parallel pipelined BLAST under different configurations of a high-performance cluster coupled with active storage.

Section 2.4.2 describes the parallel pipelined implementation of a basic local alignment search tool or BLAST. Our pipelined BLAST has two data processing stages: (1) formatting data and (2) retrieving and processing formatted data. The first stage of the pipeline preprocess is the input of a data set before passing on to the second stage that run mpiBLAST - a parallel implementation of BLAST.

Response time, speedup, and throughput are three critical performance measures for the pipelined BLAST. Denoting T_1 and T_2 as the execution times associated with the first stage and second stage in the pipeline, we can calculate the response time $T_response$ for processing each input data set as the sum of T_1 and T_2. Thus, we have

$$T_{response} = T_1 + T_2. \tag{2.1}$$

The throughput (see Eq. 2.2) of the two-stage pipelined system is inversely proportional to the maximum of the two execution times T_1 and T_2.

$$Throughput = \frac{1}{max\,(T_1, T_2)}. \tag{2.2}$$

The speedup for the pipelined BLAST is:

$$Speedup = \frac{T_{unpipelined}}{T_{pipelined}}, \tag{2.3}$$

where $T_{unpipelined}$ is the data processing time for the unpipelined BLAST and $T_{pipelined}$ is the processing time of the pipelined BLAST. If n is the number of input data sets to be processed by a cluster with active storage, then processing time of the unpipelined BLAST is the product of n and $T_response$ (see Eq. 2.1), leading to

$$T_{unpipelined} = n \times T_{response} = n \times (T_1 + T_2). \tag{2.4}$$

The processing time for the pipelined BLAST is:

$$\begin{aligned} T_{pipelined} &= T_1 + (n-1) \times max\,(T_1, T_2) + T_2 \\ &= T_{response} + (n-1) \times max\,(T_1, T_2)\,, \end{aligned} \tag{2.5}$$

where T_1 is the processing time of stage 1 for the first data set, T_2 is the execution time of stage 2 for nth data set, $(n-1) \times max(T_1, T_2)$ is the time spent on $n-1$ data sets when the two stages are carried out is parallel. Applying Eqs. 2.4 and 2.5 to Eq. 2.3, we obtain speedup as:

$$Speedup = \frac{n}{1 + (n-1) \times \frac{max(T_1, T_2)}{T_{response}}}. \tag{2.6}$$

Now we are positioned to model execution times T_1 and T_2 for the two stages in the pipeline. The processing time of the first stage is the sum of (1) data input/output times and (2) filtering/formatting time $T_{1,comp}$. Input time is proportional to unformatted input data size s_u and inversely proportional to disk read bandwidth b_i. Similarly, output time is proportional to formatted output data size s_f and inversely proportional to disk write bandwidth b_o. This leads to:

$$T_1(s_u, s_f, b_i, b_o) = \frac{s_u}{b_i} + T_{1,comp}(s_u) + \frac{s_f}{b_o}. \tag{2.7}$$

The execution time T_2 of stage two is the sum of (1) input time of formatted data and (2) processing time $T_{2,comp}$. The input time depends on data size s_f, disk input bandwidth b_i, and the number m of computing nodes in a cluster. Assuming that the formatted data size s_f is uniformed distributed among the m computing node, we can express the input data as $\frac{s_f}{m \times b_i}$. Thus, the execution time T_2 for the second stage is given below:

$$T_2(s_f, b_i, m) = \frac{s_u}{m \times b_i} + T_{2,comp}(s_f, m), \tag{2.8}$$

where $T_{2,comp}(s_f, m)$ is affected by the formatted data size s_f and the cluster size (i.e., number of computing nodes m).

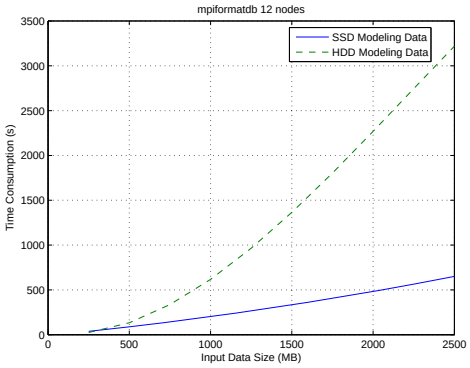


Figure 28: Time Consumption Trends Comparison: SSD vs. HDD.

2.4.5 Evaluations

Evaluation Environment:

We implemented the pp-mpiBlast in a 14-node cluster (1 node works as ASN), whose configuration is outlined in Table 10. The nodes in the cluster are connected by Ethernet adapters, Ethernet cables, and one 1Gbit switch. We choose an Intel X-25M 80GB solid state disk, and a SATA Raid tower with four WD50000AAKS disks as a RAID 0 array. The *MPICH2-1.0.7* is chosen as the message passing interface (MPI) in the cluster. The pp-mpiBlast is extended from mpiBlast-1.5.0, in which NCBI Blast 2.2.24 is the comparison tool. All applications are compiled with *gcc 4.4.1*.

Individual Node Evaluation:

We perform *mpiformatdb* program under different storage disk schemes: w/ SSD as the buffer disk and w/o buffer disk. Since the data pre-fetching is out of the scope of this paper, when we test the SSD cases, we modify the program to move the data from mass storage devices to the SSD and then trigger the format function. That means the time consumption of SSD contains both the data-transfer and data-format phases. Table 11 shows the results. Observed from the table, the case of SSD (*e.g.*the second row) always perform better than the HDD since it benefits from large amount of random read and write when the function is reordering the sequence in a descending order based on entry length. After balancing the disk capacity, storage reliability, I/O speed, and random w/r speed, the hybrid mix of mass storage and buffer disk is a promising choice. Fig. 28 shows the comparison of trends of using SSD and HDD. We can observe that the scalability of using HDD for mpiformatdb function is not good, compared with the SSD one. Thus in the following experiments, we use the SSD as the buffer disk.

System Performance Evaluation:

Figure 29 shows the system performance evaluation. The pp-mpiBlast testbed, which is configured by 12 computing nodes and 1 ASN (follows the pipelined processing pattern), is compared with two control experiments (native systems with different number of nodes) : the native mpiBlast running on 1) 12 nodes cluster (equals to the number of computing nodes in pp-mpiBlast testbed), and 2) 13 nodes cluster (equals to the total number of nodes

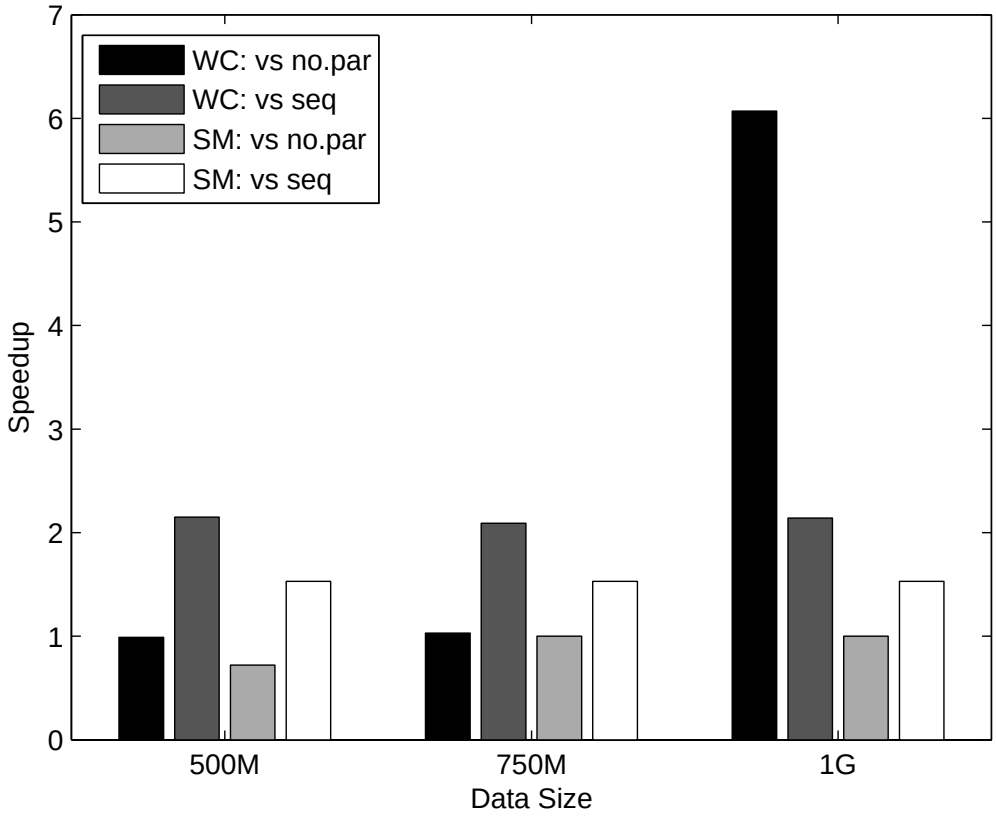


Figure 29: System Evaluation Results I: Execution time comparison between pp-mpiBlast system and native system (running on a computer cluster of 12 nodes). The pp-mpiBlast system contains a twelve nodes computing cluster and one ASN. Results are generated under 3 different partition sizes: 250 MB, 500 MB, and 1.25 GB, which are presented by three sub-figures, respectively from left to right.

in pp-mpiBlast testbed). The reason to choose two competitors is to present the performance improvement comprehensively.

In Fig.29 and Fig.30, the testbed with native mpiBlast contains 12 and 13 nodes, respectively. In each figure, results generated by pp-mpiBlast are using 3 different partition sizes: 250 MB, 500 MB, and 750 MB, which are presented by three sub-figures, respectively from left to right. Observed from the figures, the time consumption comparison results show that the performance of pp-mpiBlast beat both control testbeds: averagely reducing the execution time by 50% (i.e. 2X speedup). And Fig31 shows that: the improvement is greater when the input data size increase within a certain range of the input size, which is relative to the main memory size.

We also compare the performance in terms of different partition size. Fig. 32 presents that 500 MB partition provides a better performance in general. Based on the data, the

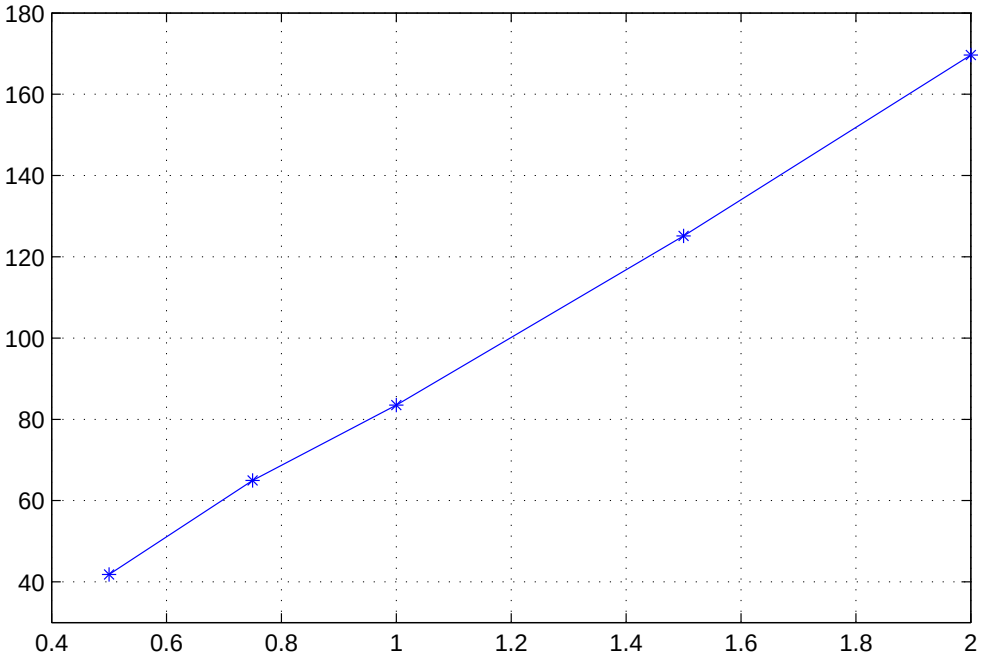


Figure 30: System Evaluation Results II: Execution time comparison between pp-mpiBlast system and native system (running on a computer cluster of 13 nodes). The pp-mpiBlast system contains a twelve nodes computing cluster and one ASN. Results are generated under 3 different partition sizes: 250 MB, 500 MB, and 1.25 GB, which are presented by three sub-figures, respectively from left to right.

reason can be summarized as followings. 1) The smaller the better is not true because small partitions always generate more overheads. And 2) the larger the better is also not convinced since the . It means that a partition-size-threshold exists for optimal performance. The issue of measuring the threshold in quantity will be dug in our future work.

Based on the test results, we can see that using intra-application pipeline parallel processing model to extend mpiBlast improves the performance and scalability. However, as we mentioned in the previous sections, the approach is not general; it requires that the target application can be decomposed into stages, such as streaming and RMS applications. Also, applications parallelized using pipeline model are very sensitive to load balancing. In order to avoid bubbles or reduce their sideeffect, how to balance the heterogeneity issue between the ASN and computing nodes will be our next topic.

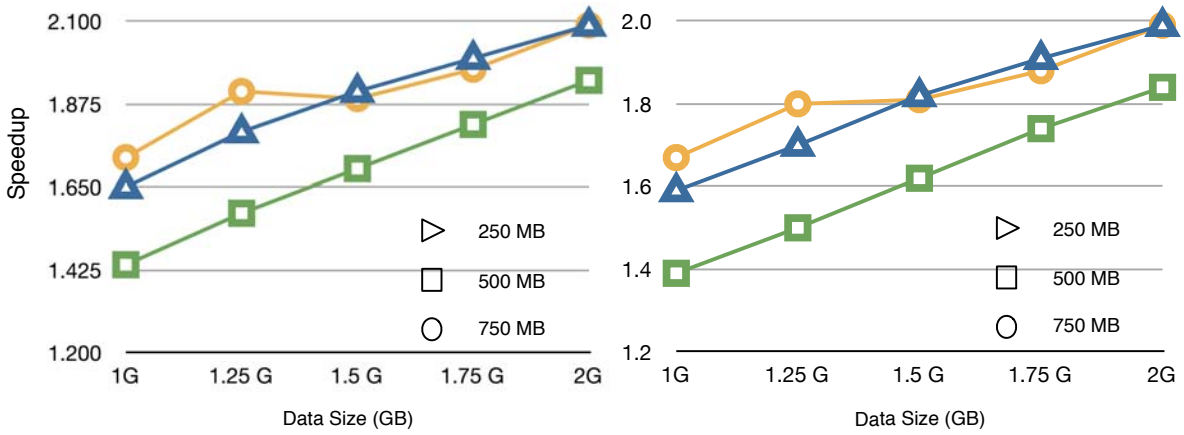


Figure 31: Speedup Trends: As input data size grows larger, the performance speedups of using pp-mpiBlast increase. Sub-figure on left is the comparison result between pp-mpiBlast and the 12-node testbed. And the right one is the result compared with the 13-node testbed.

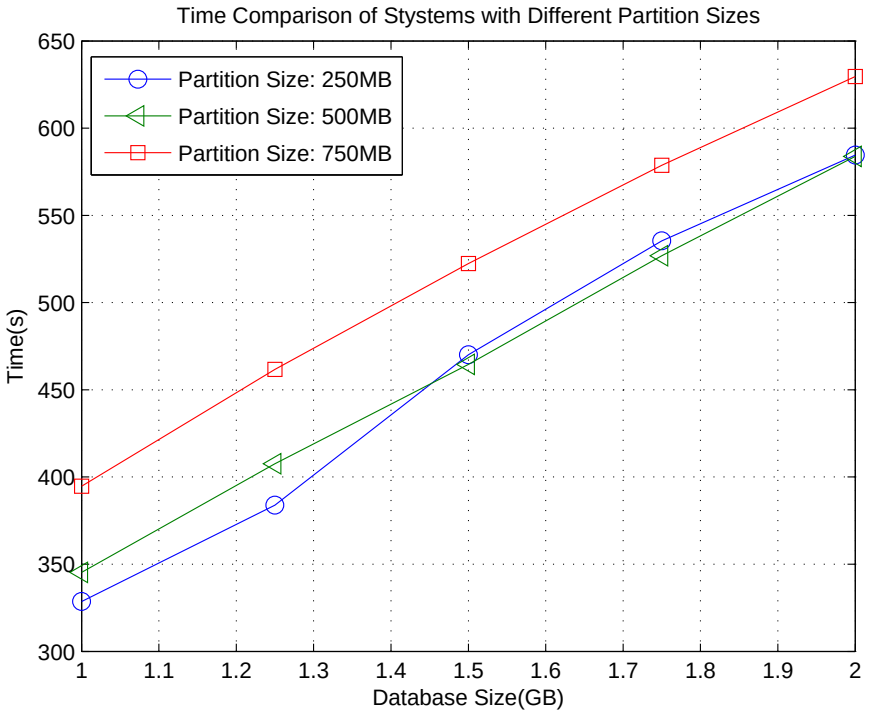


Figure 32: Time Consumption Curves Comparison: Different Partition Size.

3 Training and Development

3.1 Student Support

This project has directly supported about 8 students including 6 doctoral students and 2 undergraduate students. The project also indirectly contributed to approximately 17 graduate students who took the COMP7500 (i.e., Advanced Operating systems) class and 65 undergraduate students.

The following 6 doctoral students have been partially supported by this NSF grant in year 1 (i.e., 2009-2010):

- Adam Manzanares
- Jiong Xie
- James Majors
- Zhiyang Ding
- Xiaojun Ruan
- Shu Yin

The following 9 doctoral/master's students have been partially supported by this NSF grant in year 2 (i.e., 2010-2011):

- Xiaojun Ruan (Doctoral Student)
- Zhiyang Ding (Doctoral Student)
- Shu Yin (Doctoral Student)
- Jiong Xie (Doctoral Student)
- Maen Al Assaf (Doctoral Student)
- Yixian Yang (Doctoral Student)
- Yun Tian (Doctoral Student)
- James Majors (Master's Student)
- Jianguo Lu (Master's Student)

The following 2 undergraduate students have been partially supported by this NSF grant in year 1 (i.e., 2009-2010):

- Joshua Lewis

Figure 33: Our undergraduate research assistants helped in building a cluster system.

- Tsukasa Ogihara

  The following 4 undergraduate students have been partially supported by this NSF grant in year 2 (i.e., 2010-201a):

- Joshua Lewis
- Alfred Nelson
- Drew Pitchford
- John Barton
- Greg Poirier
- Alexander Luchs
- Bryant Haley
- Kathryn Catlett
- Riley Spahn

Figure 34: The cluster computing system built by our undergraduate research assistants.

3.2 Research Experience for Undergraduate Students

To recruit new undergraduate students, especially women and minorities, to conduct research in the area of storage systems, we designed a research program that offers ample opportunity to undergraduate students to do intensive research in data-intensive computing with the PIs. In particular, students and the PIs are brought together to conduct research experiments in the field of high-performance storage systems. The photo below shows two undergraduate students - Tsukasa Ogihara (right) and Joshua Lewis (middle) - are building a cluster computing system using commodity-off-the-shelf (COTS) hardware components.

The cluster system (see Figs. 33 and 34) built by our undergraduate research assistants will be used as a high-performance computing platform to support our computer security education. The cluster recently build in our department at Auburn supports security middleware services for secure software applications. We will use this cluster computing platform to design and implement study how to improve software application's quality-of-security without adversely affecting performance.

3.3 Contributions to Courses

This project has directly and indirectly contributed to the following classes:

- COMP7970: Storage Systems
- COMP7500 Advanced Operating Systems

- COMP4370: Computer and Network Security
- COMP2710 Software Construction
- COMP4300: Computer Architecture
- COMP7370: Advance Computer and Network Security

4 Outreach Activities

The outreach activities include curriculum enrichment presentations, engineering clubs, and tutorial services. In year 1 (i.e., 2009-2010), the PI had given 7 research talks related to this NSF funded project:

- Improving Energy-Efficiency and Reliability of Storage Systems, Seminar talk at the University of New Orleans, Sept. 4, 2009.
- Can We Improve Energy Efficiency of Secure Disk Systems without Modifying Security Mechanisms? the IEEE NAS09 Conference, ZhangJiaJie, China, July 10, 2009.
- Security-Aware Scheduling for Real-Time Parallel Applications on Clusters, Lecture at Huazhong University of Science and Technology, Wuhan, Hubei, China. June 22, 2009.
- How to Read Papers? Seminar talk at Wuhan National Laboratory for Optoelectronics, Wuhan, China, June 17, 2009.
- Energy Efficient Scheduling for High-Performance Clusters, Seminar talk at Huazhong University of Science and Technology, Wuhan, Hubei, China. June 8, 2009.
- An Overview of Auburn University, Seminar talk at Nanjing University of Information Science and Technology, Nanjing, China, June 3, 2009.
- Thinking About Going to Graduate School? Seminar talk at Nanjing University of Information Science and Technology, Nanjing, China, June 3, 2009.

The PI gave a talk on high-performance clusters at Taiyuan University of Science and Technology, China (see Figure 35). More than 30 faculty members and 300 undergraduate and graduate students attend the PIs seminar focusing on energy conservation techniques.

On April 23rd, Dr. Jiang with the University of Nebraska-Lincoln and two of his doctoral student visited the PI's new storage systems laboratory at Auburn University (see Figure 36). Dr. Dan Feng along with her two doctoral students also visited the PI's research group. Seven doctoral student from the PI's research group gave presentations, reporting their new findings from the projects supported by the U.S. National Science Foundation. Drs. Jiang and Feng provided insightful suggestions and comments on the research projects led by the PI at Auburn University.

Figure 35: The PI visited Taiyuan University of Science and Technology, China.

Figure 36: Drs. Hong Jiang and Dan Feng along with their doctoral students visited the PI's new storage systems laboratory at Auburn University

In year 2 (i.e., 2010-2011), the PI had given 7 research talks related to this NSF funded project:

- An Application-Oriented Approach for Computer Security Education, invited talk at the Information Security and Computer Applications (ISCA2011) Conference, Feb. 25, 2011.
- A Novel Application-Oriented Approach to Teaching Computer Security Courses. Poster Session at NSF CCLI/TUES Conference, January 27, 2011.
- Energy Efficient Prefetching From models to Implementation. Seminar talk at Huazhong University of Science and Technology, Wuhan, Hubei, China. June 2010.
- How to Read Papers? Training Session for REU students at Auburn University, May 18, 2010.
- How to Succeed in the AU REU Program? Training Session for REU students at Auburn University, May 17, 2010.

References

[1] Gnu core utilities. http://www.gnu.org/software/coreutils/.

[2] Gnu grep. http://www.gnu.org/software/grep/.

[3] Postgresql. http://www.postgresql.org/.

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

[5] Corba. http://www.corba.org/, 2010.

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

[7] Zhiyang Ding, Xiaojun Ruan, Jiong Xie, Shu Yin, Yu Tian, Xiao Qin, and Kai H. Chang. Multicore-embedded smart disks. In *Technical Report No. 1003, Department of Computer Science and Software Engineering, Auburn University*, Auburn, AL, USA, 2010.

[8] The PostgreSQL Gloabal Development Group. Postgresql developer's guide. http://www.postgresql.org/docs/9.0/interactive/index.html.

[9] M. Henning. The rise and fall of corba. *Queue*, 4(5), 2006.

[10] III R. B. Ross P. H. Carns, W. B. Ligon and R. Thakur. Pvfs: a parallel file system for linux clusters. *Proceedings of the 4th annual Linux Showcase and Conference*, pages 28–28, 2000.

[11] S. Kleiman D. Walsh R. Sandberg, D. Goldberg and B. Lyon. Design and implementation or the sun network filesystem, 1985.

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

[13] P. Schwan. Lustre: Building a file system for 1000-node clusters. *In Proceedings of the 2003 Linux Symposium*, 2003.

[14] Haiying Shen and Yingwu Zhu. A proactive low-overhead file replication scheme for structured p2p content delivery networks. *J. Parallel Distrib. Comput.*, 69(5):429–440, 2009.

[15] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Proceedings of the 2010 Int'l Heterogeneous Computing Workshop*, Atlanta, GA, USA, 2010.

[16] Ji Zhang, Xiaojun Ruan, Jiong Xie, Shu Yin, Yu Tian, Zhiyang Ding, and Xiao Qin. An offloading framework for i/o intensive applications on clusters. In *Technical Report No. 1104, Department of Computer Science and Software Engineering, Auburn University*, Auburn, AL, USA, 2011.

Table 8 Configuration of Testbed

Applications	Descriptions
PostgreSQL 9.0 [3]	It is a relational database management system and its offloading version is to offload executor module to storage nodes. The offloading part receives an execution plan and run real queries. The computation part manages connection to clients, transfer SQL statements to execution plan and sends results back to clients.
Word Count in GNU coreutils 7.4 [1]	It counts the number of words in a set of files. In an offloading Word Count, the offloading part is to calculate occurrence of words in one file. And the computation part sum them up.
Sort in GNU coreutils 7.4 [1]	It sorts lines of a text file in alphabetical order. In an offloading Sort, the entire program is treated as an offloading part which receives a file name and stores sorted text in a file.
GNU Grep 2.7 [2]	It searches through a file for lines which contains a given keyword. In an offloading Grep, the offloading part is to find keywords in the file. And the computation part only delivers keywords and input file name to the offloading part.
Inverted Index (our benchmark)	It loads a set of files and builds a map between words to their occurrence. The offloading part constructs a map for each file and computation part delivers file names to the offloading part.

Table 9: Running time (in seconds) of performing the Word-Count and String-Match benchmarks w/ and w/o partition function under different input data size (in GBytes) on single node. The testbed machine contains 2 GBytes main memory.

	WordCount (s)		**StringMatch (s)**	
	1 GB	**1.25 GB**	**1 GB**	**1.25 GB**
w/ partition	40.50	50.91	17.76	20.61
w/o partition	85.71	139.54	17.62	21.00

Table 10: The Test Platform

	Computing Nodes	ASN
CPU	Intel Xeon X3430	Intel Q9400
Memory	2GB	
OS	Ubuntu 9.04 Jaunty Jackalope 64bit version	
Kernel version	2.6.28-15-generic	
Network	1000Mbps	

Table 11: Time cost (in seconds) of performing mpiformatdb program under different input data size (in MB) on an ASN.

	Running time (seconds)					
Devices	500 MB	750 MB	1 GB	1.25 GB	1.5 GB	1.75 GB
HDD	108.4	369.5	639.1	945.6	1385.0	1845.1
SSD	101.9	164.4	225.9	291.1	369.5	441.1