Distributed Computing with Load-Managed Active Storage

Rajiv Wickremesinghe, Jeffrey S. Chase, Jeffrey S. Vitter
Department of Computer Science
Duke University
Box 90129, Durham, NC 27708, U.S.A.
{rajiv,chase,jsv}@cs.duke.edu

Abstract

One approach to high-performance processing of massive data sets is to incorporate computation into storage systems. Previous work has shown that this active storage model is effective for a variety of problems. This paper explores opportunities to use active storage as a basis for exploiting asymmetric parallelism in applications using a streaming computation model on collections of fixed-size records. This model is the basis for much of the research in I/O-efficient algorithms, which deals with an important class of massive data problems not studied in previous work on active storage.

We present an extension of a streaming computation model for an external memory toolkit to support a flexible mapping of computations to storage-based processors. Our approach enables load-managed active storage: it exposes parallelism, ordering constraints, and primitive computation units to the system, which can configure the application to balance load and make the best use of available processing power. Emulation results from a sorting application demonstrate the potential of dynamic adaptation in load-managed active storage.

1. Introduction

Disk storage densities have been growing by over 50% per year, and storage systems are accumulating larger amounts of data than ever before. Storage is increasingly network-based and shared by many applications—including parallel applications—accessing storage through high-speed interconnects such as SANs, Infiniband, or Ethernet networks at gigabit speeds. These interconnects allow shared access to distributed storage across large network sites, or even across multiple sites in computational grids spanning administrative domains. These large-scale decentralized storage systems are a key to addressing Big Data computational challenges in the future.

It is an open question how to best integrate computation with storage access in these systems. Technology trends favor migration of limited application processing capability down the network storage hierarchy into storage servers or even high-end disk drives [13]. This *active storage* or "smart disks" model has been the focus of a significant body of research [1, 19, 20, 24, 26, 27], mostly directed at data mining and other database processing tasks. Future storage system components may offer varying degrees of computational power at varying granularities; thus we refer to computation-enabled storage nodes generically as Active Storage Units or ASUs. ASUs correspond to what is often called a storage "brick" to evoke their role as a building block for large-scale systems that store and process massive data. However, in our model an ASU could also correspond to a larger entity such as a block storage server or a site combining storage and processing.

ASUs allow processing capacity to scale naturally with the size of storage. They also have the potential to reduce data movement across the interconnect if searching, filtering, or read/modify/write steps execute directly on ASUs. This allows aggregation of larger numbers of drives behind each network port, and it can improve host processing performance since data movement in host memory is often a leading drain on host CPU resources. However, ASUs introduce new distributed computing challenges relating to controlling the mapping of application functions to ASUs, coordinating functions across ASUs and hosts, and sharing of ASU resources.

This paper explores a programming model to support computation on ASUs, with an emphasis on flexible resource management at the system level. Three factors motivate our approach. First, network storage is a shared resource, and storage-based computation should not occur if it interferes with storage access for other applications. Second, ASUs represent an *asymmetric* parallel processing model; the processing power available in the storage hierarchy may vary widely across configurations, and applications should configure to make the best use of the available

parallelism. Third, active storage offers a potential for local control over data movement and access order to optimize storage performance; application structure should expose ordering constraints precisely, so that ASUs may reorder operations when it is beneficial to do so.

We propose a model of *load-managed active storage*, which strives to integrate computation with storage access in a way that allows the system to predict the effects of offloading computation to ASUs so that it may configure the application to match hardware capabilities and load conditions. Our approach extends a well-established model [33] for *external memory programming*. In the extended model, programs specify computations in a dataflow style by composing streaming primitives—*functors*—that operate on *streams* or *sets* of fixed-size records flowing through them. One premise of our work is that the techniques for formulating external memory algorithms expose parallelism that the system may exploit by mapping functors to ASUs.

Execution on ASUs requires functors to satisfy additional constraints: ASU functors are passive entities whose computation occurs as a side effect of data access, and they perform bounded per-record processing with bounded internal state. These functors are either prepackaged, prevalidated kernel primitives or short code sequences whose execution behavior is statically determinable. These constraints create a basis for isolating ASUs and applications from damage by competing functors. In addition, the set and stream abstractions expose sequentiality and asynchrony of iterators for groups of records; in particular, set accesses allow ASUs to load-balance the distribution of set records across instantiations of a given functor. These aspects of resource management are closely related to several previous systems for I/O-intensive distributed computing, most notably River [9] and Abacus [4]. Section 7 sets our work in context with these and other previous systems.

One contribution of this paper is to present initial results obtained by executing instances of a configurable sorting program on an emulated computing environment with ASUs. The results demonstrate the potential of dynamic resource management to balance computation for load-managed active storage. Section 2 summarizes the case for ASUs and an *active I/O model* based on I/O-efficient algorithms. Section 3 presents our ideas for load-managed asymmetric parallelism. Section 4 discusses applications of this approach from the domain of geographic information systems (GIS). Section 5 describes the time-accurate emulation methodology used for the experiments, and Section 6 presents initial results showing the benefit of adaptation to changing system and data characteristics.

2. Background

The growing capabilities of disk drive ASICs motivated the concept of active disks [1, 26]. Applying simple operations, active disks can improve the overall efficiency of the application's use of network and host resources [23]. Most of the work in active storage has focused on scan-based applications and database operations, including comparison with database machines [19, 20, 24, 27] (see Section 7).

Many large data repositories are loosely organized collections of files from automated sensors. Astronomical data include spectral measurements, and high-dimensional photographic objects with several hundred attributes. The Sloan Digital Sky Survey will collect several terabytes per year to build a searchable research database [30]. Various NASA Earth observing missions have also collected over 250TB of high-resolution image data at different wavelengths, with observations arriving in a steady stream. TerraServer's snapshot of the USA alone includes 15TB of raw data. It is estimated that Google's collection of Web data exceeds 1.5 petabytes [15].

Active storage systems offer an attractive platform for handling these large data collections for several reasons:

- Processing even a fraction of a multi-terabyte data collection requires more resources than are available at a single processor; parallelism is essential to solve large data-intensive problems. Large-scale storage systems aggregate many disks; active storage integrates computation with the disks so that computational power scales with the storage capacity.

- Filtering and aggregation operations performed directly at the ASUs can reduce data movement across the interconnect, helping to overcome bandwidth limitations.

- Active storage offers an enhanced interface at a higher abstraction level, which may expose opportunities for ASUs to improve the effective utilization of the disk head by adaptively scheduling data accesses to account for physical disk properties. In particular, ASUs can reorder or retarget accesses to exploit knowledge of layout and location [21, 22].

Combining storage with colocated computation elements results in an inherently asymmetric system, with a distributed processing and storage component that scales with data set size, attached to an independently scalable compute component. Our goal is to extend previous work on active storage by embracing this asymmetric parallelism in our framework for designing and specifying I/O-intensive algorithms. Conventional parallel programming environments generally assume a homogeneous distribution of computing

resources, while an effective active storage architecture requires explicit management of asymmetric parallelism. A key goal of our approach is to structure applications in a way that enables the system level to coordinate resource management, without direct involvement from the application.

2.1. I/O-efficient Algorithms

Writing general parallel solutions is a specialized task, so some form of automatic extraction of parallelism is essential [14]. Our approach uses previous work in I/O-efficient (external memory) algorithms [34] as a starting point. I/O-efficient algorithms (or *I/O algorithms*) are designed for efficient processing of large external datasets; they minimize the number of I/O block operations used to transfer data between main memory and external storage. The *I/O complexity model* [2] illustrated in Figure 1 defines I/O complexity as a function of problem size N, block transfer size B and main memory size M. To reduce the number of I/O operations, I/O algorithms maximize locality of data accesses and favor sequential I/O using large blocks.

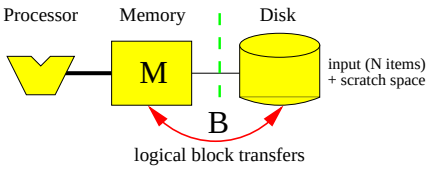

Figure 1. The I/O complexity model. Data moves between disk and memory in logical blocks of size B. The processor accesses data only in memory. Complexity in the model is the number of I/O operations needed to solve the problem.

A key insight from our work with I/O algorithms is that structuring an algorithm to be I/O-efficient implicitly exposes data parallelism inherent in the problem. I/O algorithms maximize useful computation on each block while it is in memory. Many I/O algorithms achieve this by sorting so that related data are adjacent and can be processed locally, with no reference to data in other blocks. In the restructured algorithm, these block computations may execute in parallel, independently and without synchronization. Parallelism is frequently localized in key data structures, or in program kernels that have well-known properties.

For example, *partition-and-merge* is a popular algorithmic technique that divides the problem into subproblems, solves the subproblems individually, then assembles the result from the solutions to the subproblems. Some or all of the computation on each subproblem can execute in parallel on ASUs in conjunction with block accesses for that subproblem. The merge operation then combines summary information from the ASUs. The configurable sorting algorithm discussed in Section 4.2 is one example of this paradigm.

More concretely, many useful algorithms have I/O complexity $(N/B)\log_{M/B}(N/B)$ and CPU complexity $N \log N$ for processing N records. These algorithms often use hierarchical operations that can be split, such as partitioning, aggregation (merge), and index lookups. For example, mergesort forms N/k sorted runs of size $k = M$, (consuming $N/k \cdot k \log k = N \log k$ work) and then merges the N/M runs (consuming $N \log(N/k)$ additional work), for a total of $N \log k + N \log(N/k) = N \log N$ work. The merge/combination step is often difficult to parallelize, but the block processing exposes "easy" asymmetric parallelism; algorithm runtime can improve significantly if some or all of the block operations execute on ASUs.

2.2. A Model for Active Storage

Figure 2 presents a simplified model for active storage with D ASUs and H hosts. In general, we suppose that hosts have large memories and powerful processors, and are dedicated to a single application at a time. In contrast, ASUs may be network storage units shared by multiple applications, and may be memory-constrained and/or less powerful. The model represents this asymmetry with a parameter c representing the ratio of host to ASU processing power. Additional parameters not shown in Figure 2 capture disk I/O properties, and network latency and bandwidth. The emulator and system infrastructure described in Section 5 uses these parameters to model the system and control the distribution of load across hosts and ASUs.

The relative values of these parameters determine the performance for a given application and system configuration. For example, if half the total processing power is at the hosts, the application should place half the computation there, assuming the I/O load is distributed evenly. The goal of load-managed active storage is to enable automatic dynamic placement of data and computation according to resource availability in the system.

3. Load-Managed Active Storage

This section outlines a collection of abstractions for distributed computing with ASUs. Specifying applications using these abstractions exposes data dependency information; the system can use this information to dynamically assign application components to hosts and ASUs, and control the movement of data among these components. Our approach is *data-driven* because it uses knowledge of application structure and data dependencies to choreograph computation.

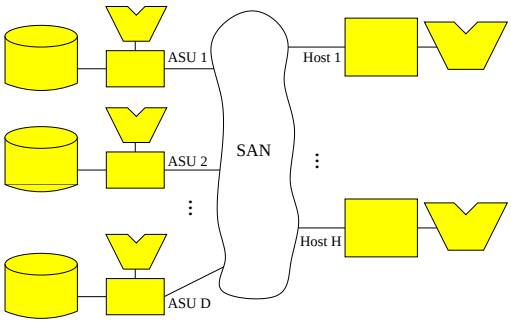

Figure 2. Active storage model. Multiple ASUs (D) with associated disk storage and processor, and multiple hosts (H) with memory and processor.

3.1 Functors

We propose a programming model that partially decomposes the computation into primitive processing steps called *functors*, which apply specific functions to streams of records passing through them. Functors may have multiple inputs and outputs, and are composed to build complete programs that process data as it moves from stored input to output, possibly in multiple passes.

A subset of the functors are capable of executing directly on ASUs. These functors are "stacked" on stored data collections (*containers*) to process data as a side effect of I/O operations. Their per-record computation demand and total memory usage are bounded, facilitating load management and resource provisioning. More complex read/modify/write operations may be permitted in common, verified computation kernels, e.g., for useful primitives such as sorting.

For functor operations that are commutative and associative, the system may replicate multiple instances of a functor across multiple computing elements. The model defines set and stream collection types (described below) that enable the system to spread load across multiple elements by routing data to any instance of a replicated functor. This is a powerful technique for exploiting the parallelism inherent in active storage.

Data-driven programming models are a good match for implementing many I/O algorithms, in part because they have predominantly sequential access patterns, in which each data record is consumed and processed once in each phase of the computation. The TPIE [33] library—the starting point for our prototype—reflects this principle by exporting a dataflow-like interface. TPIE is an external memory programming toolkit that processes data in *streams* of fixed-size records. It provides basic I/O-efficient data structures and primitives including *sorting*, *merging*, and *distribution*. A pluggable Block Transfer Engine (BTE) abstracts the underlying storage system block access operations, facilitating portability to various storage and access models. TPIE has been used to implement I/O-efficient algorithms for a range of problems in Geographic Information Systems (GIS) and related domains.

3.2 Containers

To benefit from parallelism, we propose to extend the TPIE stream paradigm with additional data aggregation primitives and *container* types for stored data collections. The container types support common access methods for I/O algorithms. These include: (1) unordered scanning, which consumes and processes records in an unspecified order; (2) ordered scanning, which consumes and processes records sequentially; and (3) random-access, which processes records in an application-specific order unknown to the system. We associate different data containers with these patterns: *sets*, *streams*, and *arrays* respectively.

- Sets are data containers that do not define the order of records returned in satisfying `read` operations. This allows the system to provide records in any order that is convenient, and spread them arbitrarily across replicated functors.

- Streams are the traditional sequential-access data type for use when the application expects to process records in a specific order. While a set may deliver the next *available* record, a read on stream always delivers the next unconsumed record in a defined sequence, even if this is less efficient.

- Arrays allow arbitrary accesses to structured collections of records. This model is useful for supporting external indexes over collections of records, such as the spatial indexes outlined in Section 4.1.

Figure 3 illustrates sets, streams and arrays. Sets and streams are always accessed and processed in their entirety: records contained in a set or stream are marked as *pending* or *completed* for each scan of the container. When intermediate data outputs are to be processed only once as input to the next phase, the scan may be *destructive*: storage for completed records is released as records are consumed, so that only pending records remain in the collection.

The model also includes a mechanism to group related records within a data collection into units called *Packets* that are always processed as a whole. Packets allow intermediate structure to be represented within sets. They impose a partial order on the records in a set, and constrain the distribution of records across functor instances.

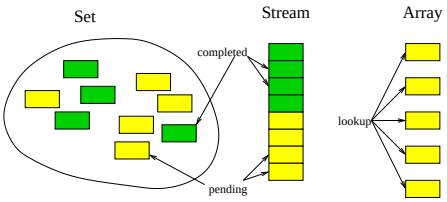

Figure 3. Basic data containers. Sets have no specific order, streams are ordered, and arrays are random-access.

Packets are useful when a functor must access all of the records in a group to complete an operation. Figure 4 illustrates an example of use of packets in a sorting program that pre-sorts batches of data into sorted groups for downstream merging. If the pre-sort executes on an ASU, the size of the packet may be limited by a memory bound on the ASU-resident sorting functor. Grouping the records in a packet preserves the sort property as the records move through later phases of a compound algorithm.

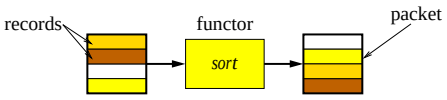

Figure 4. Functors and Packets. Example of a sort functor which sorts groups of records and uses packets to preserve the local order of sorted records.

3.3. Resource Management

One goal of the programming model is to enable the system to schedule computations to benefit from the resources available, and adapt to changes in resource availability as a program executes. Dynamic changes in load at different points of the system can cause imbalances within the system, unnecessarily compromising overall application performance. In particular, the load distribution is difficult to determine statically when ASUs are shared by multiple applications or if nodes have heterogeneous performance characteristics. Moreover, many data-intensive applications are data-dependent; static partitioning of work does not yield a predictably balanced distribution. The TerraFlow drainage modeling package (see Section 4) is one example of such an application.

The functors and data aggregation primitives proposed above expose information about application structure to enable dynamic resource management within the system. They make it possible for the system to map the computation load onto the available hardware by controlling the replication degree and placement of functors, relationships among functors, and the movement of data through the network of processing stages. Rather than layering application-specific resource management above the programming model, dynamic resource management may be integrated into the system infrastructure, and applied in a general way that avoids violating ordering constraints necessary for application correctness.

For example, sets and replicated functors allow ASUs and host nodes to perform dataflow routing between functors intelligently. The routing of records across functor instances may be responsive to dynamic load conditions visible to the system. In some cases, randomized routing techniques like *simple randomization* (SR) [35] may reduce data dependencies and interference at the ASUs. Routing policies may also consider static information about node capacity to handle heterogeneous processing rates. Known bounds on functor computation cost per unit of I/O facilitates these resource scheduling decisions. Load management may also adjust the number of functor instances for a computation stage, migrate functors between host nodes and ASUs, or adjust the assignment of functor instances to host nodes or ASUs.

In addition, it is often possible to configure functors to adjust the balance of computation load across the phases of an application, instead of or in addition to rebalancing load across the nodes participating in each phase. This can improve performance by balancing load across multiple phases executing in parallel in a pipelined fashion. In the sorting example in Section 4.2, the fan-in of merge functors and the fan-out of distribution functors may vary to adjust the balance of load between sort pipeline phases executing on ASUs and hosts. In this way, the system can adjust the computation to the degree of parallelism available, even when that parallelism is asymmetric. The emulation results in Section 6 illustrate the potential of this approach in a parallel sort executing on ASUs and hosts. This idea allows migration of compute load without moving application objects; it applies naturally to hierarchical operations like *Classify*, *distribute*, and *merge*.

4. Example Applications

Integrating computation with I/O allows data-intensive applications to process larger volumes of data by distributing part of the computational burden. We are considering the role of this model of active I/O for geographic information systems (GIS) applications, which process growing volumes of spatial data. This section outlines the potential role of active storage and ASUs in three application primitives important to GIS: terrain analysis, spatial indexing,

and sorting.

4.1 Terrain Analysis with TerraFlow

Terrain data is typically generated by sensors as raster grids. GIS programs often operate directly on the grid representations; while they are less compact than vector representations, the equally spaced points are easy to index and manipulate. Terrain analysis processes the entire grid dataset to generate indices characterizing properties of the data, e.g., by computing an attribute or index value such as water accumulation for each cell of the grid. These indices are then used to isolate features of interest.

We have built an I/O-efficient terrain analysis application—called TerraFlow [31]—that generates flow indices characterizing the slope orientation and the "upstream" area of each grid cell of a large terrain. A key component of TerraFlow identifies watersheds, which naturally partition the terrain along ridges. It takes a preprocessed grid as input and assigns different "color" labels to the cells contained within each distinct watershed. Step 1 restructures the grid to include neighbor and position information in each grid cell, allowing cells to be processed independently and effectively converting the grid from a *stream* into a *set*. This step is easily distributed (e.g., by blocking) because it has minimal data dependencies. Step 2 invokes an external sort to order records by elevation in preparation for the next step. Section 4.3 discusses sorting in more detail. Step 3 uses neighbor information to propagate colors from the lowest points up/outward to the peaks and ridges. This step is difficult to parallelize because it uses time-forward processing [12] and relies on ordering for correctness. Thus data parallelism in ASUs may improve the first two steps of the watershed computation considerably while offering limited improvement of the final step.

4.2. Spatial Indexes

Spatial index structures support efficient search for spatial locations matching specified properties, which often requires a complex multi-dimensional search. An R-tree is a general structure used to build multi-dimensional indexes by splitting a space into a hierarchy of nested and possibly overlapping regions. R-trees can be used to satisfy online queries, render visible surfaces, or locate points for map generation. The upper levels of an R-tree are more effective at refining the search space, especially for high-dimensional data. A standard technique reverts to sequential search (scanning) of subregions after traversing the top of the tree to locate subregions of interest.

We can construct a distributed R-Tree by partitioning the nodes of a tree among hosts and ASUs in a manner similar to master-client R-trees [29]. For an R-tree with multiple ASUs, the upper portion of the original tree is unchanged and placed on one host or replicated across multiple hosts. The lower part of the tree is replaced with subtrees on the disk nodes. One option to construct the subtrees is to build a tree over all the data at each ASU, and treat each as a leaf of the host tree. An alternative is to stripe a host leaf across all of the ASUs. Figure 5 illustrates these two options. Because the latter option stripes leaves across ASUs, every query executes in parallel on all of the ASUs, which is useful to bound search latency. The former option distributes the searches across the ASUs, which is useful in server applications with many concurrent searches. Hybrid solutions using a subset of the ASUs or replicating subtrees on multiple ASUs are also possible.

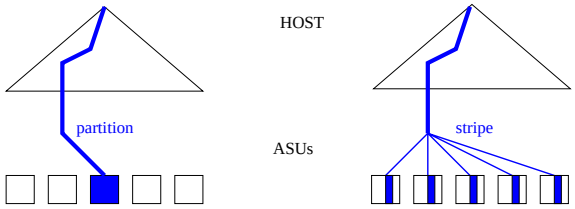

Figure 5. R-tree organization. (a) The R-Tree determines the ASU to use. (b) The R-tree determines the stripe to use.

This method of distributing R-trees uses a hierarchical decomposition at the host, and large sequential scans on the ASUs. This technique also applies to other two-level I/O-efficient index structures. For online data structures, the maintenance work (for example, rebalancing) at the lower levels can run as a batch job running on the ASUs, while the host layer maintains the upper levels online.

4.3. DSM-Sort

We constructed a hybrid distribute/merge sort program called *DSM-Sort* for active storage systems using the data-driven functor model described previously. The program combines distribute, sort, and merging functors in a configurable way, allowing flexible mapping of computation load and data to the underlying hardware. DSM-Sort distributes the data into buckets, and then sorts each bucket by forming sorted runs and merging the runs in a final merge, which may involve multiple passes for very large data sets. DSM-Sort is configurable to partition load in different ways to keep all the components of the system maximally utilized, as determined by several parameters described below.

Figure 6 illustrates an instance of DSM-sort. For this instance, we apply three types of operations as follows.

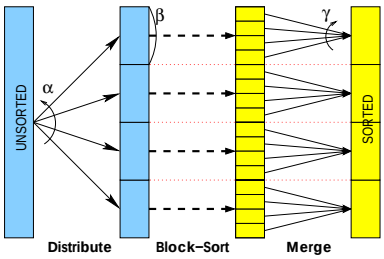

Figure 6. Overview of DSM-Sort. Each column shows the data at a different stage of the sort (unsorted, partitioned, locally sorted, sorted). The mapping of functionality to processing element is not shown here.

1. Perform an α-way distribute of the data set to partition it into α subsets that can be sorted independently. The buffer space on the ASUs restricts α.

2. For each block of β records in each subset, we use a suitable fast internal sort to form a total of N/β sorted runs. The available memory size limits the run length.

3. Use a γ-way merge to form sorted runs striped across the ASUs. The ASU buffer space restricts γ.

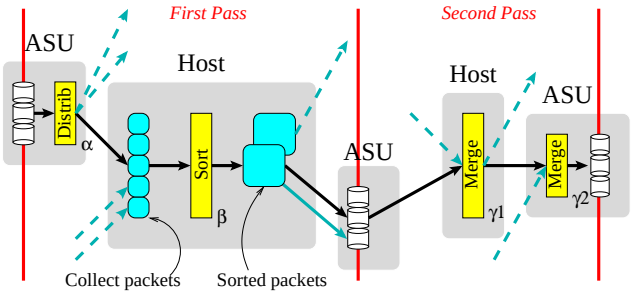

Figure 7. Sorting with active disks. Illustration of a two-pass sort performing distribute-sort-merge.

Figure 7 illustrates the operation of the algorithm when only two passes are required. This method may be applied for additional passes (more passes may theoretically be required if γ is small), but two passes are sufficient in practice. At least two passes are required if the data set is larger than the total available memory. Figure 7 shows the data flow from the ASUs to the hosts and back again during each pass. In the first pass, ASUs distribute data into buckets. Each bucket is broken into blocks and each block is sorted and stored at the ASUs. In the second pass, these sorted blocks in each bucket are then merged, giving the sorted result. The merge is divided between hosts and ASUs, so that $\gamma_1\gamma_2 = \gamma$. The total work done is parameterized by α, β, and γ, where $\log(parameter)$ is the number of compares per key:

$$\text{Total Work} = n\log\alpha + n\log\beta + n\log\gamma = n\log(\alpha\beta\gamma).$$

In particular, this means that

$$\alpha\beta\gamma = n.$$

DSM-Sort can adaptively reconfigure to match varying parameters of the active storage systems. Choosing the distribution, sort, and merge parameters appropriately allows us to balance computation at ASUs and hosts, as well as conform to memory constraints on the ASUs.

5. Emulator Implementation

We are interested in how the parameters of the active storage system affect the performance of various configurations of algorithms based on the data-driven functor model. To experiment with DSM-Sort and other active storage applications, we implemented a timing-accurate emulator [18] for systems with ASUs. The emulator serves as a test execution platform for real applications built using the model; it executes the instructions of application functors directly on the CPU of the emulation platform, and accesses data on associated storage as needed. An embedded simulator determines delays associated with communication and I/O events in the emulated system. The emulator allows us to examine the behavior of different algorithm configurations as the parameters of the system change. By varying the parameters for the simulation, we can extrapolate technology trends and explore a wide range of active storage configurations.

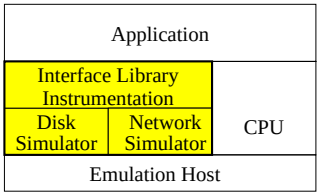

Figure 8. Emulator organization. Application executes directly on the emulation host, but network and disk I/O is provided by the interface library.

Figure 8 illustrates the overall structure of the emulator. The parameters to the emulator include the number of hosts and ASUs and their CPU speeds relative to the emulation

platform. The emulator is instrumented to report application progress, overall runtime, and resource utilization for each host and ASU in the target (emulated) system as the application executes.

The emulator divides program execution into *execution segments* separated by calls into an interface library, which exports the functor and collection abstractions and I/O primitives to the application. The emulator predicts the completion time for each execution segment based on the segment execution times, communication and storage access requests, and the parameters of the emulated configuration. It directly measures CPU time for each execution segment using the fine-grained processor cycle counter, then scales the elapsed time according to the relative speed of the emulated processor (host or ASU). The embedded event-driven simulator simulates disk I/O and communication operations to determine the delays they would impose in the emulated system.

To preserve timing accuracy, the emulator intercepts and maintains an event queue for all communication events. The queue ensures that events (and thus execution segments) take place in temporal (causal) order. Because our interface library supports synchronous (blocking) operations, the current implementation uses individual threads to store execution context for each node in the system. Context switching between threads is controlled by the event queue. It maintains a global view of time when nodes communicate, in order to maintain dependencies, predict the degree of overlap for concurrent operations on different nodes, and so determine overall run time. Blocking synchronization (condition variable *wait*) is supported by posting wakeup events at $t = \infty$, and updating the event firing time t in the *signal* primitive.

Our current emulation uses simple disk and network simulation models. The disk simulation does not model detailed seek and rotational times because our current experiments perform all I/O sequentially. The disk simulation uses a base aggregate transfer rate to calculate elapsed time under an I/O load, assuming read-ahead and write caching for sequential I/O: the disk initiates the *next* I/O automatically, and writes wait only for the *previous* write to complete. In future work we plan to integrate a full disk simulator (as in [18]) into the emulation.

The network model for the emulation uses only host-ASU communication, and assumes that the processor saturates before the individual network links. This is realistic given our processing requirements, but can introduce significant inaccuracy for wide-area systems with active storage sites. If the interconnect bandwidth is limited, direct ASU-ASU communication may be required [1, 32].

6. Experimental Results

We conducted experiments with DSM-Sort on emulated active storage configurations. We report initial results to illustrate the effect of two key forms of adaptation enabled by our approach. These experiments sort 128-byte records with 4-byte keys. The emulation host is a 750MHz Intel P-III. We simulated hosts with performance equal to the emulation host, and ASUs with performance scaled to give $c = 4, 8$ (i.e., ASU clock rate was $1/4$ or $1/8$ of the host clock rate). We report timings from the first pass of sorting (run formation), omitting the final merge phases.

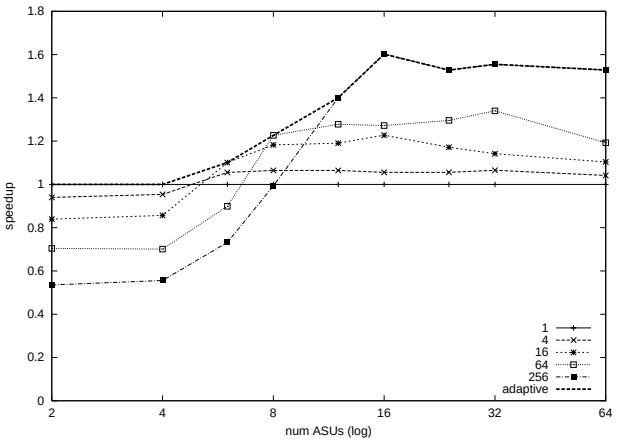

Figure 9. Speedup. Speedup achievable in DSM-Sort by adaptively configuring the mapping of function to CPUs as ASUs are added. Data series represent different configurations (α values) of the algorithm. This experiment uses one host, which saturates at 16 ASUs.

Figure 9 shows the speedup obtained by different configurations of DSM-Sort as ASUs are added to a single host, with the input data initially distributed across the ASUs. The graph reports speedup relative to a baseline using conventional storage units with no integrated processing; all computation occurs on the host. The active configurations have ASUs with one eighth the processing power of the host. The graph shows several lines corresponding to different α parameter values (distribute order) for the DSM-Sort algorithm with the distribute functors executing on the ASUs. Higher α values shift more computation load per block to the ASUs, since the number of comparisons per record grows logarithmically with the distribute order.

With small numbers of ASUs, the aggregate computation power of the ASUs is small relative to the host. In these cases, the host is limited by the ASUs and is idle much of the time. Higher α values increase the load on the bottle-

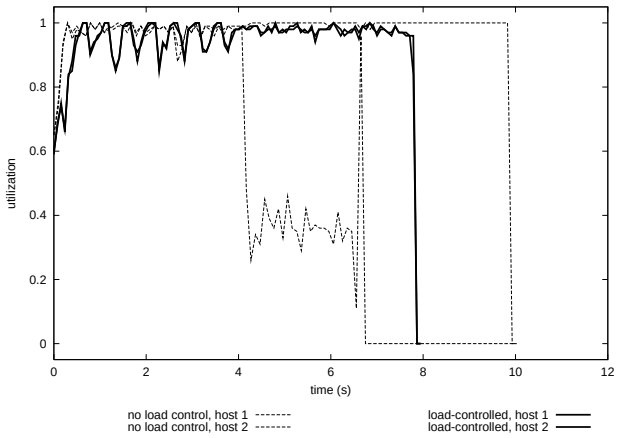

Figure 10. Effect of skew. Utilization of host CPU for two DSM-Sort runs on two hosts and 16 ASUs, with and without load management. The first half of the input data is uniformly distributed, while the second half is skewed, resulting in a potential for unbalanced load across the hosts in the distribute phase. The load-managed run terminates earlier; it shows nearly identical utilizations on the two hosts even when the input data is skewed, illustrating the potential benefit of dynamic load-managed distribution.

necked ASUs, resulting in a slowdown relative to a conventional system with no active storage. As ASUs are added to the system, each ASU processes a smaller share of the input data, and the aggregate computational power devoted to the distribute phase increases. Collectively, the ASUs begin to process records faster than the host, which ultimately saturates. With larger numbers of ASUs, speedup improves with higher α values. These shift more of the computation load to the distribute phase, which benefits from the additional power of the ASUs.

We also conducted an experiment to illustrate the potential benefit from load-managed distribution of records across replicated functors. This experiment consists of a DSM-Sort running the sort phase on two hosts. The first half of the input data is drawn from a uniform distribution, while the second is from an exponential distribution. Figure 10 shows CPU utilizations on both hosts during the course of the experiment, for runs with and without dynamic load management. The baseline run assigns half of the α distribute subsets to one host, and the other half to the second host. The skew in the input data produces a poor distribution of records across the buckets, resulting in a load imbalance. In the load-managed case, each of the α subsets is spread across both hosts, with each host sorting blocks of β records from each subset as they arrive. A *simple randomization* (SR) [35] policy assigns the records of each subset to the hosts, preserving the balance of records across the hosts. Different load balancing methods can be used, depending on the amount of information available.

7. Related Work

Key studies of active storage include [19, 23, 26, 32]. A large body of work primarily considers database and filtering problems [19, 20, 23, 24, 27]. A stream-based model is proposed in [1]. This work uses a single host as a manager/router for the active disks and focuses on limited interconnect bandwidth. In contrast, we fully utilize multiple hosts and integrate computation into I/O, rather than using disks as hosts. DataCutter [11] maps this stream model to the Grid and shows that different workloads require different numbers and placement of filter instances. DataCutter filters are statically placed and do not adapt to dynamic load imbalances.

River [8–10] is a data-flow programming environment for clusters that balances load in the presence of performance heterogeneity. River's *distributed queue* (DQ) allows consumers and producers to process data at different rates. The DQ relaxes ordering constraints for better performance, similar to our sets. Graduated declustering uses mirrored data to smooth read performance.

Abacus [3–5] is a mobile object system. It dynamically balances load by relocating mobile objects based on black-box monitoring. Abacus focuses on deciding where to locate objects. Our approach is a hybrid function-moving, data-driven system. Higher-level distributed data structures are parameterized to dynamically redistribute load, and sets route packets based on resource needs.

Ninja [17] provides scalable clustering and storage for distributed Internet services using Distributed Data Structures (DDS) [16]. Scalability is obtained by partitioning (using client-side maps) and replicating across cluster nodes. Two-phase commits keep replicas coherent. Although DDS operations are *atomic*, they do not support *transactions* across multiple objects or operations. SEDA [36] exposes application structure to enable resource management for Internet services. Internet services process many small independent requests, making them naturally parallel; Ninja is not concerned with extracting parallelism in large individual computations.

Optimal parallel sorting algorithms are complex because they extract a maximum degree of parallelism. This is not necessary in practice since we only have a limited number of processors in the system. For clusters, HPVM MinuteSort [37] is a one-pass sort based on NOW-Sort [6, 7]. It uses *sort nodes* with more memory and CPU, and *I/O nodes* with more disks. The I/O nodes distribute records to the sort nodes which then sort and return them. Most of the work in

this system is done on the sort nodes; the I/O nodes are statically selected to partition the data.

8. Conclusion

This paper proposes extensions to a streaming computation model to enable a flexible mapping of computations to active storage units (ASUs). Computations are specified as a network of *functors* that operate on groups of records that flow through them. Applications may be configured for varying degrees of computation in their functors, which may be mapped to execute on ASUs. We emphasize opportunities to use active storage as a basis for exploiting *asymmetric* parallelism as a natural extension to I/O-efficient algorithms for an important class of massive data problems not studied in previous work on active storage.

A key goal of our approach is to enable *load-managed* active storage. Our approach exposes primitive computation units and their costs to the system, enabling the system to control the mapping of computational workload to processing units in order to maximize global system performance. Functors perform bounded computation as a side effect of I/O access; the mapping of functors to ASUs and hosts is configurable and potentially dynamic. Specifying record collections as *sets* enables the system to balance the distribution of load across instances of functors, without violating ordering constraints. We present results from a reconfigurable active storage mergesort algorithm to demonstrate the potential of load-managed active storage. Future work will address performance isolation and security issues with this approach. We are also investigating external decentralized indexing structures.

Acknowledgements. David A. Hutchinson contributed to early discussions of the ideas in this paper. This research was supported in part by the National Science Foundation through research grant CCR–0082986, ESS grant EIA–9870724, and RI grant EIA–9972879. Vitter was also supported by the Army Research Office through grant DAAD19–01–1–0725.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, 1998.

[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] K. Amiri, G. Gibson, and R. Golding. Highly concurrent shared storage. In *20th International Conference on Distributed Computing Systems (ICDCS '00)*, pages 298–307. IEEE, Apr. 2000.

[4] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 307–322, Berkeley, CA, June 18–23 2000. USENIX.

[5] K. S. Amiri. *Scalable and manageable storage systems*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2000.

[6] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In J. M. Peckman, editor, *Proceedings, ACM SIGMOD International Conference on Management of Data: SIGMOD 1997: May 13–15, 1997, Tucson, Arizona, USA*, volume 26(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 243–254, New York, NY 10036, USA, 1997. ACM Press.

[7] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. M. Patterson. Searching for the sorting record: Experiences in tuning NOW-sort. In *SPDT'98: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 124–133, Oregon, Aug. 1998. ACM SIGMETRICS. U.C. Berkeley.

[8] R. H. Arpaci-Dusseau. *Performance Availability for Networks of Workstations*. PhD thesis, Graduate Division of the University of California at Berkeley, 1999.

[9] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, May 1999. ACM Press.

[10] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. The architectural costs of streaming I/O: A comparison of workstations, clusters and SMPs. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 90–101, Las Vegas, Nevada, January 31–February 4, 1998. IEEE Computer Society TCCA.

[11] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Optimizing execution of component-based applications using group instances. In *IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society Press, May 2001.

[12] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 139–149, 1995.

[13] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 92–103, 1998.

[14] J. Gray. Storage bricks. Talk at Conference on File and Storage Technologies, Jan 2002.

[15] J. Gray and T. Hey. In search of petabyte databases. Talk at Conference on High Performance Transaction Systems, Oct. 2001.

[16] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 319–332, Berkeley, CA, Oct. 23–25 2000. The USENIX Association.

[17] S. D. Gribble et al. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks (Amsterdam, Netherlands: 1999)*, 35(4):473–497, Mar. 2001.

[18] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, and G. R. Ganger. Timing-accurate storage emulation. In *Proceedings of the Conference on File and Storage Technologies (FAST)*. USENIX, January 2002.

[19] K. Keeton. *Computer architecture support for database applications*. PhD thesis, Graduate Division of the University of California at Berkeley, July 1999.

[20] K. Keeton, D. Patterson, and J. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(3), 1998.

[21] C. Lumb, J. Schindler, G. R. Ganger, E. Riedel, and D. F. Nagle. Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 87–102, Berkeley, CA, Oct. 23–25 2000. The USENIX Association.

[22] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the Conference on File and Storage Technologies (FAST)*. USENIX, January 2002.

[23] E. Riedel. *Active Disks – Remote Execution for Network-Attached Storage*. PhD thesis, CMU, November 1999.

[24] E. Riedel, C. Faloutsos, G. R. Ganger, and D. Nagle. Data mining on an OLTP system (nearly) for free. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 13–21. ACM, 2000.

[25] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, June 2001.

[26] E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage. Technical Report CMU-CS-97-198, Carnegie Mellon University, December 1997.

[27] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the Twenty-fourth International Conference on Very Large Databases, New York, NY, USA, 24–27 August, 1998*, pages 62–73, 1998. also appears in [25].

[28] L. F. Rivera-Alvarez. Disk-to–disk parallel sorting on HPVM clusters running Windows NT. Master's thesis, University of Illinois at Urbana-Champaign, 2000.

[29] B. Schnitzer and S. T. Leutenegger. Master-client R-Trees: A new parallel R-Tree architecture. In *Statistical and Scientific Database Management*, pages 68–77, 1999.

[30] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner. Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data: May 16–18, 2000, Dallas, Texas*, volume 29(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 451–462, New York, NY 10036, USA, 2000. ACM Press.

[31] L. Toma, R. Wickremesinghe, L. Arge, J. S. Chase, J. S. Vitter, P. N. Halpin, and D. Urban. Flow computation on massive grids. In *Proc. ACM Symposium on Advances in Geographic Information Systems*, 2001.

[32] M. Uysal. *Programming Model, Algorithms and Performance Evaluation of Active Disks*. PhD thesis, University of Maryland, 1999.

[33] D. Vengroff. TPIE user manual and reference, 1995 with subsequent revisions. Available via WWW at http://www.cs.duke.edu/TPIE/.

[34] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[35] J. S. Vitter and D. A. Hutchinson. Distribution sort with randomized cycling. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01)*, pages 77–86, New York, Jan. 7–9 2001. ACM Press.

[36] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for Well-Conditioned, scalable Internet services. In G. Ganger, editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 230–243, New York, Oct. 21–24 2001. ACM Press.

[37] X. Zhang, L. Rivera, and A. Chien. HPVM MinuteSort. White Paper, c1999. see Rivera's Thesis [28].