Chapter 1

Active Storage

1.1 Background

General-purpose computation capability is no longer localized at a single point (the CPU) of computer systems. *Active storage* describes a system that provides computation capability at the storage elements, in addition to dedicated compute entities. Active storage is the nexus for multiple avenues of research, discussed in the following sections.

1.1.1 Active Disks

Several groups have investigated the addition of processing capability to storage elements (*active disks, intelligent disks*) [Riedel et al. 1998; Keeton et al. 1998; Amiri et al. 2000b; Gibson et al. 1998; Acharya et al. 1998]. A key assumption in the following related work is that the aggregate computation capabilities of the active disks will exceed the processing power available at the computation server. This implies the target application will be data intensive – that is, they will perform relatively small computation on large volumes of data. In fact, all the active disk work only considers a system with one host and multiple disks.

Carnegie Mellon

Network attached secure disks (NASD) [Gibson et al. 1996; 1997]. are storage devices that allow direct client interaction. For example, they could reduce the load on file servers by letting data operations go directly to the device, while the file server handles the more complex metadata operations. These devices export an object interface to clients, with access controlled through the use of capabilities obtained from the file manager [Gibson et al. 1998].

Performing more general computation on disks is a natural extension of moving some file system functionality to the disks. The growing capabilities of disk drive ASICs, and the possibility of including a microprocessor core on each drive motivated active disks [Riedel and Gibson 1997]. Active disks are disk drives that can execute general-purpose (application) code locally. Applying simple operations (such as filters, collection of statistics) at the disk can improve the overall efficiency of the application's use of network and host resources. This paper briefly discusses the language model, safety issues, and resource management. The main focus is on remote execution of small parts of the computation as opposed to a general parallel computation. The main applications are database select and sorting. Although no results are presented for sorting, the connection with parallel sorting on clusters is interesting because sorting is inherently harder than the other problems proposed in the active disk literature. It also hints at the link with parallel algorithms.

The subsequent paper [Riedel et al. 1998] connects the work with database machines (discussed in Section 1.1.2). A simple analytical model that considers disk read time, interconnect transfer time and server processing time is proposed. This model indicates that active disks are suitable for applications with high selectivity and large datasets (i.e. requiring many disks). Riedel's thesis [Riedel 1999] includes a wealth of information on active disks including related work. Scan-based applications discussed include:

- Nearest neighbor - finds the k entities closest to given entity.
- Frequent sets - counts instances of a condition across a database.
- Classification.
- Edge detection - applies a mask across multiple images.
- Image registration - determines the rotations and translations necessary to align images relative to a base image.

Select, aggregation and selective joins are important database operations. The sorting problem is also presented and merge sort discussed as a solution. Software implementation issues are addressed in the context of the scan-based applications only. The interface simply specifies the function to be applied in parallel for each block of the file. The work at Maryland [Acharya et al. 1998, etc.] considers the programming model in more detail.

Riedel et al. [2000] propose running a data mining application on an OLTP system. The idea is that some blocks can be read for free as the disk head moves to satisfy OLTP requests. The seek time is not increased because the free block is along the path of the head, and the transfer time is part of the rotational delay. They conclude that a third of the disk bandwidth can be obtained for the background (data mining) task without compromising OLTP response time.

Maryland

The Jovian library [Bennett et al. 1994] provided high-performance parallel I/O for multiprocessor architectures with disk arrays. An investigation [Acharya et al. 1996] of several data-intensive problems, including an Earth-science application (satellite imagery manipulation) and an out-of-core matrix factorization found that co-location of data and processing and peer to peer communication were needed for high performance.

A stream-based model of computation for active disks is proposed by [Acharya et al. 1998; Uysal 1999]. A *disklet* (a task executed on an active disk node) takes one or more streams as input and produces one or more streams as output. Disklets have initialization and finalization functions. Security is handled by restricting the functions available on the disks; sensitive operations including I/O are initiated by the hosts. Applications similar to those of [Riedel et al. 1998] are proposed. They also discuss a two-pass external sort based on NOWsort [Arpaci-Dusseau et al. 1997] and note that the (single) host processor can become a bottleneck in some configurations with many (32) disks. Simulation results are based on a disk model, and coarse-grained models of the processor and I/O interconnect.

Three architectural alternatives for processing large datasets are compared in [Uysal et al. 1998], with special emphasis on decision support for databases [Uysal et al. 2000]. The architectures considered are active disks, clusters and SMPs. Active disks and clusters were found to achieve significantly better raw performance for large datasets. As expected, adding more memory to active disks did not provide significant benefit because of the streaming nature of the applications. Because the active disk configuration had a centralized host, direct communication between disks was necessary. Active disks were an order of magnitude cheaper than SMPs and up to an order of magnitude faster; compared to clusters, they had similar performance at half the price.

DataCutter [Beynon et al. 2001] maps the stream model [Acharya et al. 1998] 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.

Berkeley

A database-motivated approach is taken by [Keeton et al. 1998; Keeton 1999]. *Intelligent disks* (IDISKS) are similar to the active disks discussed above, but are more powerful and have more memory. The IDISK concept is based on the database machines (discussed in Section 1.1.2) but the authors argue that they could succeed this time because disk manufacturers are looking for ways to add value, and because of algorithmic advances in shared-nothing architectures. (Shared-nothing architectures were developed after database machines).

Other Work

Ma and Reddy [Ma and Reddy 1998; Ma et al. 2000] evaluate network attached disks for storage systems. They used web server and file system (NFS) traces to compare a system with a single server and multiple (active and normal) disks. Active disks succeeded in reducing the load on the server for both sets of traces. However, they found it necessary to serve all data requests from the disks in order to reduce the load on the server. This in turn meant that server-based caching could not be used (the server cache would absorb 80-90% of the requests). Because the disks had smaller caches, performance was reduced as more data was fetched directly from disk.

The Multi-View Storage System (MVSS) [Ma and Reddy 2001] uses *views* of files to be created and migrated to active disks. The views appear as block-devices. MVSS overloads the device-level block addressing by mapping virtual blocks to unused blocks in the file. It is not clear how this scheme handles mappings that can not be computed quickly; for example, variable bit-rate compression.

1.1.2 Database Machines

Moving processing closer to the storage is not a new idea. *Database machines* were widely studied in the 1970's and 1980's. Database machines relied on special-purpose hardware, and had processing available per head, track, or disk. A survey of database machine architecture is given by [Riedel 1999]. Database machines were effective at greatly improving the performance of simple database queries. However, they failed because of a combination of reasons: (1) The hardware used was expensive and the disk technology was inadequate at the time; (2) Concurrent algorithmic improvements such as indexing proved to be more useful for improving performance; and (3) Commercial database systems did not support them [Keeton 1999].

Although database machines were not a success, they highlighted the importance of using parallelism. *Parallel database systems* [DeWitt and Gray 1992] combine large numbers of commodity components with a scalable interconnect to form a very powerful shared-nothing [Stonebraker 1986] system. These systems have been commercially successful (e.g.: Teradata, IBM SP-2). The specialized functionality required by the earlier database machines is now done by software (*software database machines*).

There were several research projects focusing on software database machines. Gamma [DeWitt et al. 1990] employs three key ideas to obtain better scalability: (1) Relations are partitioned across multiple disks; (2) Hash-based algorithms are used to implement join and aggregate functions; and (3) Dataflow techniques are used to coordinate composite operations. Gamma depended on hash-based algorithms to distribute data among the disks. Many of these are applicable in the context of active disks. The second Gamma prototype [DeWitt et al. 1990] used 32 Intel 386s with 330 MB of storage. In today's hardware, this corresponds to a Pentium III processor and 40 GB of storage, not unrealistic for an active disk. Network bottlenecks were reported as a performance problem for this machine.

Paradise [Patel et al. 1997] is a parallel geospatial database system. The Paradise system consists of one Query Coordinator and multiple Data Servers. Database tables are partitioned using striping, hashes or spatial declustering. The Paradise project focuses on parallel algorithms for geo-spatial databases; for example, spatial join [Patel and DeWitt 1996]. Spatial operations, including spatial join, typically utilize an initial filtering step to eliminate tuples that cannot be part of the result; for example, using the bounding box of a complex object. It is possible that these operations can be performed on active disks.

1.1.3 Parallel Algorithms

Parallel algorithms attempt to extract the maximum degree of parallelism from a problem. This is usually difficult, and sometimes not possible. Furthermore, parallel algorithms are typically designed for a collection of homogeneous processors. Therefore active storage algorithms are not the same as parallel algorithms as we know them. However, useful insights and techniques can be carried over from parallel algorithms. Details of parallel models and programming techniques can be readily found. The following descriptions are based on [Zomaya 1996] and [JaJa 1992].

The complexity of a problem in a parallel model is $T_p(n)$ *basic operations*, where p is the number of processors and n is the input length. Arithmetic and logical operations as well as single memory accesses are usually considered basic operations [JaJa 1992]. The *uniform cost criterion* implies that the cost of an operation is independent of the word length. These assumptions do not hold for real machines, so we need to adhere to the 'spirit' of the model, rather than the letter. In general, an algorithm is said to have *optimal speedup* in a parallel system if $T_p(n) = \Theta\Big(T^*(n)/p\Big)$ where $T^*(n)$ is the best sequential time.

The **parallel random access model (PRAM)** is widely used to design and evaluate parallel algorithms. The PRAM assumes an arbitrary number of processors with a large *shared memory*. All processors operate in lock-step. The **(asynchronous) shared memory model** assumes each processor operates under a separate clock, and requires the algorithm to explicitly synchronize as required. The **LogP model** more closely describes a distributed-memory multiprocessor. The parameters of the model are: L = latency of communication between processors; o = overhead incurred by communication; g = minimal time between successive transmissions; and P = number of processors. Algorithms developed for the LogP model are substantially different from ones for the PRAM model. In particular, low-level handling of messages is usually required. The **block distributed memory (BDM)** model considers a multiprocessor build by combining processors in a hierarchy. The model uses four parameters: p = the number of processors; m = the packet size; τ = the communication latency; and σ = the transmission rate of the network. In this model, k read operations will take $\tau + km\sigma$ time, assuming prefetching.

Some common techniques are repeatedly useful in the construction of parallel algorithms.

- **Balanced Trees** - For example, when computing a sum, combine pairs of values, and then pairs of pairs and so on, until a single result is obtained.
- **Divide and Conquer (Merging)** - Create sub-problems (trivially), solve

them in parallel, and merge the results together (work). Merge sort is the canonical example of this type.

- **Partitioning (Distribution)** - Break the problem into sub-problems (work), solve them in parallel, and then combine the results (trivially). Quicksort is an example of partitioning.
- **Combining** - Sometimes a combination of techniques is useful: One to reduce the problem size, and the other to solve the smaller problem.

One problem with the PRAM model is that memory references are assumed to take unit time; all memory is considered equal. Unfortunately, this may take $O(\log(n))$ time for an arbitrary memory reference from n processors. Blelloch [1989] proposes including *scans* as unit-time primitives. A scan is a parallel prefix operation; that is, for each processor, it computes the 'sum' of the values from the processors 'preceding' it.

Parallel Sorting Optimal parallel sorting algorithms are known, see for example [JaJa 1992]. These algorithms utilize the methods mentioned above. Parallel sorting algorithms are more complex than necessary for the simpler active storage systems. However, they highlight the basic techniques necessary. Randomized algorithms can be simpler or more efficient than their deterministic counterparts. One of the first optimal (n-processor, $\log(n)$-time) sorts is Flashsort [Reif and Valiant 1987]. Key parts of Flashsort find the *splitters* that divide the data into approximately equal parts, and then route the data to the appropriate nodes. Note that sub-logarithmic times are possible if the keys have known properties and radix sort can be used [Rajasekaran and Reif 1989]. The experimental study of sorting algorithms for the CM-2 [Blelloch et al. 1991] found that a Flashsort variant was the most efficient, followed by the simpler counting radix sort [Cormen et al. 1990].

1.1.4 External Memory Algorithms

Chapter 3 describes the I/O model [Aggarwal and Vitter 1988] and external memory (EM) algorithms [Vitter 2001] in detail. Figure 1.1 summarizes the model. Data-intensive applications do a lot of I/O, and EM algorithms try to minimize the number of I/Os. In addition, they usually access data sequentially. As mentioned above, sequential access is usually more efficient. There is a close relationship between parallel algorithms and EM algorithms (for example [Hutchinson 1999]).

Sorting in External Memory External sorting starts and ends with the data stored on disk. A theoretical treatment of external sorting can be found in [Aggarwal and Vitter 1988; Knuth 1998]. Datamation [Anonymous 1985] was the standard sorting benchmark for many years. It sorted one million 100-byte records with 10-byte keys.

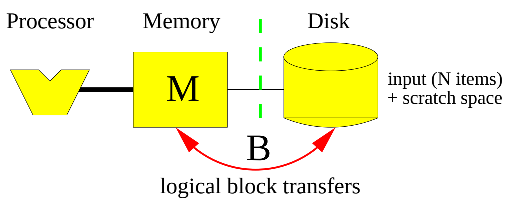

Figure 1.1: The I/O complexity model. Data moves between disk and memory in logical blocks of size B. The processor accesses data only in memory. Complexity in the model is the number of I/O operations needed to solve the problem.

The authors of AlphaSort [Nyberg et al. 1994; 1995] introduced MinuteSort and DollarSort as more realistic modern benchmarks. They are similar to Datamation, but measure how many records can be sorted in one minute or for one Dollar, respectively. The key idea of AlphaSort is awareness of the memory system characteristics. It uses three techniques: (1) Quicksort data as it comes off the disk; (2) Separate keys from the data to reduce data movement; and (3) Merge runs using a replacement-selection tree. Relying on comparisons to order elements slows down sorting. Agarwal [1996] uses bucket-sorting to almost fully overlap computation and I/O. This kind of sort can be scaled easily with higher disk bandwidth (more disks) and extra processing power (active disks, for example).

A theoretical EM algorithm uses two passes to sort a large dataset, because there is not enough memory to cache the entire dataset. (The last record read may be the first in sorted order). This is a **two-pass** sort. Performance-oriented (benchmark) applications typically have enough memory available in the complete system so that the entire dataset can be cached; these are **one-pass** sorts.

1.1.5 Clusters and Distributed Systems

Clusters are built by connecting similar-class machines with a high-performance interconnect. Each cluster node is an independent entity, with a local instance of the operating system. Typically, each node has it's own storage. Clusters built from commodity components offer impressive price/performance ratios.

Sorting is a popular and important benchmark [Gray 2001] for clusters. Cluster sorts are parallelized external memory algorithms. Berkeley's NOW-Sort [Arpaci-Dusseau et al. 1997; 1998a] is such an instance. Although the benchmark versions of this sort had sufficient memory to store the entire problem instance, the general version uses a traditional two-pass EM sort. The cluster nodes could only use two disks because of I/O bus bandwidth limitations. This configuration is very similar to an active storage system with relatively powerful disk nodes that can directly communicate. HPVM MinuteSort [Zhang et al. c1999; Rivera-Alvarez 2000] is based on NOW-Sort, but runs on a cluster with two distinct types of nodes: *Sort nodes* with more memory and CPU power; and *I/O nodes* with more disks. The I/O nodes read the records and distribute them to the sort nodes. The sort nodes sort the records and write them to the I/O nodes. There is no extra communication required because this is a one-pass sort. The I/O nodes in this configuration (4 disks per node) could be replaced with active storage elements.

An obstacle to achieving peak performance is *performance heterogeneity* [Arpaci-Dusseau et al. 1999]. River [Arpaci-Dusseau et al. 1998b; 1999] is a data-flow programming environment for clusters that balances load in the presence performance heterogeneity. The main features of River are a *distributed queue* and *graduated declustering*. The distributed queue connects (possibly) multiple consumers and producers. Ordering constraints may be relaxed for better performance, essentially creating a *distributed set* (see Section 1.4). Graduated declustering is based on the *chained declustering* idea from database machines. Data is mirrored so that two disks can satisfy a request; for example, host i might be able to read from disks i and $i + 1$, and host $i + 1$ might be able to read from disks $i + 1$ and $i + 2$. This forms a chain, so that any drop in performance can be amortized by splitting reads between disks, equalizing the performance of all the hosts. These features have general applicability, but also introduce overhead by their generality. In practice, buffering can absorb transients, so only long-term imbalances are of concern We hope that limiting this generality, or using more approximate solutions will allow better performance.

Abacus [Amiri et al. 2000a;b; Amiri 2000] attempts to balance load by moving the computation, rather than changing the data flow. Mobile functional units can be moved using the *checkpoint/restore* mechanism. Important components of Abacus monitor load and decide when to move computation. This contrasts with the data-flow approach of River. Because I/O algorithms tend to use multiple *phases*, each consisting of several filter-like functional units that operate on streams, we believe a data-flow approach is a better match.

Ninja [Gribble et al. 2001] provides scalable cluster-storage for distributed Internet services using Distributed Data Structures (DDS) [Gribble et al. 2000]. 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. Internet services process many small independent requests, making them naturally parallel; Ninja is not concerned with extracting parallelism in large individual computations.

1.2 Active Storage

Current schemes for dealing with heterogeneity in systems fall into two categories: (1) Static allocation of work; and (2) General-purpose mechanisms for dynamic load-balancing. We believe it is possible to combine the advantages of both systems by incorporating 'hooks' into the algorithms. This requires more structure in the programming model, and tunable-algorithm designs.

Adding parallelism to existing EM applications is the logical next step. The TPIE system currently allows EM algorithms running on a single-processor system to adapt to varying amounts of memory. We will extend this concept to processing, communication and storage capabilities. EM applications like TerraFlow can be parallelized to a limited degree, making them suitable subjects for further experimentation.

1.3 An Analytical Model

Suppose we have a ADPs (active disk processors) labelled $\mathcal{A}_1, \ldots, \mathcal{A}_a$, and h host computers labelled $\mathcal{H}_1, \ldots, \mathcal{H}_h$, interconnected by a Storage Area Network (SAN) as shown in Figure 1.2. ADP $\mathcal{A}_i$, $0 \leq i \leq a$ contains internal memory of size M_i items, and has D_i attached disks labelled $\mathcal{D}_{i,1}, \ldots, \mathcal{D}_{i,D_i}$. $\mathcal{H}_j$, $0 \leq j \leq h$ contains internal memory of size m_j items. We assume that the disk and host processors are capable of executing a common, canonical set of instructions, but in general, the speed at which they can execute these instructions varies according to the processor.

For modelling purposes, we select a distinguished (perhaps imaginary) processor $\mathcal{P}$, and a representative instruction of $\mathcal{P}$. We refer to this instruction as a *standard instruction*, and the time for this instruction to be executed by $\mathcal{P}$ as a *standard instruction time*(sit).

Communication, computation and I/O are assumed to be synchronous in the model. There is no overlap between them.

The interconnection network supports $d = (a + h)/2$ simultaneous point-to-point two-way conversations (between different disk processors, between different host processors, or between a disk processor and a host processor) at link speed Γ_0 items per sit. We assume that the maximum bandwidth Γ_0 of an individual conversation is attainable only for large messages of size at least b characters, and that smaller messages incur the same delay as a message of size b. The rationale for such assumptions is that there is some fixed message overhead due to switching delay and control information, and that there is also a per-packet delay at the source and destination attributable to the processing in the TCP protocol. See [Chase et al. 2000] and [Bäumker et al. 1998] for related discussions.

In a single *communication round* a single block of size b can be sent and received by each processor over the network at a cost of g sits.

Algorithms on our model operate in a manner characteristic of algorithms on the Bulk Synchronous Parallel (BSP) model of Valiant [Valiant 1990]. Computation proceeds in a series of "supersteps", each consisting of a computation step and a communication step. During a computation step, all processors operate independently, solving a subproblem using data available locally at the beginning of the step by

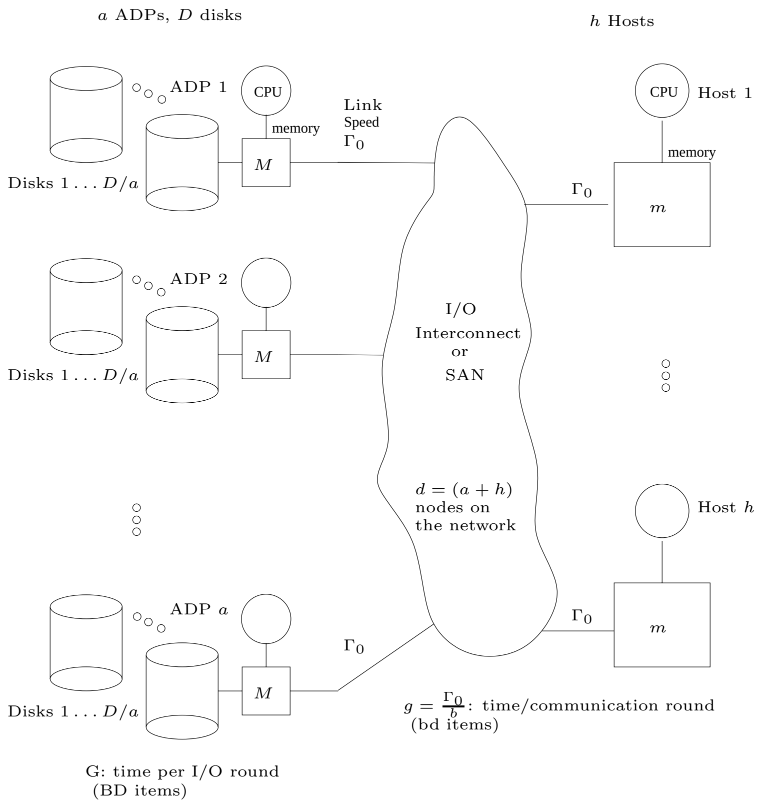

Figure 1.2: Active disks: The figure shows h host processors on a SAN with a active storage units. Host processors have internal memory of size m items. Storage processors contain internal memory of size M items, and have D/a attached disks, giving D disks in total on the network. In one I/O round, D blocks of size B (one per disk) can be transferred into the memory of the ADPs at a cost of G time units. In one communication round, $d = a + h$ blocks of size b can be sent and received by each processor at a cost of g time units.

performing local computations and I/O operations. During a communication step, they exchange data among themselves.

We use the following additional parameters to characterize the cost of a computation $\mathcal{C}$ on our model. For every processor, $\mathcal{A}_i$, $0 \leq i \leq a$ and $\mathcal{A}_j$, $0 \leq j \leq h$ we record:

- β_0 : the number of computation operations performed, and β, the equivalent time, expressed in sits. β is therefore the execution time, expressed as the number of cycles of our distinguished processor $\mathcal{P}$.
- α : the number of communication operations of size b,
- κ : the number of parallel I/O operations (for host processors this quantity will be zero).

We also refer to the following quantities:

- g : the time, expressed as the number of sits, for a "block" of b bytes to be sent or received over the network by any processor. The parameter g is an upper bound and is applicable to both disk and host processors. Since $g = O(\frac{\Gamma_0}{b})$, we drop the parameter Γ_0 in favour of g in most discussions of the model.[1]
- G : the time, expressed as the number of sits, for a block of size B data items to be written to, or read from every local disk (in parallel) by a disk processor. The parameter G is an upper bound and is applicable only to disk processors.

The total processing time for a disk processor $\mathcal{A}_i$, $1 \leq j \leq a$ on a single superstep s of $\mathcal{C}$ is given by $T_{A_i}(\mathcal{C}(s)) = \beta_{A_i}(s) + \alpha_{A_i}(s) \cdot g_{A_i} + \kappa_{A_i}(s) \cdot G_{A_i} + L$. Similarly, the processing time for a host processor $\mathcal{H}_j$, $1 \leq j \leq h$ on s is $T_{H_j}(\mathcal{C}(s)) = \beta_{H_j}(s) + \alpha_{H_j}(s) \cdot g_{H_j} + L$. The parameter L represents the time required for the processors to synchronize at the end of a superstep.

Since communication and I/O are both performed in sufficiently large blocks (size b and B respectively), we do not separately represent the latency of these operations in the model. In Section 1.3.1 we discuss a number of assumptions that we can make to further simplify the model.

The overall maximum time for superstep s is therefore

$$T(\mathcal{C}(s)) = \max\left\{\max_{1\leq i\leq a}\{T_{A_i}(\mathcal{C}(s))\}, \max_{1\leq j\leq h}\{T_{H_j}(\mathcal{C}(s))\}\right\}.$$

If we have λ such supersteps in $\mathcal{C}$, the overall cost of the computation on our model is $T(\mathcal{C}) = \sum_{s=1}^{\lambda} T(\mathcal{C}(s))$.

[1] $\frac{\Gamma_0}{b}$ plus an allowance for setting up the switched connections and other overhead.

1.3.1 Simplifying the General Model

Our basic model already incorporates the latencies associated with communication and disk I/O operations in the values of the parameters b, B, g, and G. In practice we can also eliminate the parameter L, representing the latency of a superstep by recognizing that in practice the processors will run asynchronously whenever possible. Each processor need only ensure that it has received all of the data sent to it in the previous superstep before proceeding with the next one. This is not difficult to ensure if we adopt the convention that each processor sends at least a minimal message (equivalent to a message of size b) to every other processor at the end of a superstep.

Let c_A be the computation time of a disk processor for a representative operation, and let c_H be the corresponding execution time of a host processor. We assume that $r = \frac{c_A}{c_H} < 1$.

We further simplify the model by assuming that

- There are D identical disks on the network in total, and $G_{A_i} = G$ for all i.
- Every disk processor is identical, has the same number D/a of disks, and the same amount M of internal memory.
- Every host processor is identical, and has the same amount m of internal memory.
- We assume that the difference in speed between ADPs and host processors does not affect their ability to move data through the network at the per-connection bandwith limit. Since all of the processors are connected to the same network, $g_{A_i} = g_{H_j} = g$ for all i, j.

Incorporating these simplifications, the cost of a computation $\mathcal{C}$ on the model is

$$T(\mathcal{C}) = T_{comp} + T_{comm} + T_{I/O} \tag{1.1}$$

where

$$\begin{aligned} T_{comp} &= \beta & (1.2) \\ T_{comm} &= \alpha g & (1.3) \\ T_{I/O} &= \kappa G & (1.4) \end{aligned}$$

and

$$\beta = \sum_s \max\{\max_{1 \leq i \leq a}\{\beta_{A_i}(s)\}, \max_{1 \leq j \leq h}\{\beta_{H_j}(s)\}\} \tag{1.5}$$

$$\alpha = \sum_s \max\{\max_{1\leq i\leq a}\{\alpha_{A_i}(s)\}, \max_{1\leq j\leq h}\{\alpha_{H_j}(s)\}\} \quad (1.6)$$

$$\kappa = \sum_s\{\max_{1\leq i\leq a}\{\kappa_{A_i}(s)\} \quad (1.7)$$

1.4 Programming Model

We are now ready to discuss the abstractions that are necessary and useful in implementing algorithms for active storage. The number and pattern of data accesses determine overall performance (given sufficient processing power). The data accesses of EM algorithms can be characterized as follows.

- Scanning (unordered) - performs an operation on each element of the dataset. This may be part of a selection or aggregation. The actual order of accesses is not important.
- Sorting - enforces an ordering constraint on the data.
- Scanning (ordered) - performs order-sensitive *scan operations* [Blelloch 1989]. Examples include prefix-sum and merging.
- Scattered access - could access elements anywhere in the dataset (typically blocked). For example, traversing index structures, following graph edges.

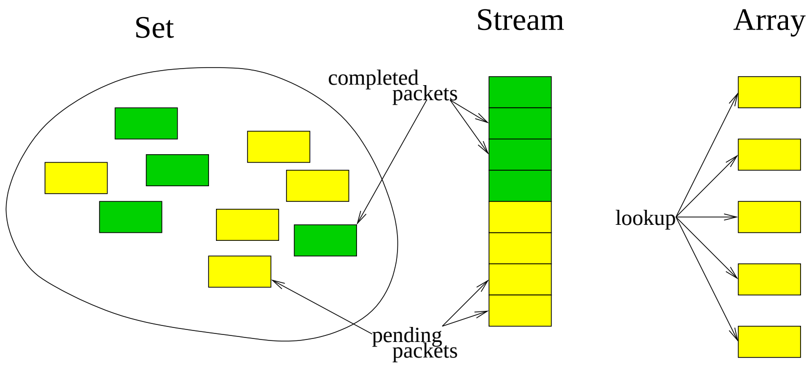

Figure 1.3: Three basic containers. Sets and streams have completed and pending packets. Arrays require a lookup to locate packets.

We can argue for three types of data container based on how they are accessed.

1. **Sets** are accessed without a specific order. Reads may return items in an order different from the written order. This allows read and writes to take place in parallel.

2. **Streams** are processed in a specific order. In particular, a sequence of reads will return items in the same order as written. This means that reads and writes have to be serialized.

3. **Arrays** allow random access to blocks when it is necessary to access blocks in a specific order. This requires the system to find any given block quickly.

Sets and streams occur most frequently in the algorithms we discuss, so we will focus on them here. Arrays are necessary to support indexing structures. Sets offer the greatest room for optimization because they can be created by several nodes operating in parallel; all nodes can add to a set at the same time. A stream can only be created by one entity, since, by definition, it has some global order; a single node has to synchronize writes in order to enforce an ordering constraint. Traditionally, sets and streams have been considered together because data is stored linearly, and thus always has a implicit order.

Queues Processing stages may be connected in a pipeline using sets and streams. Streams connecting a reader to a writer are queues. In some cases, it is not necessary to maintain ordering between writers and readers, so sets may be used instead on streams. This is similar to the distributed queues in River.

Write-once Sets and streams are write-once; once an object is created in its entirety, it becomes immutable. The rationale is that sets and streams are accessed in their entirety, and then discarded. For example, the primary operations we are concerned with (e.g.: merging, distribution, sorting) always process sets and streams in their entirety, and then discard them. If 'stream with update' functionality is required, arrays can be used.

Downcasting Streams can always be used as input in place of sets, but this usually indicates that resources were wasted when creating the stream, because the ordering imposed by the stream is not being used.

1.4.1 Methods

Read and write functions can be used to access all three data types. Because sets do not impose ordering, writes can take place in parallel, and reads can be parallelized for most of the dataset. Synchronization may be required depending on the semantics of end-of-file (EOF) notification. Streams have much more overhead because they preserve ordering. Exposing this to the algorithm designer should lead to better designs. Finally, the array type is completely synchronous. These properties are summarized in Table 1.1. We have not yet considered the question of determining when operations are sharing and object, and when they are independent.

Data Type	Reads	Writes (Restriction)
Set	Parallel (usually) Serialized (EOF)	Parallel (Add only)
Stream	Serialized if shared; Parallel if independent	Serialized (Add only)
Array	Parallel	Arbitrary (Unrestricted)

Table 1.1: Restrictions on access patterns for different data types. Parallel means that any node may perform the operation independently of the other nodes. Serialized means that there is a single serialization point. Arbitrary means that no restriction is imposed.

Traditional read operations simply return data, EOF or an error condition. In some cases, it may not be possible to determine if the object has been completely read, even though no more data is currently available. This leads us to propose a status value that could indicate: (1) More data is available now; (2) No more data in the object (EOF); or (3) No data is available at the moment, but the object is still open for writing so more may be available later. Cookies passed with the read and write functions can minimize the application state maintained by the system. The basic read and write functions could then be sketched as follows.

- `readSet(setID, cookie)` → `(status, cookie, data)`
- `writeSet(setID, cookie, data)`
- `readStream(streamID, cookie)` → `(status, cookie, data)`
- `writeStream(streamID, cookie, data)`
- `readArray(arrayID, offset)` → `(data)`
- `writeArray(arrayID, offset, data)`

1.4.2 Processing on Disk Nodes

Functors are abstract computational units that can be located on disk nodes (also known as *disklets*). It is possible to read and write through a functor, which may perform some computation on the data. A read on a filter may result in other writes (distribution).

In order to facilitate resource provisioning, functors process data as a side-effect of I/O operations using bounded resources. Read/modify/write operations are currently limited to specific kernels like sorting. Functors operate on *packets* of records and require resources proportional to packet size. For example, sorting a packet of r records with quicksort uses $O(r \log r)$ CPU and no extra memory. Application functors may consume or emit packets of records. For example, packets may enclose groups of records with some global property defined, e.g., they are sorted. (see

Figure 1.4).

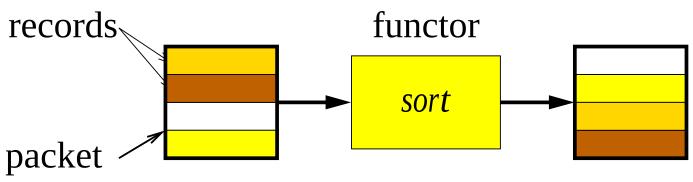

Figure 1.4: Functors operate on packets of records. The sort functor can use packets to preserve local order.

The basic functions to read and write sets and streams using filters are as follows.

- `readSet(setID, functor, cookie)`
- `writeSet(setID, functor, data)`
- `readStream(streamID, functor, cookie)`
- `writeStream(streamID, functor, data)`
- `readArray(arrayID, functor, position)`
- `writeArray(arrayID, functor, data)`

In some cases, functors may need to be restricted for efficiency in computing offsets in the result. For example, random access to the result of a selection filter could be very inefficient.

We would like to make full use of the processing capabilites of the disk nodes while keeping the interface as simple as possible. The following section discusses different primitive operations that may be exported to the disks, and how each higher level requires more functionality be available to functors. The above methods may not be adequate to support this added functionality.

1.4.3 Functional Primitives

Active disk algorithms frequently use several standard primitive operations to synthesize the final algorithm. Table 1.2 lists some common primitives.

- SCAN performs a per-item computation; for example, an arithmetic operation, aggregation or filtering. Scans on sets and streams should require unordered and preserve ordered semantics of the applied function respectively.

- MERGE combines multiple streams to form a single stream. $\text{MERGE}_w(n)$ merges w streams into a single stream of n records. Merge only makes sense for streams since it assume a relationship between the current input elements at all times.

- DISTRIBUTE partitions a set into multiple (sub)sets based on *splitters*. DISTRIBUTE$_w(n)$ partitions n records into w buckets of expected size n/w. Partitioning and finding suitable splitters is an interesting problem in itself (for example, B-Flashsort [Hightower et al. 1992]).
- BLOCK-SORT is another useful primitive we will use in run formation. BLOCK-SORT$_w(n)$ simply sorts blocks of size w, forming a total of n/w sorted runs. BlockSort is really a scan operation that creates a partially ordered set as output.

1.	`scan(inSetID, functor)`
2.	`distribute(inSetID, k-distributeFunctor, outSetID[k])`
3.	`merge(inStreamID[k], k-mergeFunctor, outStreamID)`
4.	`sort(inSetID, compareFunctor, outStreamID)`

Table 1.2: Common primitives

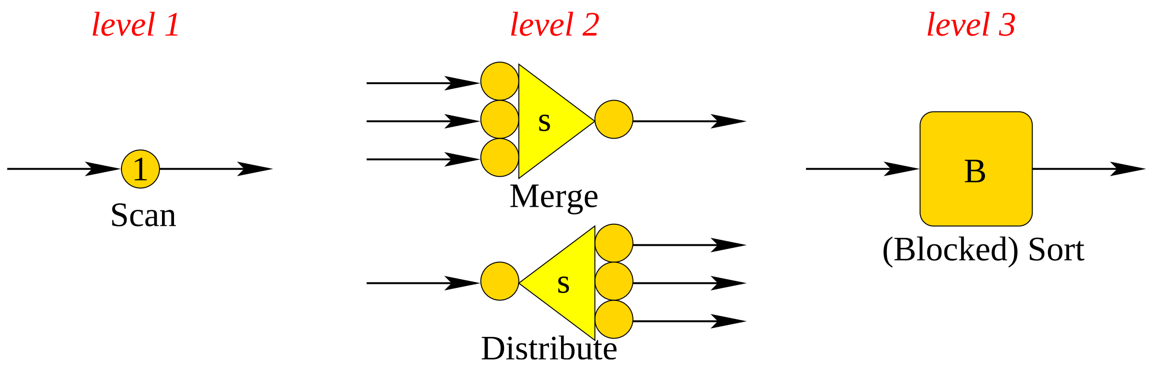

Figure 1.5: Examples of possible operations at the first three levels in the computation model. A merge or distribute with s streams requires $\Theta(s)$ space. A sort of a block of B records requires $\Theta(B)$ space.

The ADP computation model [Hutchinson et al. 2001] defines several levels of processing capabilities for disk nodes. Figure 1.5 provides examples for the first three levels. Level 1 permits streaming operations such as simple filtering, computing aggregates and other operations that access records sequentially. Level 2 also only permits streaming operations, but allows multiple input and output streams to be accessed in parallel. This allows distribution and merging; they can read one record, and them output one record (after an initial startup period). Level 3 allows arbitrary operations on large chunks of data. These operations are streaming at a courser grain (read and output a chunk instead of record).

operation	extra space	CPU (per item)
SCAN	$\Theta(1)$	$\Theta(1)$
MERGE (s-way)	$\Theta(s)$	$\Theta(\log(s))$
DISTRIBUTE (s-way)	$\Theta(s)$	$\Theta(\log(s))$
BLOCK-SORT (B-sized)	$\Theta(B)$	$\Theta(\log(B))$

Table 1.3: Resource requirements of some common operations.

Table 1.3 lists the resource requirements for several common operations. MERGE, DISTRIBUTE and BLOCK-SORT are of approximately equal cost when $s = B$ because $\text{MERGE}_w(n) = n \cdot \text{MERGE}(w) = n \log w$, $\text{DISTRIBUTE}_w(n) = n \cdot \text{DISTRIBUTE}(w) = n \log w$ and $\text{BLOCK-SORT}_w(n) = n/w \cdot \text{SORT}(w) = n/w \cdot (w \log w) = n \log w$.

1.4.4 Resource Management

Performance heterogeneity is an obstacle to achieving peak performance. Imbalances can occur because of data skew, when ASUs are shared between applications, or if nodes have different performance characteristics. Many data-intensive applications like TerraFlow (see Section 1.5) are data-dependent. ASUs may be shared between applications that have dedicated hosts (or vice versa). Static differences can be corrected relatively easily. Dynamic load balancing techniques usually add additional overhead because they are built on top of the programming model. Our design exposes opportunities for dynamic load balancing that can be handled in the infrastructure.

Sets and replicated functors allow nodes to perform dataflow routing operations intelligently. Data dependencies can be eliminated by using randomized routing techniques like *simple randomization* (SR) [Vitter and Hutchinson 2001]. SR uses a round-robin policy with different starting points. Capacity-aware policies are required to handle nodes with differing processing rates. Bounding computation time for functors on the ASUs facilitates these decisions.

Building applications from processing elements that dynamically split between ASUs and hosts allows functionality to be moved without moving application objects. `Classify`, `distribute`, `merge` and other hierarchical operations are examples of this kind of processing.

Load-Managed Data Structures, similar to the Berkeley Distributed Data Structures, will make it easy to parallelize exisiting applications. Obviously this will only apply in selected cases. However, the limited processing available at the storage elements combined with the stream-based programming model means that this could be an important approach.

1.5 Distributed Applications

Data-intensive applications that make predictable accesses to data are potential candidates for an active storage system. Sorting has already been considered in the preceding chapters as the prototypical data-intensive application. Other applications can be found in geographic information systems (GIS), computational geometry, spatial databases, and image processing.

The truly massive datasets of today are generated by automated sensing devices. These datasets can not be used directly because of their size, and various methods of summarizing the data are required. Two large collections of large datasets are described below.

- NASA's Earth Observing System - Data and Information System (EOSDIS) [NASA 2002] collects and manages data from NASA Earth science research satellites and land observations. The current database is several hundred terabytes, and is growing at hundreds of gigabytes per day. Data products include surface images and elevation models.
- The Sloan Digital Sky Survey (SDSS) [SDSS 2002] is the most ambitious astronomical survey project ever undertaken. The survey will map in detail one-quarter of the entire sky, determining the positions and absolute brightnesses of more than 100 million celestial objects. It will also measure the distances to more than a million galaxies and quasars. SDSS is expected to contain approximately 40 TB of raw data that will be accessed directly or via various indexes [Szalay et al. 2000].

As mentioned earlier, many GIS applications are of interest because they process large volumes of data which performing relatively simple computations. While we are actively searching for other test cases, we will redesign the following three applications (in addition to sorting) to use active storage.

1. Differencing - motion detection, feature extraction.
2. TerraFlow - delineating watersheds.
3. Triangulation - raster to TIN conversion.

The following sections discuss each application in more detail.

1.5.1 Differencing

Image differencing is a popular way of identifying 'interesting' components of a sequence of images. Common applications include remote surveillance systems, data compression, medical image analysis and satellite image analysis (GIS applications). Temporal image sequences may be differenced by comparing adjacent frames, or by comparing with a background image. A filtering stage after the diff removes noise. The quality of the result is dependent on the diff procedure and filtering used. Cutoff values for filtering may also be computed based on the local image.

1.5.2 TerraFlow Watershed Delineation

Delineating watersheds is a fundamental component of TerraFlow. Many hydrological processes occur within watersheds, so this allows us to partition the problem into smaller subproblems. Watersheds are identified by processing points in reverse topological order (based on flow directions) and propagating the drain point backwards to identify the watershed. This corresponds to pushing flow (water) upward from local minima, The relationship between topological order and elevation allows TerraFlow to obtain a topological order by simply sorting.

Computing watersheds in parallel is much harder because we can no longer reduce TOPOLOGICAL-SORT to SORT and still partition the work. However, each watershed exhibits good locality, so partitioning by local minima could yield an efficient parallelized solution.

1.5.3 Raster to TIN Conversion

Terrain data is usually collected by remote sensing devices as raster data (grids). However, grids are an inherently inefficient representation of a terrain. TINs can represent a terrain with fewer points, and consequently lend themselves to more efficient computation. A grid may be trivially converted to a TIN by creating two triangles to represent each grid-cell. This does not reduce the number of points in the result. We would like to derive a triangulation that reduces the number of points as much as possible, yet limits the error introduced.

Parallel algorithms for Delaunay triangulation exist, though many have practical limitations. [Blelloch et al. 1996] presents a projection-based approach that works by recursive partitioning using the median line. There are also approaches that work well with uniform distributions. These approaches may be amenable to triangulation of large grids.

1.6 Sorting

Sorting is an important operation in many applications. EM algorithms make heavy use of sorting to organize their access patterns. Berkeley's NOW-Sort is the standard sorting algorithm for clusters. This section formally applies the distribution and merge-sorting techniques to active storage. It also presents an alternate heapsort approach based on priority queues.

1.6.1 DSM-Sort

DSM-Sort is a combination sort using both partitioning and merging paradigms. A sorting algorithm may be composed of a combination of the three basic ordering operations: DISTRIBUTE (D), BLOCK-SORT (S) and MERGE (M). The natural order for these operations is D-S-M; it does not make sense to distribute already ordered records, or to sort (partially) sorted records, etc. We will therefore discuss the DSM-sorting algorithm. Note, however, that not all three operations may be present in a particular instance; for example, dropping the DISTRIBUTE gives a merge-sort (S-M), and dropping the MERGE gives a distribution-sort (D-S).

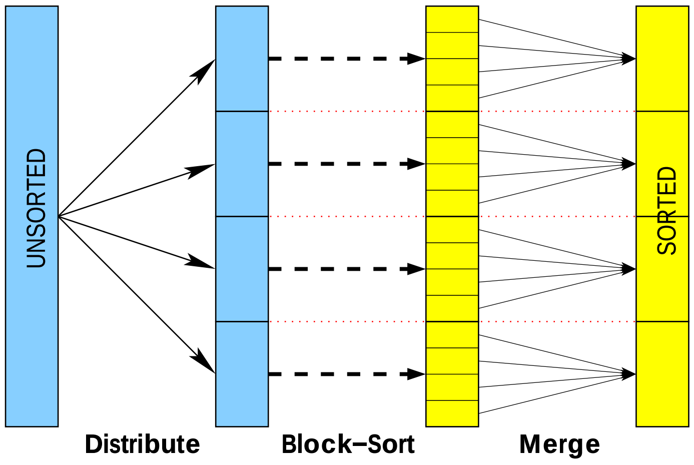

Figure 1.6: General procedure for sorting

DSM-SORT separates the data into buckets, and then sorts each bucket by forming sorted runs and merging the runs in a final (possibly multi-pass) merge. In an active storage system, this work has to be split between the disk nodes and the host nodes. In particular, we would like to keep all the components of the system maximally utilized in order to improve overall efficiency. Figure 1.6 illustrates an instance of DSM-SORT. In this example, DISTRIBUTE partitions the dataset into four subsets (or *buckets*). BLOCK-SORT then forms sorted runs within each subset (4 in each subset, in the illustration). Finally, MERGE combines runs within each subset to form the final result.

The general idea is that the sort will take place in several *passes*. In one pass, the data will flow from the disks to the hosts and back to the disks, passing through one or more DISTRIBUTE, MERGE and/or BLOCK-SORT routines. Ideally, we would

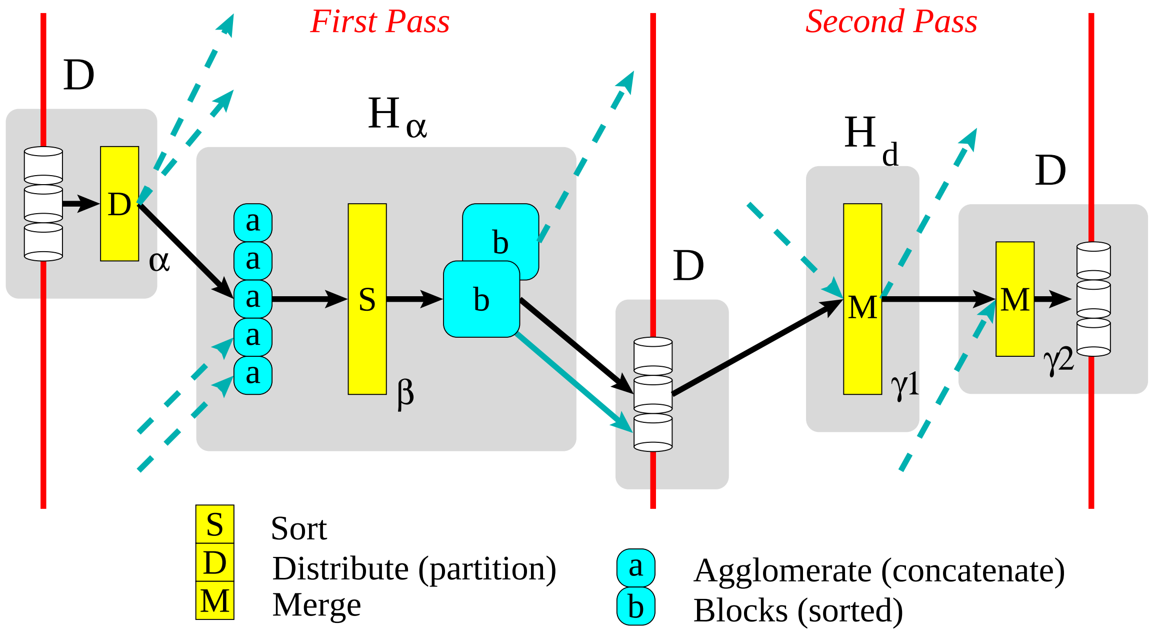

Figure 1.7: Sorting with active disks – illustration of a two-pass sort performing DISTRIBUTE-SORT-MERGE.

like to do $w_d \cdot w_h$ work in each pass. In order to keep the disk nodes working during the first pass, we we have the choice of performing one or more of the following three operations.

- DISTRIBUTE while reading the data.
- BLOCK-SORT while reading or writing.
- MERGE while writing the data.

DSM-SORT consists of three phases; the first two phases constitute the first pass, and the third phase, the second pass.

1. Perform an α-way DISTRIBUTE to partition the data into α sets. (We hope that $\alpha = \kappa \cdot d$; where κ is a positive integer and d is the number of disks.) We can do part of distribution on the hosts if the disks are not powerful enough.

2. Use a suitable fast internal sort on blocks of β items to form a total of n/β sorted runs within the α sets. Run-formation takes place on the hosts, and a suitable in-core sort such as RADIX-SORT should be used.

3. Use a combination of concatenation (if required) and a γ-way MERGE to form d sorted runs. Note that concatenation is only required if $\alpha > d$, and we have more buckets than disk nodes.

1. Let partitioning assign *colors* $(1, 2, \ldots \alpha)$ to items (blackbox; assume fairly even division – can we compensate for bad division?). The color (modulo the i factor) of an item corresponds to the identity of the disk it occupies at the end of the sort.
2. Partition the data by color, and form sorted runs on hosts.
   - disk nodes distribute to host nodes. If disks have at least $d + h$ buffers available they may perform the coloring. All data of color i may go to a single host, or be striped(?) across several or all. Depending on the relative resources of the disk nodes they may partially sort the data before shipping to hosts.
   - host nodes MERGE (or agglomerate) data from disk nodes (keeping colors in separate buffers) and sort buffers as they fill.
   - hosts can empty buffers by performing step-2, which drains host buffers to the disk nodes
3. Move data to disks so that disk nodes $D - \{d_i\}$ will contain all data with color i in sorted order.
   - host nodes distribute entire buffers to disk nodes based on color and striping(?) policy.
   - disk nodes MERGE buffers received from host nodes to form one run of each color.
4. Merge data for each disk from the other disks (d_i gets all data of color i from the other disks $D - \{d_i\}$).
   - if D is small, hosts are idle and MERGE takes place on disks
   - if D is large, hosts perform MERGE and send data to disk

Figure 1.8: Steps in algorithm DSM-SORT

Figure 1.7 illustrates the operation of the algorithm when only a single merge pass is required. In the first pass, disks distribute data into buckets. Each bucket is block-sorted and the sorted blocks are stored on the disks. In the second pass, These sorted blocks (in each bucket) are then merged, giving the sorted result. The merge is divided between hosts and disks, so that $\gamma_1\gamma_2 = \gamma$.

Figure 1.8 described the algorithm in more detail. The total work done is parameterized by α, β, γ:

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

In particular, this means that

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

Data is pipelined through these stages to the greatest extent possible. It is expected that the entire dataset will not fit in memory. Given a system with adequate bandwidth and reasonable parameters, this means that the sort must be done in two passes. Performing the sort in two passes has the additional effect of reducing the memory requirements of the sort. For example, sorting 60M 100-byte records with a 60MB host results in only 100 buckets, which in turn can be easily merged in 60MB.

1.6.2 Heapsort

Heapsort works by simply inserting all the elements into a priority queue (PQ) and then extracting them in sorted order. Heapsort has good performance, but poor cache (disk) behaviour. The advantage of a PQ-based technique is that the `insert` and `extract_min` operations can do most of their (heap maintentance) work in the background. We propose to adapt the I/O-efficient external-memory PQ by Toma [Toma 2001] based on the RAM-model algorithm by Thorup.

1.7 Evaluation by Emulator

We are interested in the performance changes that occur as the parameters of an active storage system change. Real hardware is tied to the current technology, and can only instantiate a small subset of the parameter space. Furthermore, future technology is not available to researchers. An emulator lets us examine the effect of different algorithm configurations as parameters of the system change, and explore arbitrarily complex systems on arbitrary datasets.

The interface library handles the system support required to distribute computation throughout the active storage system, perform I/O on the ASUs, and communicate between nodes.

Figure 1.9 illustrates the overall structure of the emulator. It is instrumented to measure performance, and resource utilization. Real service demands are determined by executing application code on the emulation host. The emulation interface library simulates disk I/O and communication operations. The parameters to the emulator include the number of hosts and ASUs and their CPU speed as compared with the emulator host.

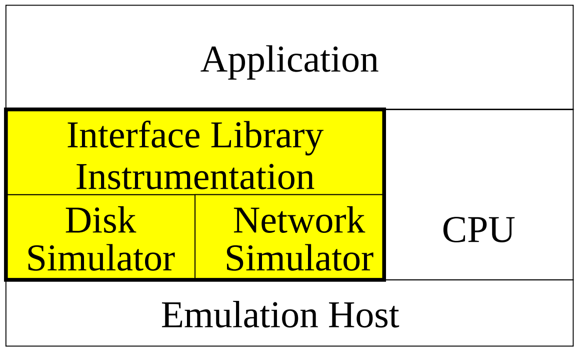

Figure 1.9: Emulator organization.

We can compute the queueing delays and initialization and completion times for each process on each processor using simulation models. We can use simple simulation models because the I/O applications we consider have regular sequential access patterns. The disk simulation uses the transfer rate to calculate time used. It includes support for simple read-ahead and write caching for sequential I/O. This means we assume that the disk initiates the *next* I/O automatically, and that writes only wait for the *previous* write to complete. A full disk simulator could easily be integrated (as in [Griffin et al. 2002]) using conditions (below). The network model only uses host-ASU communication, and assumes that the processor saturates before the individual network links. This is realistic given our processing requirements, but can become significant for wide area computation. If the interconnect bandwidth is limited, direct ASU-ASU communication may be required [Acharya et al. 1998; Uysal 1999].

Execution time for the application is divided into segments between calls into the library. Fine-grained cycle counters measure physical time within each segment. This is then scaled appropriately for the simulated node. Because our interface library supports synchronous (blocking) operations, the current implementation uses individual threads to store execution context for each node in the system. ASUs and hosts are simulated entities with separate memory buffers, thread pools and clocks. A globally consistent view of time is maintained when nodes communicate, so dependencies are maintained and overall run time can be determined. The emulator includes an event queue to ensure that context switching takes place in temporal (causal) order. Condition *wait* is supported by posting events at $t = \infty$ and having *signal* update t. Modifications to the programming model semantics under consideration could allow more efficient implementations. An event-driven model would offer better performance, but require application-level modifications to handle asynchronous operations.

1.8 Experimental Results

These experiments show the effect of the two forms of adaptation. Records are 128 bytes and keys are 4 bytes. The emulation host is a 750MHz Intel with 256MB. We simulated hosts (equal to the emulation host) with performance $c = 4, 8$ compared to ASUs. Timings are for the first pass of sorting.

Figure 1.10 shows the speedup obtained by different configurations of DSM-Sort as ASUs are added to the system. The base for comparison is the non-active disk configuration. This system has one host that is 8 times faster than an individual ASU. Initially, the host has excess processing capacity, so increasing the work on the ASUs actually hurts performance. As ASUs are added, they saturate the host, and it makes sense to move increasing amounts of work to the ASUs.

Figure 1.11 shows the effect of data skew. The first half of the input data is drawn from a uniform distribution, while the second is from an exponential. The base version of the application statically partitions the data between hosts. The balanced version uses SR (Section 1.4.4) to eliminate data skew. Different load balancing methods can be used, depending on the amount of information available.

1.9 Conclusion

As we build larger and larger storage systems, information extraction becomes a significant challenge. Data-intensive applications sift through large datasets and search for, or derive, useful information. Large storage systems are built by aggregating many storage elements. Active storage can reduce the load on the rest of the system by performing part of the computation directly at the active storage element.

External memory (EM) algorithms attempt to access storage efficiently by better modeling of the storage system. In many cases, this results in programs that fit well into a data flow programming model. Interface libraries allow easy implementation of EM algorithms. A richer programming model can allow programs to make use of active storage, while avoiding the need for a general parallelized implementation.

Our study has three related components. First, we will investigate the properties of several problems in an active storage setting and characterize them based on how they fit the active storage paradigm. Second, we will attempt to identify the parameters of these systems and how they relate to performance. Third, we will propose a set of programming primitives that allows easy implementation of active storage

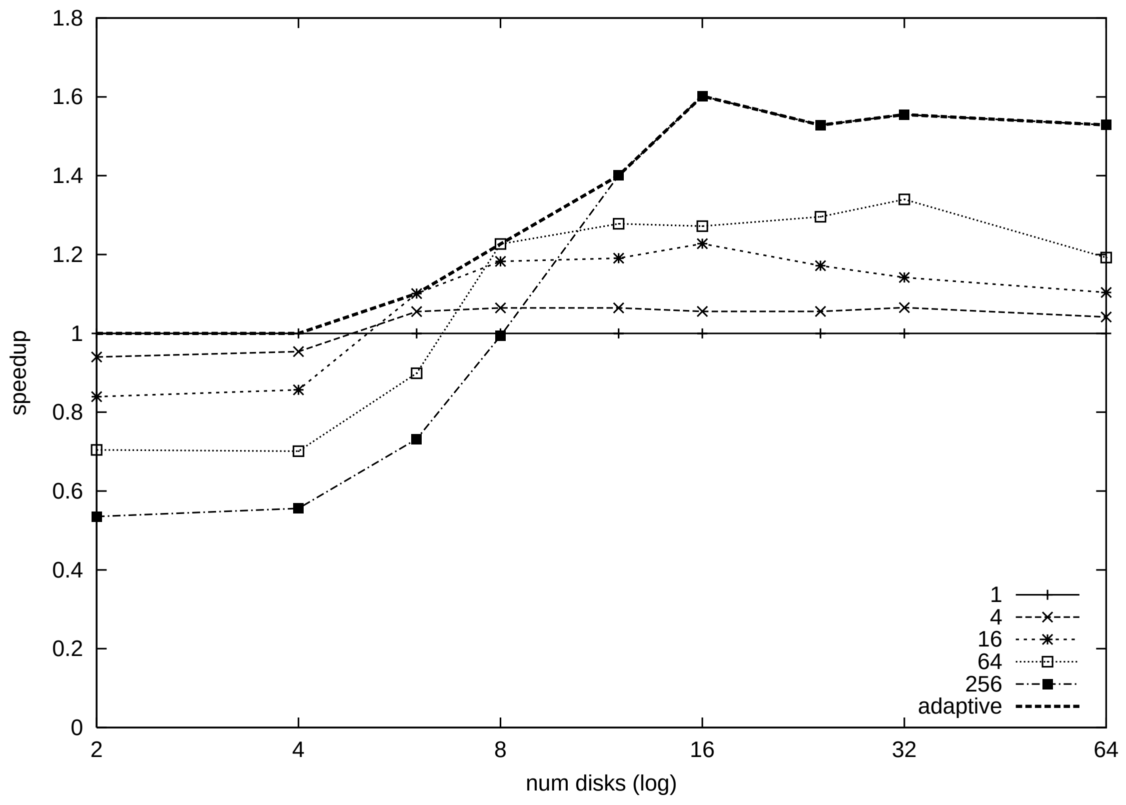

Figure 1.10: Speedup achievable in DSM by adaptively configuring the mapping of function to CPUs as ASUs are added. Data series represent different configurations (α values) of the algorithm. This experiment uses one host, which limits the performance with more ASUs.

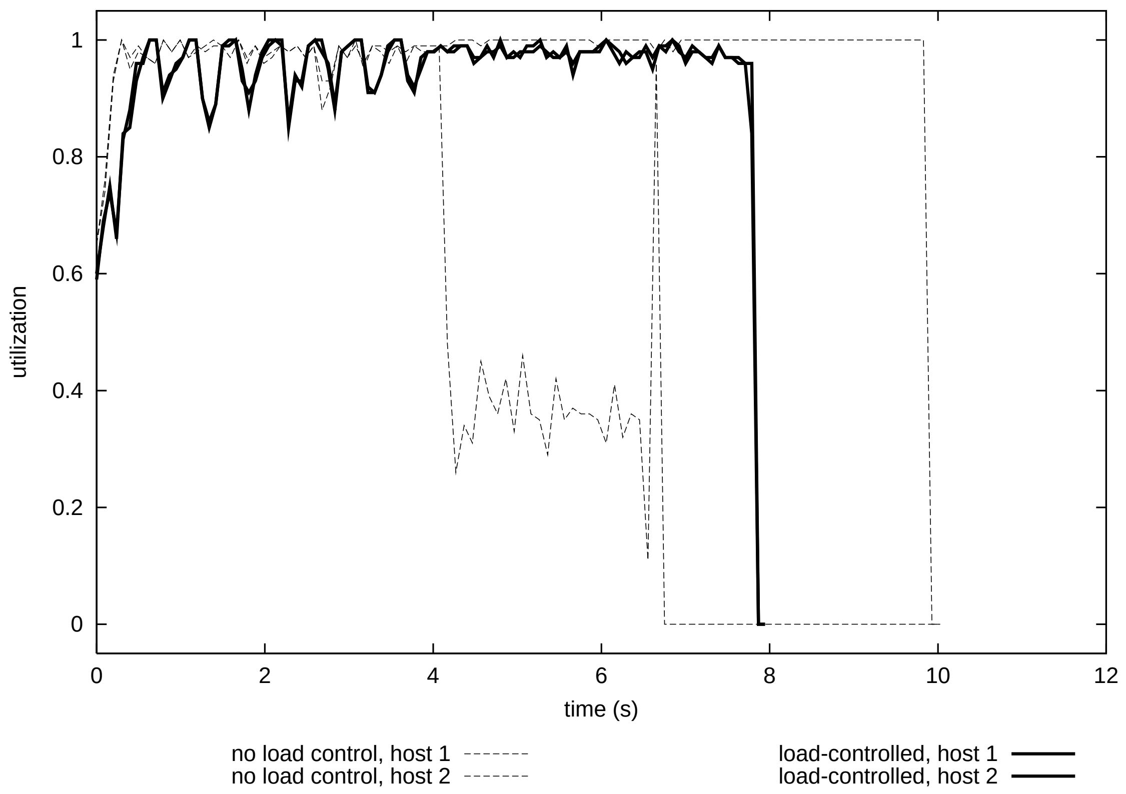

Figure 1.11: Effect of skew: The graph shows utilization of host CPU. Only the second half of the input data is skewed; the first half is uniform. The balanced version terminates before the base case. In load-managed case, CPU utilization of both hosts are nearly identical.

algorithms, and also exposes sufficient structure to allow efficient use of system resources. The theoretical framework based on the EM model will guide our algorithm development. However, practical systems are too complex to analyze accurately, so our primary evaluation methodology will be simulation-based. Simulation also allows the exploration of a larger parameter space than otherwise possible.

Active storage can benefit a class of data intensive applications. We will identify it, and make it possible to exploit these benefits in a practical development environment.

Bibliography

A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. K. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O intensive parallel applications. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 15–27, Philadelphia, May 1996. ACM Press.

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.

R. C. Agarwal. A super scalar sort algorithm for RISC processors. In Jagadish and Mumick [1996], pages 240–246. ISBN 0-89791-794-4. URL `http://www.acm.org/pubs/articles/proceedings/mod/233269/p240-agarwal/p2%40-agarwal.pdf;http://www.acm.org/pubs/citations/proceedings/mod/233269/p240-agarwal/`.

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

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. 2000a. ISBN 0-7695-0601-1.

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 2000b. USENIX Ass.

K. S. Amiri. *Scalable and manageable storage systems*. PhD thesis, CMU, 2000.

Anonymous. A Measure of Transaction Processing Power. *Datamation*, 31(7): 112–118, 1985. Also in *Readings in Database Systems*, M.H. Stonebraker ed., Morgan Kaufmann, San Mateo, 1989.

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. ISBN 0-89791-911-4. URL `http://www.acm.org/pubs/articles/proceedings/mod/`

253260/p243-arpaci-dus%seau/p243-arpaci-dusseau.pdf;http://www.acm.org/pubs/citations/proceedings/mod/253260/p243-arpaci-dusseau%/.

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. 1998a. ACM SIGMETRICS. U.C. Berkeley.

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. URL http://vibes.cs.uiuc.edu/IOPADS/Accepted/Remzi.ps.

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, 1998b. IEEE Computer Society TCCA.

A. Bäumker, W. Dittrich, and F. M. auf der Heide. Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. *Theoretical Computer Science*, 203:175–203, 1998.

R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, Mississippi State, MS, October 1994. IEEE Computer Society Press. URL ftp://hpsl.cs.umd.edu/pub/papers/splc94.ps.Z.

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. URL citeseer.nj.nec.com/beynon01optimizing.html.

G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989. URL citeseer.nj.nec.com/46523.html.

G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, Hilton Head, South Carolina, July 21–24, 1991. SIGACT/SIGARCH.

G. E. Blelloch, G. L. Miller, and D. Talmor. Developing a practical projection-based parallel delaunay algorithm. In *Symposium on Computational Geometry*, pages 186–195, 1996. URL `citeseer.nj.nec.com/blelloch96developing.html`.

J. Chase, A. Gallatin, and K. Yocum. End-system optimizations for high-speed tcp. In *(submitted)*, June 2000.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. M.I.T. Press, Cambridge, Massachusetts, U.S.A., 1990. ISBN 0-262-03141-8.

D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992. ISSN 0001-0782. URL `http://www.acm.org/pubs/toc/Abstracts/0001-0782/129894.html`.

D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsaio, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990. URL `citeseer.nj.nec.com/dewitt90gamma.html`.

G. A. Gibson, D. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Measurement and Modeling of Computer Systems*, pages 272–284, 1997. URL `citeseer.nj.nec.com/gibson97file.html`.

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.

G. A. Gibson, D. P. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. G. C. Lee, B. Ozceri, E. Riedel, and D. Rochberg. A case for network-attached secure disks. Technical Report CMU–CS-96-142, Carnegie-Mellon University, June 1996.

J. Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/, 2001. URL `http://research.microsoft.com/barc/SortBenchmark/`.

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.

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. ISSN 1389-1286. URL `http://www.elsevier.nl/gej-ng/10/15/22/52/30/33/abstract.html;http://www.elsevier.nl/gej-ng/10/15/22/52/30/33/article.pdf`.

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*. Carnegie Mellon University, January 2002.

W. Hightower, J. Prins, and J. Reif. Implementations of Randomized Sorting on Large Parallel Machines. In *Symposium on Parallel Algorithms and Architectures*, June 1992.

D. Hutchinson. *Parallel Algorithms in External Memory*. PhD thesis, Carleton University, May 1999. URL `http://www.cs.duke.edu/~hutchins/dhthesis.ps.gz`.

D. Hutchinson, J. S. Vitter, J. S. Chase, and R. Wickremesinghe. Algorithms for active disks. Work in Progress, 2001.

H. V. Jagadish and I. S. Mumick, editors. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4–6, 1996*, New York, NY 10036, USA, 1996. ACM Press. ISBN 0-89791-794-4.

J. JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley, New York, 1992. ISBN 0-201-54856-9.

K. Keeton. *Computer architecture support for database applications*. PhD thesis, Univ. of California at Berkeley, July 1999. URL `citeseer.nj.nec.com/keeton99computer.html`.

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. ISSN 0163-5808.

D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.

G. Ma, A. Khaleel, and A. L. N. Reddy. Performance evaluation of storage systems based on network-attached disks. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):956–??, Sept. 2000. ISSN 1045-9219. URL `http://www.computer.org/tpds/td2000/l0956abs.htm;http://dlib.computer.org/td/books/td2000/pdf/l0956.pdf`. also appears as [Ma and Reddy 1998].

G. Ma and A. Reddy. An evaluation of storage systems based on network-attached disks. In *Proceedings of the 1998 International Conference on Parallel Processing (ICPP '98)*, pages 278–286, Washington - Brussels - Tokyo, Aug. 1998. IEEE USA. ISBN 0-8186-8650-2. also appears as [Ma et al. 2000].

X. Ma and A. Reddy. MVSS: Multi-view storage system. In *21st International Conference on Distributed Computing Systems (ICDCS' 01)*, pages 31–38, Washington - Brussels - Tokyo, Apr. 2001. IEEE. ISBN 0-7695-1077-9.

NASA. Earth observing system data and information system (EOSDIS). http://eosdismain.gsfc.nasa.gov/eosinfo/EOSDIS_Site/index.html, 2002.

C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: a RISC machine sort. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data / SIGMOD '94, Minneapolis, Minnesota, May 24–27, 1994*, volume 23(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 233–242, New York, NY 10036, USA, 1994. ACM Press. ISBN 0-89791-639-5. URL `http://www.acm.org/pubs/articles/proceedings/mod/191839/p233-nyberg/p23%3-nyberg.pdf;http://www.acm.org/pubs/citations/proceedings/mod/191839/p233-nyberg/`.

C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. AlphaSort: A cache-sensitive parallel external sort. *VLDB Journal: Very Large Data Bases*, 4 (4):603–627, Oct. 1995. ISSN 1066-8888 (print), 0949-877X (electronic). URL `http://ftp.informatik.rwth-aachen.de/dblp/db/indices/a-tree/n/Nyberg:Ch%ris.html;http://ftp.informatik.rwth-aachen.de/dblp/db/indices/a-tree/b/Barclay:Tom.ht%ml;http://ftp.informatik.rwth-aachen.de/dblp/db/indices/a-tree/c/Cvetanovic:Zar%ka.html;http://ftp.informatik.rwth-aachen.de/dblp/db/indices/a-tree/g/Gray:Jim.html;http://ftp.informatik.rwth-aachen.de/dblp/db/indices/a-tree/l/Lomet:David_B=%.html`. Electronic edition.

J. Patel, J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellmann, J. Kupsch, S. Guo, J. Larson, D. DeWitt, and J. Naughton. Building a scaleable geo-spatial DBMS: Technology, implementation, and evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 336–347, New York, May 13–15 1997. ACM Press.

J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In Jagadish and Mumick [1996], pages 259–270. ISBN 0-89791-794-4. URL `http://www.acm.org/pubs/articles/proceedings/mod/233269/p259-patel/p259%-patel.pdf;http://www.acm.org/pubs/citations/proceedings/mod/233269/p259-patel/`.

S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithm. *SIAM Journal on Computing*, 18(3):594–607, June 1989.

J. H. Reif and L. G. Valiant. A Logarithmic time Sort for Linear Size Networks. *Journal of the ACM*, 34(1):60–76, Jan. 1987.

E. Riedel. *Active Disks – Remote Execution for Network-Attached Storage*. PhD thesis, CMU, November 1999. URL `http://www.pdl.cs.cmu.edu/ftp/Active/riedel_thesis.pdf`.

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. ISBN 1-58113-218-2.

E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage. Technical Report CMU-CS-97-198, CMU, December 1997. URL `citeseer.nj.nec.com/riedel97active.html`.

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 [**?**].

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.

SDSS. Sloan digital sky survey. http://www.sdss.org/, 2002.

M. Stonebraker. The case for shared nothing. In *IEEE CS Technical Com. on Database Engineering Bulletin*, volume 9, Mar. 1986. Also published in/as: IEEE Database Engineering, Vol.5, 1986, pp.4–9.

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. ISBN ???? URL `http://www.acm.org/pubs/articles/proceedings/mod/342009/p451-szalay/p45%1-szalay.pdf;http://www.acm.org/pubs/citations/proceedings/mod/342009/p451-szalay/`.

L. Toma. External memory graph algorithms and applications. Second Year Project Report, 2001.

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

M. Uysal, A. Acharya, and J. Saltz. An evaluation of architectural alternatives for rapidly growing datasets, active disks, clusters, SMPs. Technical Report CS-TR-3957, University of Maryland, College Park, Nov. 1998.

M. Uysal, A. Acharya, and J. Saltz. Evaluation of active disks for decision support databases. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 337–348, Toulouse, France, January 8–12, 2000. IEEE Computer Society TCCA.

L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

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

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.

X. Zhang, L. Rivera, and A. Chien. HPVM MinuteSort. White Paper, c1999. URL `http://www.microsoft.com/windows2000/hpc/hvpmminsort.asp`. see Rivera's Thesis [Rivera-Alvarez 2000].

A. Zomaya, editor. *Parallel and Distributed Computing.* McGraw-Hill, 1996.