Design and Evaluation of Distributed Smart Disk Architecture for I/O-Intensive Workloads

Steve Chiu, Wei-keng Liao, and Alok Choudhary

Department of Electrical and Computer Engineering,
Northwestern University, Evanston, IL 60208 U.S.A.
{schiu,wkliao,choudhar}@ece.northwestern.edu

Abstract. Smart disks, a type of processor-embedded active I/O devices, with their on-disk memory and network interface controller, can be viewed as processing elements with attached storage. The growing size and access patterns of today's large I/O-intensive applications require architectures whose processing power scales with the storage capacity. We evaluate a distributed smart disk architecture with representative I/O-intensive workloads including TPC-H queries, association rule mining, data clustering, and 2-D fast Fourier transform applications to study the proposed architecture.

1 Introduction

A hard disk drive typically consists of one or more disk platters, read/write heads, and other embedded general- or special-purpose processors. While much of the processing power on the disk is currently used for scheduling and optimizing disk bandwidth, the rapid advancements of processor and memory technologies promise to provide future hard disk drives with application-level processing capabilities to perform more complex operations directly at the drive [1]. For I/O-intensive applications that process large amount of disk-resident data, various storage architectures have been proposed to exploit the parallelism available from the large number of disks by offloading certain elementary operations to the disk drive. We study the benefits of a distributed smart disk (SD) architecture using TPC-H queries, association rule mining, clustering, and fast Fourier transform.

The remainder of this paper is organized as follows. Section 2 briefly reviews the work related to various SD architectures. In Section 3, we describe a distributed SD architecture in single- and multiple-disk configurations. Section 4 explains the algorithms and implementations of our workloads. Section 5 presents our experiments and results. Conclusions and future work are in Section 6.

2 Background

Five major factors catalyzed the evolution of storage architectures: I/O-bound workloads, improved disk drive attachment technologies, increased on-drive transistor density, emergence of new interconnects, and cost of storage systems [2]. Typically,

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2660, pp. 230–241, 2003.

a SD consists of an embedded processor, a controller, on-disk memory, local disk space, and a network interface. The embedded processor would have a speed of 300 – 500 MHz, and the memory size between 16 and 128 MB, while the disk space would be in tens of GB or more.

2.1 Database Machines and Parallel Databases

The concept of performing application-level processing on the disk originates from the research in database machines in the late 1970s. The limited disk bandwidth and the complexity of programming special-purpose hardware at that time eventually lead to the extinction of the database machines. Nevertheless, parallel database research has since produced efficient algorithms for large distributed-memory database servers that employ commodity processing elements [3]. This development and the growing processing power on the disk drive have prompted some researchers to suggesting that all application-level processing would eventually be performed at the disk [4].

2.2 Active Disks, IDISKs, and Smart Disks

Processor-embedded disk architectures changed the processing model of large databases by offloading I/O-intensive tasks to the disk. Thus, rather than sending to the host, data are filtered or processed directly at the disk, significantly reducing the network traffic between host and disk. Taking this approach, an Active Disk architecture was proposed by assuming application-level on-disk processing and large on-disk memory [5]. Active Disks use a stream-based programming model to address the software structure and implementation, in which host-resident code interacts with disk-resident code using streams [6]. The Active Disks work in [7] investigated scan-based algorithms for databases, nearest neighbor search, frequent sets, and edge detection. The Intelligent Disks (IDISKs) [8], proposed to succeed networks of workstations, uses on-disk integrated processor-in-memory called IRAM, and can perform direct disk-to-disk communication.

The Smart Disks architecture [9] introduced operation bundling to optimize query execution. Execution of each operation bundle is offloaded to the disk. While data transfer between disks was through the central unit, there was less involvement by the central unit in the overall query execution. Results from [9] showed that Smart Disks outperformed both host-based and cluster systems in comparable configurations.

3 Distributed Smart Disks (SD) Architecture

We propose a fully distributed SD architecture, where a SD group consists of SDs interconnected by a switched network. Applications such as database queries can be applied onto the group from a remote client. Figure 1 illustrates this single SD group (SDG) architecture. To exchange, combine, or re-distribute data within an SDG, data is communicated between the involved SDs without being routed through the client. For data queries to multiple database machines located remotely across the network, communication within a SDG as well as between several SDGs will be necessary. As

shown in Figure 2 of the multiple-SDG architecture, inter-SDG communication would occur when databases reside on separate SDGs, which may belong to different storage area networks that are connected to a common router. This architecture exploits data parallelism to meet the storage, computation, and communication requirements of future I/O-intensive applications. The architecture would utilize emerging interconnect technologies for low-latency and high-bandwidth data communication.

3.1 Storage Interconnect

Since the design of the SD architecture pushes data queries down to the disk, a network with high connectivity is favorable as its communication infrastructure. Traditional I/O architectures employ PCI/PCI-X bus protocols to communicate with storage systems and networks. Recent research on network-attached storage systems suggests the use of switched networks to maintain sufficient I/O bandwidth. We use the InfiniBand Architecture (IBA) [10] to represent the storage interconnect. IBA networks employ channel-based switched fabric, host/target channel adaptors (HCA/TCA), switches and routers to provide simultaneous point-to-point communications between end nodes. All end nodes communicate through switches with any other nodes in a subnet; or via a router to another end node.

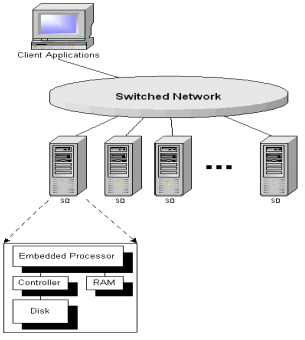


Fig. 1. The single smart disk group (SDG) architecture

4 Workloads: TPC-H, ARM, Data Clustering, and 2-D FFT

4.1 TPC-H Queries

For distributed SD architectures, performing parallel database operations is one of the logical approaches to exploit data parallelism. We used full queries Q_1, Q_6 and Q_{12} from the TPC-H benchmark suite to define our database workload.

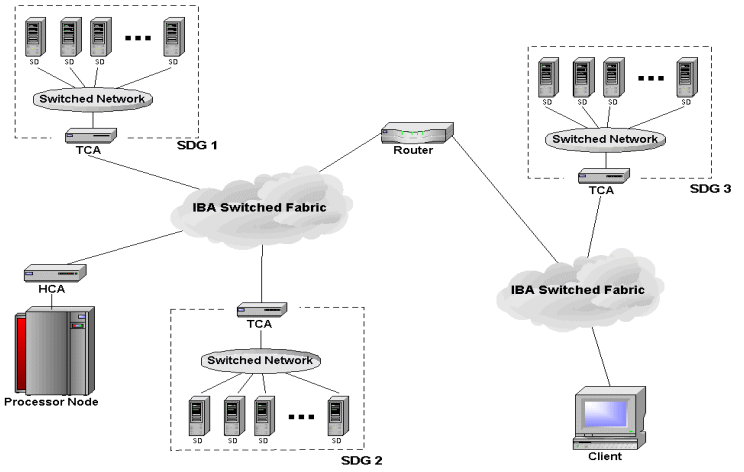


Fig. 2. The multiple-SDG architecture over an InfiniBand-based network

4.1.1 Query Primitives

We offloaded the *scan*, *join*, *sort*, *group-by* and *aggregate* primitives to the SDs, and implemented these primitives in parallel. We assume the database tables are evenly distributed across all SDs so that full parallelism of database queries can be exploited. In our implementation, the entire processing of a primitive is performed on the SDs without the remote client's participation or management. The *scan* primitive is implemented by sequentially retrieving and evaluating every tuple in the disk-resident data. The main purpose of *scan* is to filter data for further processing. Due to the limited amount of on-disk memory, out-of-core paging strategy was implemented. Since *scan* involves only local disk access, there is no inter-disk communication. For the remaining primitives, disk-to-disk communication is required. We adopted an out-of-core parallel bucket sort [11] for our *sort* implementation. It has four stages: sampling, re-distribution, internal sort, and external merge. Since *group-by* and *aggregate* are usually combined with *sort* and each is performed with a global communication, an analysis on *sort* would cover *group-by* and *aggregate*.

4.1.2 *Join* with Two SDGs

Two scenarios exist for the *join*: the two tables to be joined can reside in the same or different SDGs, or likewise, in one or more database machines. The former is the traditional implementation of a parallel *join*. In this work, we are interested in the latter scenario. Assume that the two tables to be joined, R and S, reside in SDG 1 and 2, respectively. Parallel join of these two tables will thus involve both intra- and inter-SDG communications. Assuming R is evenly partitioned across P SDs and S is evenly partitioned across Q SDs (P need not equal Q), the *join* processing proceeds as illustrated in Figure 3. Each SDG, denoted "SDG 1" or "SDG 2", starts by locally hashing its R or S tuples into "destination files" (or send buffers) for each of the SDs in both SDG 1 and SDG 2. Each SD then reads the tuples from its local "destination files", and sends the tuples to their destination SDs. Meanwhile, each SD also receives tuples destined for it from both SDGs. Upon completion of this re-distribution phase, each

SD will hold its R and S tuples, which are joined locally using a simple nested-loop join algorithm.

4.1.3 Full Query Execution

We assume that input tables are initially distributed across the SDs, and use PSQL's (a PostgreSQL database) *explain* command [12], along with TPC-H executable query text files to generate the query plans. The processing flow of these full TPC-H queries proceeds according to the full query execution protocol specified in Table 1.

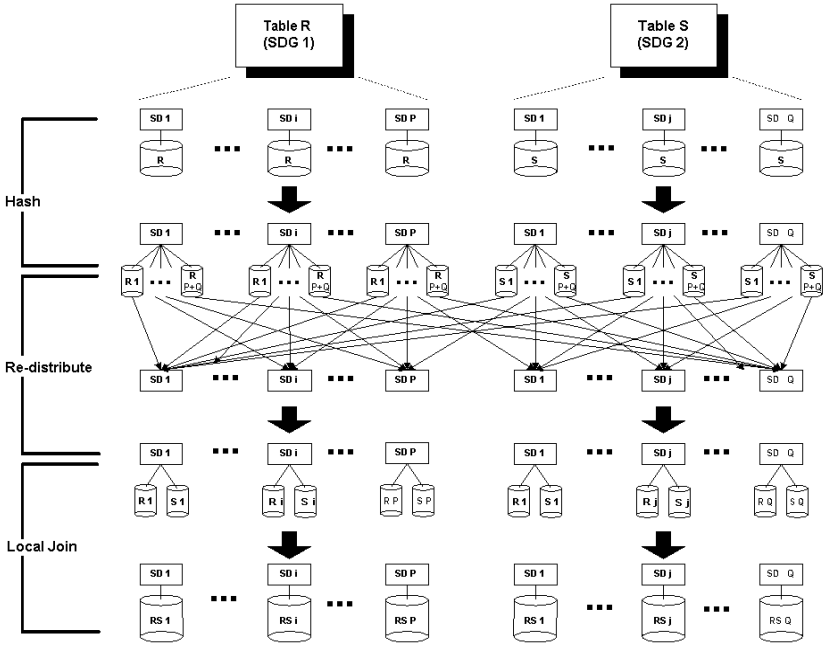


Fig. 3. Processing of *join* from two separate SDGs.

Table 1: TPC-H full query execution protocol

TASK	REMOTE CLIENT	SMART DISKS
Query parsing/optimization	Parse and optimize query	—
Query plan generation	Generate query plan	—
Query execution	Send the query plan to SDs Await completion signal from SDs Receive signal from the SDs	Receive the query plan Execute the query plan – Perform primitives – Re-distribute data – Store results locally – Communicate data Signal client of completion
Response to user	Receive results from the SDs Respond to user	Send results to client

4.2 Association Rule Mining (ARM) and Frequent Set Counting (FSC)

ARM techniques have been used to help address marketing questions such as "the 90% of customers who buy A probably also buy B". Given a collection of transactions T and each transaction contains items from an itemset I, an association rule takes on the expression $X \Rightarrow Y$, where itemset $X \subseteq I$ and itemset $Y \subseteq I$. An association rule $X \Rightarrow Y$ holds in T with a confidence C and a support S, if at least $C\%$ of all the transactions (in T) containing X also contain Y, and $X \Rightarrow Y$ exists in at least $S\%$ of all the transactions in T [13,14]. Typically, an ARM process consists of two phases. In the first phase, a set of *frequent* itemsets is found. In the second phase, the rules that satisfy the minimum S and the minimum confidence C are identified.

4.2.1 Count Distribution and Hybrid Distribution Methods

Our ARM FSC primitive exploits the total memory of an SD system for higher efficiency. The Count Distribution (CD) method is based on the *Apriori* algorithm in [15]. In the CD method, each SD executes the *Apriori* algorithm serially, followed by a global reduction to obtain the final results. The Hybrid Distribution (HD) method partitions the frequent itemsets into sufficiently large sections, then assign a group of SDs to each section. Thus, the HD method dynamically configures the processor grid for more effective load balancing.

4.3 Clustering for Large Data Sets

Clustering techniques help discover interesting patterns (called clusters) from large data sets. Such patterns often exist in high-dimensional space. Common applications that exploit clustering techniques include collaborative filtering [16], organization of document datasets [17], image processing and seismic studies, to name a few. Efficient clustering techniques address data and noise that both exist in high-dimensional spaces and subspaces, which result in an exponential growth of the search space for clusters. We implemented a data clustering primitive based on the parallel adaptive-grid and density-based algorithms studied in [18]. Define $S=A_1\times A_2\times...\times A_d$ as a d-dimensional numeric space, where $A=\{A_1,A_2,...,A_d\}$ is the attribute set and $D=\{D_1,D_2,...,D_d\}$ is the domain set. Assume $r=(r_1,r_2,...,r_d)$ is a d-dimensional input record. The space S is divided into a grid of non-overlapping cells, where each cell C is defined to be $c_{1k'}\times c_{2k''}\times...\times c_{dk'...'}$ and for all $i \in \{1,2,...,d\}$, $c_{ik'} \subseteq D_i$ and $\cup_k c_{ik} = D_i$. That is, $c_{ik'}=[l_{ik'},u_{ik'})$ is the partitioning interval for A_i. The cell C is *dense* if the fraction of the total data points contained in C is greater by some given factor than the value expected if the data were uniformly distributed in S. A cluster is thus a union of connected dense cells. The clustering primitive first computes the histogram for A_i, adaptively setting the bin size, then builds candidate dense cells and perform subspace clustering on the local records r to compute the local dense cells. Reduction is performed to obtain global data on the histogram, candidate dense cells, dense cells, and bounds for the dense cells.

4.4 Two-Dimensional Fast Fourier Transform (2-D FFT)

FFT is used in many scientific problems. The memory requirement on the SD motivates our evaluation of out-of-core 2-D FFT. The FFT takes $O(N \lg N)$ time to compute the discrete Fourier transform of an $N \times N$ matrix. For a distributed-memory model such as our SD architecture, the 2-D FFT can be performed using a parallel out-of-core transpose- and redistribution-based algorithm [19,20] as follows:

(1) Distribute input matrix A by rows as (BLOCK, *).
(2) Perform 1-D FFT on the rows of matrix A.
(3) Perform 2-D transpose on the intermediate matrix B from step (2).
(4) Perform 1-D FFT on the rows of B^T.

We evaluated the out-of-core 2-D FFT over a 2048×2048 complex matrix on the distributed SD architecture. Our implementation is based on the distributed-memory 2-D FFT algorithm prescribed in [21].

5 Experiments and Results

We first compared the performance of the fully distributed SD architecture with that of the partially distributed SD architecture, using TPC-H Q_1, Q_6 and Q_{12}. Then, we evaluated the performance and scalability of the fully distributed SD architecture for the ARM FSC, data clustering, and 2-D FFT. The following platform was used to simulate the workloads: 32 Pentium III Linux PCs running at 500 MHz, each with 64 MB of RAM and a local disk of 6 GB. The PCs are connected via a switched network. We used TPC-H database generator [22] to populate synthesized data into tables with scale factors (SF) of 0.1 and 1.0. *DiskSim* 2.0 simulator [23] was used to simulate the cost of accessing the disk drive.

5.1 Fully vs. Partially Distributed SD Architectures

We simulated with 4, 8, 16, and 32 SDs, page size of 8 KB, on-disk memory size of 32 MB, and SF of 0.1 and 1.0. We measured the execution times of TPC-H Q_1, Q_6 and Q_{12}. Figures 4 and 5 suggest that the fully distributed SD system provides further performance improvement over the partially distributed system. We used the formula T_{PD}/T_{FD} to evaluate the improvement obtained with a given number of SDs, where T_{PD} denotes the query execution time on the partially distributed SD system and T_{FD} denotes that on the fully distributed SD system. For example, in processing Q_6, hardly any noticeable improvement was achieved regardless of the number of SDs used, since Q_6 contains less than 0.01% of communication time. On the other hand, Q_1 contains about 10.4% communication time, while Q_{12} averages 0.56%. The low proportion of communication in Q_{12} is attributed to the highly selective *scan* that was performed prior to the *join*, *sort*, and *aggregate* operations in that query. Consequently, when running at an SF of 1.0 (the validation data size required by the TPC-H benchmark) the T_{PD}/T_{FD} values for Q_1 ranged from 2.88 to 3.41, and from 1.20 to 2.30 for Q_{12}, as

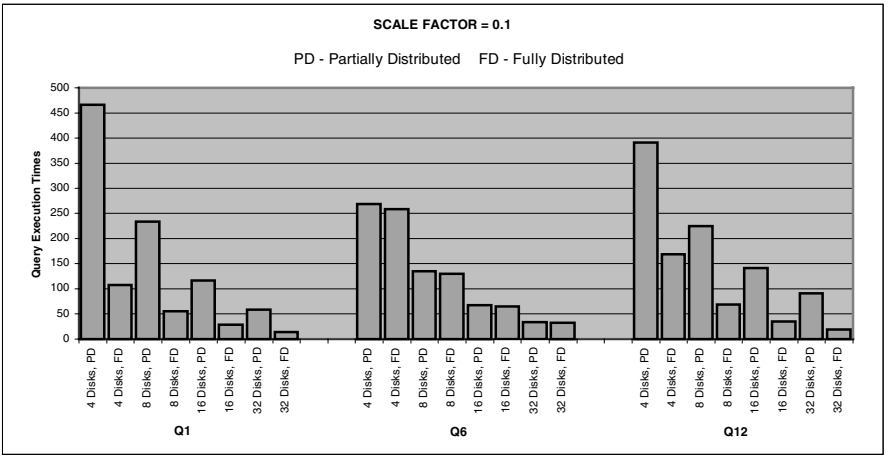


Fig. 4. Performance of partially vs. fully distributed SDs on Q_1, Q_6 and Q_{12} with SF of 0.1

expected. Increasing the data size from SF=0.1 to SF=1.0, however, exposed the inefficiency of our nested loop-based *join* primitive.

5.2 Association Rule Mining Frequent Set Counting

A synthetic transaction database was generated using the data generator from IBM [24]. The database had an average transaction size of 20, each transaction containing up to 25 distinct items. 100K transactions were generated for each SD. Figures 6 and 7 present the results for our ARM FSC primitive executed on SD systems with 2, 4, 8, 16 and 32 disks, using data page sizes of 4K, 8K, 16K, 32K and 64K bytes, and 64MB on-disk memory. The single-SDG processing model was assumed in this experiment. Figure 6 shows the results when the primitive uses the CD method; Figure 7 shows when the primitive uses the HD method.

Note that since data size increases linearly with the number of SDs, keeping the response time constant demonstrates scalability. Except for the data page size of 16 KB, the performance of the two methods are quite comparable at the 5% minimum support level– an indication that the hash tree for the frequent itemsets was able to fit into the available (64 MB) on-disk memory used in our evaluation.

5.3 Data Clustering Primitive

The data-clustering primitive discussed in Section 4.3 was executed over a data set of 2,234,961 records (≈ 179 MB), which contains 20-dimensional data with 5 clusters, and each cluster is of 5 dimensions. As with the ARM FSC primitive, we experimented with SD systems of 2, 4, 8, 16 and 32 disks, using data page sizes of 8K, 16K, 32K and 64K bytes with 64 MB on-disk memory. We also executed the primitive on a system with only 1 SD, where the processing is effectively serial.

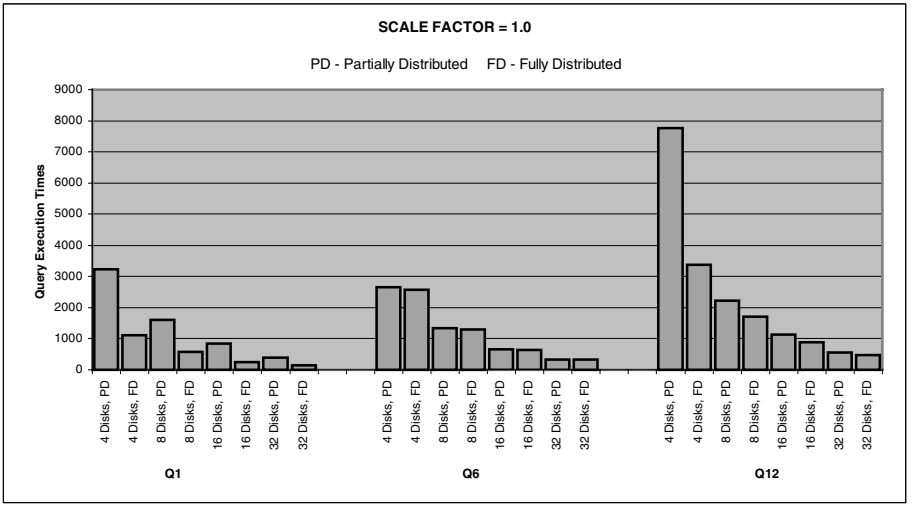


Fig 5. Performance of partially vs. fully distributed SDs on Q_1, Q_6 and Q_{12} with SF of 1.0

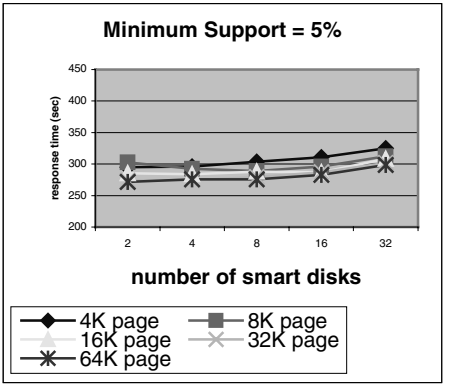


Fig. 6. Performance of FSC with the CD method

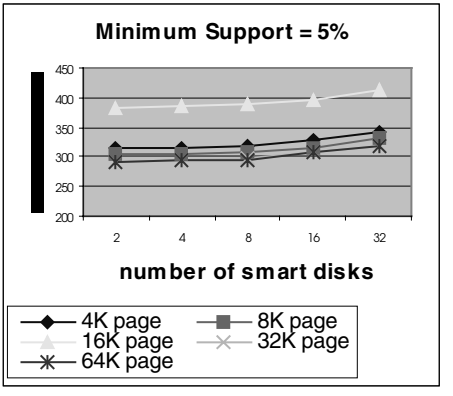
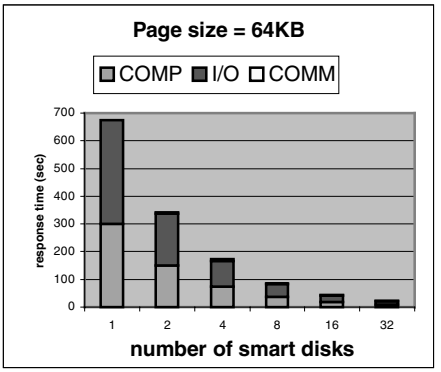


Fig. 7. Performance of FSC with the HD method

Figure 8 shows the performance of the clustering primitive when system size is scaled up, with the page size fixed at 64 KB. The results demonstrated that both the compute time taken to populate the candidate dense cells and the I/O time decreased when the number of SDs increases. Results for page sizes of 8 KB, 16 KB and 32 KB showed similar trend. Figure 9 presents the total response times of the clustering primitive versus number of SDs for all four page sizes. The results show a reduced response time as we increased data page size.

5.4 Two-Dimensional FFT

We evaluated the performance of the 2-D FFT primitive with 1, 2, 4, 8, 16 and 32 SDs for various data page sizes and 64 MB on-disk memory, and normalized all execution times with respect to the base case where 1 SD with a data page size of 4 KB are used.



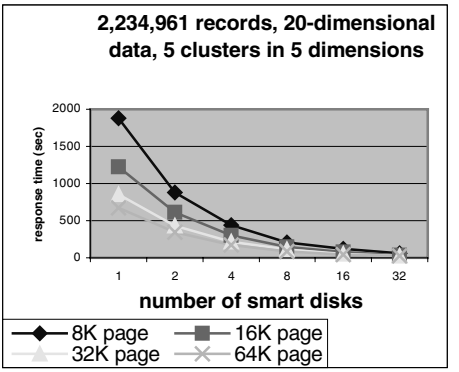


Fig. 8. Performance of clustering primitive **Fig. 9.** Overall scalability of clustering primitive

Our results show larger speedups for smaller data pages at smaller system size. As we increase the number of SDs, such difference in speedups diminishes. Figure 10 presents our results for the 2-D FFT workload.

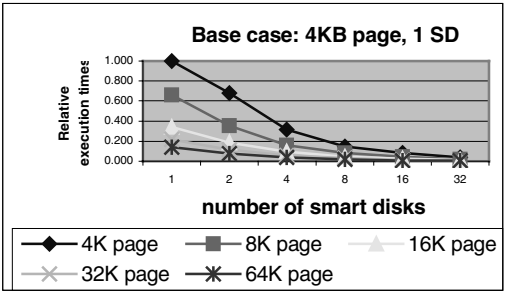


Fig. 10. Relative execution times of 2-D FFT

6 Conclusions and Future Work

For disk-based storage systems, increasing processing power on the disk results in offloading user-level code closer to data. Built on the progress to date, we showed that a distributed Smart Disk model is feasible. We evaluated a distributed Smart Disk architecture using TPC-H queries, ARM FSC, data clustering and 2-D FFT workloads. Our results suggest that such architecture would outperform a partially distributed, and thus, a centralized system. Our results also demonstrate that the Smart Disk architecture scales well with increased system size and data size. For future work, query optimization techniques for distributed-memory parallel databases can be studied to exploit cost models and load-balancing techniques.

References

1. W. Hsu, A. Smith and H. Young, "Projecting the Performance of Decision Support Workloads on Systems with Smart Storage (SmartSTOR)". Report No. UCB/CSD-99–1057. August 1999.
2. G. Gibson and R. Van Meter, "Network Attached Storage Architecture". Communications of the ACM, 43(11). November 2000.
3. D. Dewitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Processing". Communications of the ACM, 36(6). June 1992.
4. J. Gray and A. Reuter, Transaction Processing: Concepts and Technique. Morgan Kaufmann.
5. M. Uysal, A. Acharya and J. Saltz, "Evaluation of Active Disks for Decision Support Databases". International Conference on High Performance Computing Architecture. January 2000.
6. Acharya, M. Uysal and J. Saltz, "Active Disks: Programming Model, Algorithms and Evaluation" International Conference on ASPLOS. 1998.
7. E. Riedel, C. Faloutsos and D. Nagle, "Active Disk Architecture for Databases". Carnegie Mellon University. Report No. CMU-CS-00-145. April 2000.
8. K. Keeton, D. Patterson and J. Hellerstein, "A Case for Intelligent Disks". SIGMOD Record, 27(3).
9. G. Memik, M. Kandemir and A. Choudhary, "Design and Evaluation of Smart Disk Architecture for DSS Commercial Workloads", International Conference on Parallel Processing.
10. InfiniBand Trade Association, "InfiniBand Architecture Specification Volume 1 Release 1.0.a".
11. X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong and H. Shi, "On the Versatility of Parallel Sorting by Regular Sampling". Parallel Computing, Vol. 19, pp. 1079–1103. 1993.
12. Yu and J.Chen, "The PostGreS95 User Manual". University of California, Berkeley. July 1995.
13. E-H Han, G. Karypis and V. Kumar, "Scalable Parallel Data Mining for Association Rules". IEEE Transactions on Knowledge and Data Engineering, Vol. 12, No. 3. May/June 2000.
14. S. Orlando, P. Palmerini, R. Perego and F. Silvestri, "An Efficient Parallel and Distributed Algorithm for Counting Frequent Sets". VECPAR 2002 HPC for Computational Science. June 2002.
15. R. Agrawal and J. C. Shafer, "Parallel Mining of Association Rules". IEEE Transactions on Knowledge and Data Engineering, 8(6): 962–969. December 1996.
16. L. H. Ungar and D. P. Foster, "Clustering Methods for Collaborative Filtering". AAAI Workshop on Recommendation Systems. 1998.
17. Y. Zhao and G. Karypis, "Evaluation of Hierarchical Clustering Algorithms for Document Datasets". University of Minnesota Technical Report #02–022.
18. H. Nagesh, S. Goil and A. Choudhary, "Parallel MAFIA: Parallel Subspace Clustering for Massive Data Sets". Data Mining for Scientific and Engineering Applications. Academic Publishers. 2001.
19. R. Bordawekar, A. Choudhary and J. Ramanujam, "Compilation and Communication Strategies for Out-of-Core Programs on Distributed Memory Machines". CIT Center for Advanced Computing Research. Scalable I/O Initiative Technical Report. November 1995.

20. M. Kandemir, A. Choudhary and J. Ramanujam, “Improving Locality in Out-of-Core Computations Using Data Layout Transformations”. Carnegie Mellon University. May 1998.
21. LLNL website: http://www.llnl.gov/computing/tutorials/workshops/workshop/mpi/exercise.html.
22. Transaction Processing Performance Council, Soft Appendix for TPC-H and TPC-R Benchmarks.
23. G. Ganger, B. Worthington and Y. Patt, “The DiskSim Simulation Environment Version 2.0 Reference Manual”. Carnegie Mellon University. December 1999.
24. IBM Quest Data Mining Project website: http://www.almaden.ibm.com/cs/quest/syndata.html.