Cluster I/O with River: Making the Fast Case Common

Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft,
David E. Culler, Joseph M. Hellerstein, David Patterson, Kathy Yelick

Computer Science Division
University of California, Berkeley

Abstract

We introduce *River*, a data-flow programming environment and I/O substrate for clusters of computers. River is designed to provide maximum performance in the common case — even in the face of non-uniformities in hardware, software, and workload. River is based on two simple design features: a high-performance *distributed queue*, and a storage redundancy mechanism called *graduated declustering*. We have implemented a number of data-intensive applications on River, which validate our design with near-ideal performance in a variety of non-uniform performance scenarios.

1 Introduction

Scalable I/O systems form the basis for much of the high-performance computing market. In recent years, manufacturers have found that growth in customer appetite for I/O capacity is outstripping Moore's law [40]. Cluster systems are a key component in the design of today's most scalable data-intensive architectures [1, 2, 17, 50].

A frustrating aspect of cluster I/O systems is that their common-case performance is often a great deal worse than their reported peak performance. This discrepancy arises from various forms of *performance heterogeneity* across clustered components. The simplest heterogeneity is in hardware: a cluster may be composed of machines of differing speeds or capacities. In principle, this problem can be solved by fiat, as is done in packaged clusters such as IBM's SP-2. More nefarious are heterogeneities in software performance, which can arise dynamically from a multitude of sources: unexpected operating system activity, uneven load placement, or a heterogeneous mixture of operations across machines. Software heterogeneity is particularly hard to control, since it changes quickly over time. Surprisingly, hardware heterogeneity is non-trivial to control as well. For example, the inner cylinders of a disk have much less bandwidth than the outer [36], and two apparently identical disks can have different bandwidths depending on the locations of unused "bad" disk blocks.

Presented at IOPADS '99: Input/Output for Parallel and Distributed Systems, May, 1999, Atlanta, Georgia

Rather than attempting to prevent performance heterogeneity, we instead have designed an I/O system that takes it into account as an inherent design consideration. In this paper we describe *River*, a data-flow programming environment and I/O substrate for clusters. The goal of River is to provide common-case maximal performance to I/O-intensive applications. This is achieved using two basic system mechanisms: a *distributed queue* (DQ) balances work across consumers of the system, and a data layout and access mechanism called *graduated declustering* (GD) dynamically adjusts the load generated by producers.

At the center of the River design is a high-performance DQ implementation. River uses DQs to let data flow between operators at autonomously adaptive rates: at any given time, each producer places data into the DQ as fast as it can, and each consumer takes data from the DQ as fast as it can. By interposing DQs between operators in a data flow, load is naturally balanced across consumers running at different rates. An advantage of this simplicity is the lack of global coordination required: consumers can change their rate autonomously over time, without communicating with other clients. The result is full-bandwidth, balanced consumption: all available bandwidth is naturally utilized at all times, and all consumers of a given set of data complete near-simultaneously.

The second important aspect of River is a flexible, redundant disk layout and access mechanism called graduated declustering. A generalization of a mechanism proposed for early parallel database systems [26], GD allows the task of data *production* to be shared among multiple producers in a flexible fashion. GD mirrors large sequential collections on the disks of different producers. During data flow, a producer multiplexes its I/O bandwidth across all the data sets it is currently handling, to ensure that it produces its share of the global bandwidth available for each collection. The result is full-bandwidth, balanced production: all available bandwidth is utilized at all times, and all producers of a given data set complete near-simultaneously.

In introducing River, we also describe its programming model, and a graphical interface for composing River programs. These are based on traditional data-flow diagrams composed of operators, similar to those used in database query plans [24] and scientific data-flow systems [31, 46]. This intuitive interface allows programmers to focus on application-specific logic, while River transparently handles the issues of high-performance I/O and parallelism within the application.

We demonstrate the River interface with a number of data-intensive applications, and use them to validate the perfor-

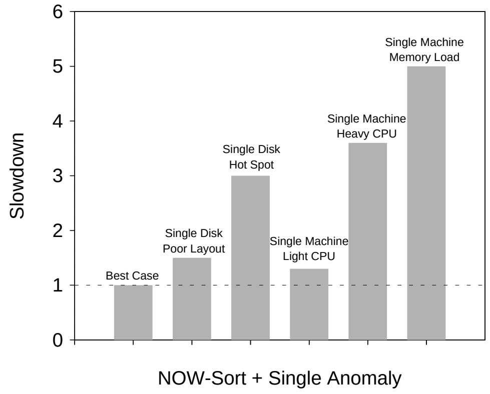

Figure 1: **NOW-Sort Under Perturbance.** *The graph depicts the best-case performance of NOW-Sort, versus the performance under slight disk, CPU, and memory perturbations. All performance results are relative to the 8-node NOW-Sort, which delivers data at a near-peak disk rate of 40 MB/s throughout the run.*

mance of the system. In all cases River provides near-ideal performance in the face of severe performance perturbations.

1.1 Motivation: A Case Study

To better motivate the problem of performance heterogeneity, we perform a simple experiment with NOW-Sort [2], a high-performance parallel external sort for clusters. In the experiment, the sort runs on 8 machines, and in each run, we perform a slight perturbation of the sort on just one of those machines. The results from these perturbation experiments are shown in Figure 1.

As we can see from the graph, each of the perturbations on just a single machine has a serious global performance effect. If a single file on a single machine has poor layout (inner tracks versus outer), overall performance drops by 50 percent. When a single disk is a "hot spot", and has a competing data stream, performance drops by a factor of 3. CPU loads on any of the machines decrease performance proportional to the amount of CPU they steal. Finally, when the memory load pushes a machine to page, a factor of five in performance is lost.

While it may be possible to build a system that avoids all of these situations by balancing load across the system perfectly at all times and meticulously managing all resources of the system, we believe it is difficult. As system size and complexity increase, carefully managing such a system becomes near-impossible. Therefore, we are approaching the problem in a different manner, by assuming the presence of such "performance faults", and providing a substrate that can operate well in spite of them.

1.2 Outline

The rest of the paper is structured as follows. In Section 2, we describe the design of the system and its current implementation, *Euphrates*. In Section 3, we validate the performance properties of our dynamic I/O infrastructure, with measurements of both distributed queues and graduated declustering. We present initial application experience in Section 4. Related work is found in Section 5. In Section 6, we present our plans for future work, and in Section 7, we conclude.

2 The River System

This section describes the design of the River environment, as well as the current implementation, *Euphrates*. First, we briefly describe our hardware and software environment. Then, we present the River data model: how data is stored and accessed on disk. We continue by explaining the components of the River programming model, including details of how a typical River program is constructed. We conclude with a discussion.

2.1 Hardware and Software Environment

The River prototype, Euphrates, currently runs on a cluster of Ultra1 workstations connected together by the Myrinet local-area network [9]. Each workstation has a 167 MHz UltraSPARC I processor, two Seagate Hawk 5400 RPM disks (one used for the OS and swap space in the common case), and 128 MB of memory. Solaris 2.6 is the operating system on each machine, a modern multi-threaded UNIX [29].

All communication is performed with Active Messages (AM), a second generation communication layer designed for distributed computing [34]. AM exposes most of the raw performance of Myrinet while providing support for threads, blocking on communication events, and multiple independent endpoints. Other fast message layers [39, 48, 49] do not support blocking on communication events and thus require polling the network interface to receive messages; boundless polling consumes CPU cycles and is not appropriate for building an I/O infrastructure such as River.

2.2 The Data Model

2.2.1 Single Disk Collections

On a single disk, data is represented as a group of on-disk records known as a *collection*. Each record has a set of named fields, which can be of various types. This catalog of information is kept as meta-data by the system.

Data can be accessed on disk as an *unordered collection*. Unordered collections provide no ordering constraints between records of the collection. Thus, an application reading such a collection may receive records in an arbitrary order, subject to optimizations by the system.

When ordering is desired by the application, data can be accessed as a *stream*. A stream is an ordered set of records. Thus, when an application writes a collection to disk as a stream, the write order is preserved; applications accessing the collection directly will receive the records in that order.

The Euphrates implementation uses the underlying Solaris 2.6 UNIX file system (UFS) to implement record collections. To read from disk, we use either `read()` with `directio()` enabled (an unbuffered read from disk), or the `mmap()` interface, both of which deliver data at the raw disk rate for sequential read access. Simple use of the `read()` interface without `directio()` leads to double-buffering inside of the file system, which is undesirable for most of our applications. Writes to disk use the `write()` system call, with or without `directio()` enabled.

When implemented on top of UFS, layout information is not available, and therefore the optimizations that would be possible with unordered collections are not currently implemented in the disk manager. The next implementation of River will include a disk manager on top of raw disk, in order to exploit the range of scheduling optimizations that would thus be enabled.

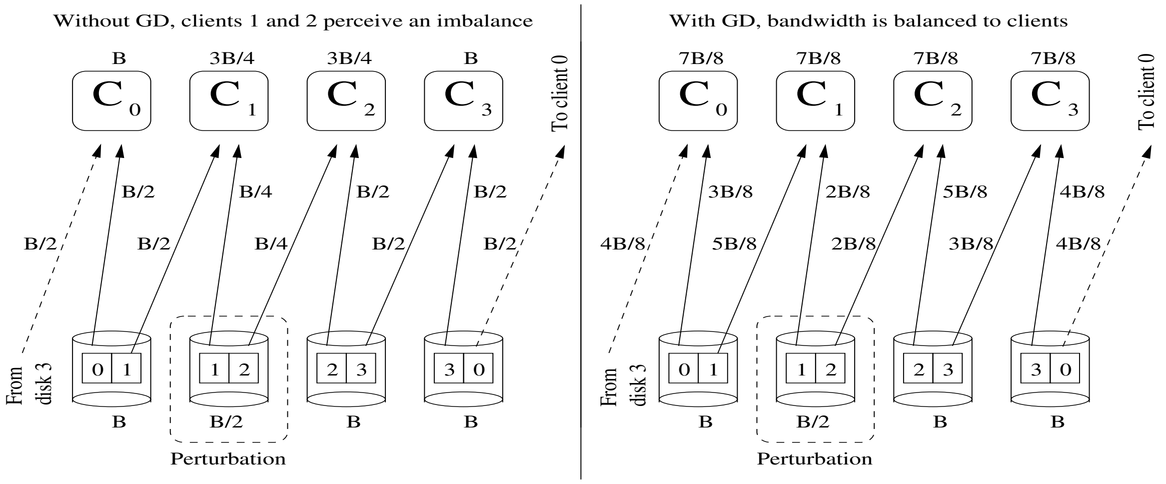

Figure 2: **Graduated Declustering.** *These two diagrams depict two scenarios, without and with graduated declustering under a perturbation. Unperturbed disks normally deliver B MB/s of bandwidth, and the one perturbed disk delivers half of that, B/2. On the left, the disk serving partitions 1 and 2 to clients is perturbed, and thus only half of its bandwidth is available to the application. Left unchecked, the result is that clients 1 and 2 do not receive as much bandwidth as clients 0 and 3. On the right, the bandwidths from each disk have been adjusted to compensate for the perturbation, as is the case with graduated desclustering. With the adjustments, each client receives an equal share of the available bandwidth.*

2.2.2 Parallel Collections

Most of the applications in our system wish to access data spread across multiple disks. To facilitate this, we provide the abstraction of a *parallel collection*. This allows the grouping of a set of single-disk collections into a single logical entity. The parallel collection facility only tracks parallel meta-data, such as the names and physical locations of each single-disk collection that form the parallel collection, and any desired ordering between the single-disk collections.

In Euphrates, the parallel collection meta-data is stored in NFS. Because NFS provides no consistency guarantees under concurrent access, all parallel meta-data operations are serialized through a single process of the application. These operations are rare (that is, they only occur when a file is being opened or created), and therefore are not a performance bottleneck.

2.2.3 Redundancy

In large scale clusters, the presence of a data availability strategy is important. Without one, data will frequently be unavailable due to disk and machine failures. In River, applications that wish to have increased data reliability and availability can choose to *mirror* each single-disk collection across disks housed in different machines within the cluster.

We are interested in exploiting the redundant data contained in mirrors to improve the consistency of application performance. We do so by building on earlier work in [26], in which the authors introduced *chained declustering*. The key insight behind chained declustering is that, after the failure of one disk in a mirrored system, a read-only load can be balanced evenly across the remaining, working disks. This balance is achieved through a carefully-calculated distribution of read requests to the mirror segments on the remaining disks.

We generalize this technique to what we call *graduated declustering* in order to solve the performance consistency problem. In the common case, all disks storing a mirrored collection are functional, but each may offer a different bandwidth over time (for reasons enumerated earlier) to any individual reader. Under traditional approaches to mirroring, these variations are unavoidable because a reader will choose one mirrored segment copy from which to read the entire segment. Such variations can lead to a global slowdown in parallel programs, as slow clients complete later than fast ones.

To remedy this, we approach the problem somewhat differently. Instead of picking a single disk to read a partition from, a client will fetch data from all available data mirrors, as illustrated in Figure 2. Thus, in the case where data is replicated on two disks, disk 0 and disk 1, the client will alternatively send a request for block 0 to disk 0, then block 1 to disk 1; as each disk responds, another request will be sent to it, for the next desired block.

However, this alone does not solve the problem. Graduated Declustering must provide each client that is reading a set of collections an equal portion of the bandwidth available to the application as a whole. Clients that receive less than the expected bandwidth from one of the two disk mirrors must receive more bandwidth from the other mirror as compensation. Thus, the implementation of graduated declustering must somehow observe these bandwidth differences across clients and adjust its bandwidth allocation appropriately.

The Euphrates implementation of GD uses a simple algorithm to balance load amongst data sources. Each disk manages two different segments of a parallel collection, and continually receives feedback from two consumers as to the total bandwidth that the consumers are receiving. When a performance inequity between two clients is detected, the disk manager biases requests towards the lagging client, and thus attempts to balance the rates at which the readers progress. An example of the result of such a balancing is shown in the right-side of Figure 2. There, both disks 0 and 2 compensate for a perturbation to disk 1 by allocating 5/8 of their bandwidth to clients 1 and 2. The resulting bandwidths to each client are properly balanced.

```
// module loop: get records + process
while ((msg = Get() != NULL) {
    // operate on given message
    rc = Operate(msg);

    // conditionally pass message downstream
    if (rc) Put(msg);
}
// indicate completion
return NULL;
```

Figure 3: **Module API.** *This is a simple River module. The module Get()s messages from upstream, performs some operation on them by calling a user-defined Operate(), and then (conditionally) Put()s messages downstream.*

2.3 The Programming Model

River provides a generic data flow environment for applications, similar to parallel database environments such as Volcano [24]. Applications are constructed in a component-like fashion into a set of one or more *modules*. Each module has a logical thread of control associated with it, and must have at least one input or output channel, often having one or more of each. A simple example is a filter module, which gets a record from a single input channel, applies a function to the record, and if the function returns *true*, puts the data on a single output channel.

Modules are connected both within a machine and across machine boundaries with queues. A queue connects one or more producers to one or more consumers and provides rate-matching between modules. By dynamically sending more data to faster consumers, queues are essential for adjusting the work distribution of the system.

To begin execution of an application, a master program constructs a *flow*. A flow connects the desired set of modules, from source(s) to sink(s). Any time a single module is connected to another, a queue must be placed between them. When the flow is instantiated by the master program, the computation begins, and continues until the data has been processed. Upon termination, control is returned to the master program.

2.3.1 River Modules

A module is the basic unit of programming in River. Modules operate on records, calling `Get()` to obtain records from one or more input channels, and then calling `Put()` to place them onto one or more output channels. For convenience, we refer to a set of records that is moving through the system as a *message*. Logically, each module is provided a thread of control. Thus, a one-input, one-output module performs a simple loop: `Get()` to obtain records from an upstream channel, operate on those records, and then `Put()` to pass the records downstream, as illustrated in Figure 3.

More complex modules may have more than one input or output; in that case, they must specify the input/output number as an argument to `Get()` or `Put()`. Non-blocking versions of these interfaces are also available, as is the ability to perform a `Select()`: this operation waits until one of a specified set of channels is ready, and then returns control to the user.

In Euphrates, modules are written as C++ classes. In the current implementation, each module is given its own thread of control, which has both its benefits and drawbacks. The main advantage of this approach is that applications naturally overlap computation with data movement; thus, the user is freed from the burden of carefully managing I/O. However, thread switches can be costly. To amortize this cost, modules should pass data (a set of records) amongst themselves in relatively large chunks. In our experience, this has not complicated modules in any noticeable fashion; thus, we felt that the inclusion of complex buffer management was not worth the implementation effort.

2.3.2 Queues

Queues connect multiple producers to multiple consumers, both in the local (same machine) and distributed (different machines) cases. During flow construction, queues are placed between modules and messages are transmitted from producers to consumers. Modules that are placed on either side of local or distributed queues are oblivious to the type of queue with which they interact.

Messages in River may move arbitrarily through the system, depending on run-time performance characteristics and the constraints of the flow. Dynamic load balancing is achieved by routing messages to faster consumers through queues that have more than one consumer.

To improve performance, ordering may be relaxed across queues. In a multi-producer queue, a consumer may receive an arbitrary interleaving of messages from the producers. The only ordering guarantee provided in a queue is point-to-point; if a producer places message A into queue Q before message B, and if the same consumer receives both messages, it receives A before it receives B. This ordering is necessary, for example, to retain the ordering of a disk-resident stream. By attaching a single consumer to the single producer of a stream, the ordered property of the stream can be properly maintained.

In our implementation, local queues are data structures shared between threads with the appropriate locking and signalling protocol. The Euphrates implementation of the DQ is more interesting, and takes on two different flavors. In the general case, we use a lightweight, randomized, credit-based scheme to balance load across consumers. In this *push-based* algorithm, each producer tracks the number of outstanding messages sent to any consumer, and sends new messages randomly only to consumers that have few messages currently outstanding (less than a threshold value). This has the desired behavior of automatically sending more records to nodes that are consuming at higher rates, and can be implemented efficiently: the randomized algorithm adds near-zero CPU overhead on top of the normal message transfer costs.

In some cases, we have found that load balancing must be provided for larger-than-record size units. For example, after a sort module has sorted its input data, it may wish to pass the entire sorted run to a disk write module, with the order preserved (we will see this exact example in Section 4). To provide this functionality, the DQ implementation can be handed an arbitrarily large set of records; it then uses a *pull-based* algorithm, with consumers querying producers for data, to balance load. The randomized, push-based algorithm does not work well in this case, because a single bad decision is quite costly. The guarantee provided by this version of the DQ is that a single consumer will receive the entire set of records, in the order that it is given to the DQ. Thus, load balancing occurs at the granularity of the large (potentially many MB) unit handed to the DQ.

```
// simple copy program
Flow f;
Module *m1, *m2;
// instantiate module instances
m1 = f.Place("UFSRead", "file=in.1");
m2 = f.Place("UFSWrite", "file=out.1");
// attach read module to write
f.Attach(m1, m2);
// execute flow
f.Go();
```

Figure 4: **Flow API.** *A simple reader to writer flow is shown. The UFSRead module reads in collection "in.1"; its output goes to the input of the UFSWrite module, which writes it to disk under the name "out.1".*

2.3.3 Flow Construction

To execute a program in the River environment, one or more modules must be connected together to form a *flow*. A flow is a graph from data source(s) to sink(s), with as many intermediate stages as dictated by the given program.

There are three phases involved in instantiating a flow: construction, operation, and tear-down. During construction, a *master program* specifies the global graph, describing where and how data will flow, including which modules to use and their specific interconnection. When the construction phase is complete, the master program instantiates the flow. In the operation phase, threads are created across machines as necessary, and control is passed to each of the modules. The flow of data begins at the data sources, and flows through the system as specified by the graph, until completion.

Flow construction can be performed programmatically (a flow API is provided) or graphically. The flow construction API is quite simple: to add a node to a graph, the `Place()` routine is called, specifying the name of the module and any arguments it might take. For example, to read an on-disk collection, the programmer might specify the UFSRead module, with an argument of the filename, as shown in Figure 4.

`Place()` returns a reference to the module, which is then used to attach modules together via a simple `Attach()` interface, the interface used to specify graph edges. In the figure, a simple copy flow is formed: both a Read and Write module are placed in the flow, and then attached together. Attaching two modules together places a queue between them. Modules can have more than one input or output; in this case, the user must specify extra arguments to the `Attach()` routine, to specify which input to connect to which output.

Finally, to instantiate the flow, a `Go()` interface is provided, which starts the threads, performs the necessary attachments, and waits for their completion. An asynchronous version of `Go()` is also available.

The flow description up to this point has been restricted to single-machine flow specification, for the sake of simplicity. To construct *parallel flows* across multiple machines, the programmer need only specify which nodes to place the various modules upon; local and distributed queues are inserted where appropriate, and when the program is run, it is spawned across the nodes of the system using a simple remote execution module, internal to the system. The user can add extra arguments to the `Attach()` routine to specify details about remote connections between producers and consumers: whether to use a single m-to-n distributed queue, n 1-to-1 distinct queues, or an $m \times n$ fully-connected graph.

In the Euphrates implementation, numerous languages can be used to program flows. A C++ interface is available, but we have found it overly cumbersome to re-compile codes for each simple change to a flow. Therefore, we provide both Tcl and Perl interfaces, allowing for the rapid assembly of flows in a scripting language.

Finally, we have built a graphical user interface (GUI) for specifying data flow graphs, similar in spirit to Tioga [46]. The GUI allows programmers to select modules from a module library and draw the data flow graph as desired. The user can then execute the program, or generate the flow construction code for later re-use. The GUI also allows variables to be added to the program, thus enabling the user to easily construct generic programs. In the example of the simple copy, the user might choose to have the input and output collection names as variables, and then generate a general-purpose copy program. In general, we have found this simpler to use than the programmatic interface, and less bug-prone.

2.4 Discussion

We conclude the section with a discussion of the system, and how we expect it will be used. The typical programmer writing a new program will most likely spend their time programming the individual modules; this is where the bulk of application code should live. They will then use their modules and perhaps some of the modules that come with the system in order to construct a flow. We imagine that a user community interested in similar problem areas would have one or more libraries of standard modules that all would share, and that have been tuned for high performance.

Achieving parallelism is then rather straight-forward; the user must construct the flow either in a script or with the GUI tool, and specify nodes on which to run; the system will spawn modules across multiple nodes quite easily, and generate the desired connections (queues, local or distributed) between modules.

However, the focus of River is not simply enabling the construction of high-performance, parallel, I/O-intensive applications; we seek to provide the necessary framework for building performance-robust programs. The system provides one part of the solution transparently to application writers, with the graduated declustering algorithm. By enabling mirroring, applications automatically gain robustness to read perturbations.

However, the other component of River that provides performance robustness is distributed queues, which must be inserted by the application writer where appropriate. In most cases, where to place DQs depends on program semantics, and therefore it is difficult to automate such a decision. In general, DQs can be quite easily inserted wherever there is embarassing parallelism; in those cases, producers place work into the queue, and consumers take work from the queue, all at their individual rates. The addition of DQs in other situations is a bit more difficult, and requires a solid understanding of the application.

Overall, the construction of performance-robust applications requires the application writer to construct and optimize sequential modules, and describe a flow to connect them, inserting distributed queues where possible. By spending some programmer effort on DQ placement, the user will gain in return a scalable application that runs well in the face of variable-rate producers and consumers; indeed, a well-designed River application will run with high performance across a set of machines with highly varying performance characteristics.

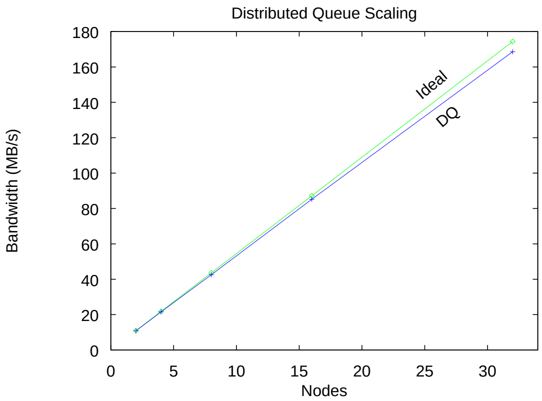

Figure 5: **Distributed Queue Scaling.** *In this experiment, the scalability of the DQ is under scrutiny. During the run, from 1 to 32 producers read data blocks from disk and put them into the distributed queue, and 1 to 32 sources pull data from the DQ. The ideal line shows the aggregate bandwidth that is available from disk.*

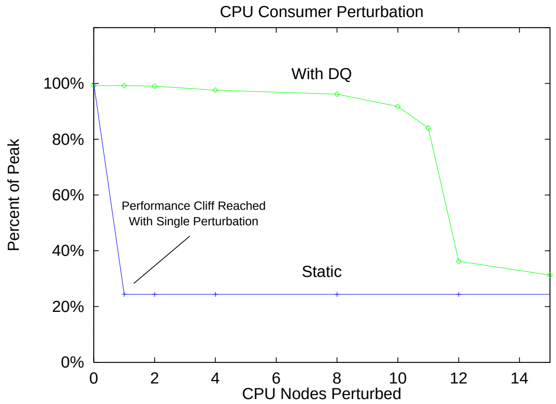

Figure 6: **DQ Read Under Perturbation.** *This figure shows the percent of peak performance achieved as consumer perturbations are added into the system. Without a DQ to balance load across unperturbed consumers, performance drops as soon as a single consumer is slowed. With a DQ, performance is unaffected until a large number of nodes are perturbed. A CPU perturbation steals 75% of the processor; the test consists of 15 producers and 15 separate consumers.*

3 Experimental Validation

In this section, we perform experiments to validate the expected performance properties of the system. First, we explore the absolute performance and adaptability of the distributed queue. The performance of the queue is crucial to the system, as this is the primary mechanism for providing load balancing within a flow. We will see that the distributed queue is effective in balancing load across consumers, and moving more data to faster consumers.

We then perform experiments on graduated declustering, our performance enhancement for mirrored collections. Balancing work across consumers (via distributed queues) alone does not solve the problem of achieving consistent performance; when a single producer slows, performance of the system drops proportionally. In this case, it is important for the system to avoid the producer "hot-spot". This is precisely what GD transparently provides, using a simple distributed algorithm to adapt to run-time perturbations of data sources.

3.1 Distributed Queue Performance

3.1.1 Absolute Performance

First, we explore the scaling behavior of the distributed queue. In the first experiment, we have the following set-up: data is read from n disks, put into a distributed queue, and consumed by n CPU sinks. We scale n from 1 to 32. The results of this scaling experiment are shown in Figure 5.

As the graph reveals, the scaling properties are near ideal. Each disk is capable of delivering 5.45 MB/s. From 32 disks, we thus would expect a peak read bandwidth of 174.4 MB/s. With the data moving through the DQ, we achieve 168.6 MB/s, or about 97 percent of peak. If the distributed queue is found to have scaling problems at a given cluster size, we could design a less aggressive algorithm, where each producer only sends data to some subset of the consumers; we have not yet seen the need for this. The performance when writing to disks through a DQ (not shown) scales equally well.

3.1.2 Performance Under Perturbation

We next examine the results when one or more consumers is arbitrarily slower than the rest. This type of perturbation could arise from dynamic load imbalance or hot spots in the system, or could be due to the presence of CPUs or disks with different performance capabilities.

Figure 6 shows the effect of slowing down 1 to 15 CPU consumers both with and without a DQ, when reading from 1 to 15 disks. Without a DQ, work is pre-allocated across consumers; thus, if a single consumer slows down, the performance is as bad as if all consumers had slowed down (this is labeled "static" for static allocation in the figure).

When a DQ is inserted between the producers (disks) and consumers, more data flows to unperturbed consumers, thus flowing around the hot spots in the system. Because the CPUs are not fully utilized in the unperturbed case, there is no noticeable performance drop-off under perturbation until 8 to 10 consumers are perturbed.

Whereas the previous experiment was a form of a parallel read, the next experiment is a parallel write. In this experiment, we place a DQ between CPU sources (which generate records) and the disks in the system. The results of the write experiment are shown in Figure 7.

Once again, the static allocation behaves quite poorly under slight perturbation. In this case, however, the performance when writing to disks through the DQ degrades immediately under perturbation, gradually falling off; in fact, performance becomes slightly worse than the static application when all 15 of the disks are under perturbation. The cause of the immediate degradation is that the disk bandwidth is fully utilized to begin with, unlike the CPUs in the DQ read experiment above. Thus, when a single disk of the system is perturbed, the total bandwidth available is reduced. The difference is that with the DQ, more data is sent to unperturbed disks, whereas the static application does not adapt.

We have now demonstrated that the distributed queue has the desired properties of balancing load among data *consumers*; however, without mirroring, each *producer* of data has a unique

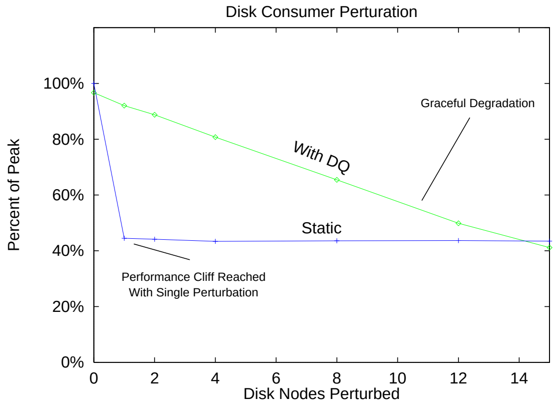

Figure 7: **DQ Write Under Perturbation.** *This figure shows the effect of disk perturbation during writes, and how the DQ dynamically adapts. The system under test consists of 15 disks. Instead of falling off the performance cliff, the DQ routes data to where bandwidth is available, and thus gracefully degrades. In this case, each perturber continually performs sequential, large-block, writes to the local disk, stealing roughly half of the available bandwidth.*

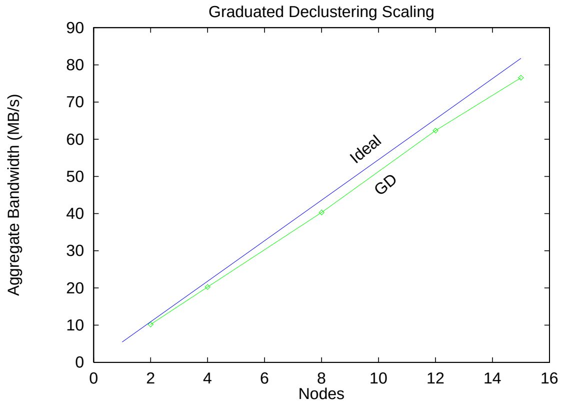

Figure 8: **Graduated Declustering Scaling.** *The graphs shows the performance of GD under scaling. The only performance loss is due to the fact that GD reads actively from both mirrors for a given segment; thus, a seek cost is incurred, and roughly 93% of peak performance is delivered.*

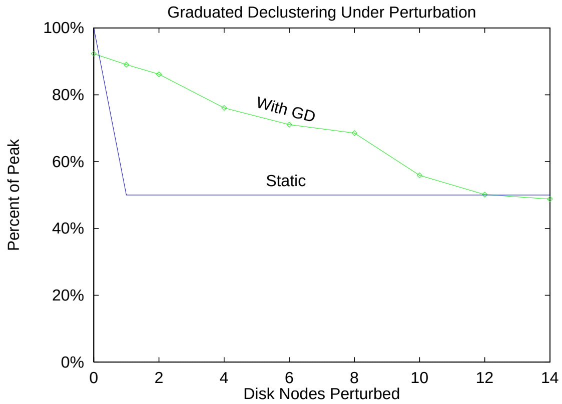

Figure 9: **GD Under Read Perturbation.** *The graphs shows the performance of GD under read perturbation. Performance degrades slowly for the GD case, whereas a typical non-adaptive mirrored system suffers immediate slowdown. Each perturber is a competing read-stream to disk.*

collection of records, and to complete a flow, must deliver that data to the consumers. Thus, when the producers are the bottleneck in the system (as is often the case when streaming through large data sets), slowing a single producer will lead to a large global slowdown, as the program will not complete until the slow producer has finished. This "producer" problem is the exact problem that graduated declustering attempts to solve.

3.2 Graduated Declustering

We now describe our experimental validation of the graduated declustering implementation. We find that both the absolute performance and behavior under perturbations is as expected in our initial implementation.

3.2.1 Absolute Performance

The performance of graduated declustering under reads, with no disk perturbation, is slightly worse than the non-mirrored case. This is a direct result of our design, which always fetches data from both mirrors instead of selecting a single one, in order to be ready to adapt when performance characteristics change. Multiplexing two streams onto a single disk has a slight cost, because a seek must occur between streams. Increasing the disk request size to 512KB or 1MB amortizes most of the cost of the seek, and thus we achieve 93 percent of the peak non-mirrored bandwidth, as seen in Figure 8. Writes, each of which must go to two disks, incur the same problem.

3.2.2 Performance Under Perturbation

The real strengths of GD come forth for read-intensive workloads, such as decision support or data mining. In these cases, applications reading from a non-adaptive mirroring system would slow to the rate of the slow disk of the system. With GD, the system shifts the bandwidth allocation per disk, and thus each consumer of the data receives data at the same rate.

The results of a 28-machine experiment are shown in Figure 9. In this scenario, half of the machines serve as disk nodes, and the other half serve as data consumers. As explained above, the performance of GD mirroring as compared to no mirroring is slightly worse in the unperturbed case. However, a single perturbation slows the application on the non-GD system to the bandwidth of the slow disk, which in this case delivers data at roughly half of peak rate due to a single competing stream. With GD, performance degrades slowly, spreading available bandwidth evenly across consumers. However, when all disks are equally perturbed, the performance of GD once again dips below the non-GD system, again due to the overhead of seeking between multiple streams.

Finally, perturbing a write stream to a collection and its mirror has the expected effect of slowing the write to the speed of the slower disk. In some sense, this represents the fundamental cost of using mirroring; applications that write out scratch data or other data of lesser value should not use mirroring because of this potential performance cost.

4 Applications

We now describe some initial application experience. We begin with an example of how an unmodified, sequential program can use the River infrastructure. Then, we proceed with two parallel applications that we have written, a parallel sort and a parallel hash-join. The section focuses on how application writers add robustness into their applications via distributed queues, and therefore does not show performance with mirroring enabled (as mirroring would be transparent to the user).

4.1 Trace-Driven Simulation

The first application we examine is a trace-driven simulator, a second generation version of a file system simulator used in [35]. This complex, sequential program simulates multiple file system layout policies, buffer management, and includes a complete disk simulator. This application uses River as a fast data source; the simulator did not have to be modified in order to do so.

To access data in the River system, the simulator loads data into the River system via a simple copy-in script, and then accesses it with a copy-out script. The latter constructs a flow from a record collection to standard output, which is piped into the standard input of the simulator. In this case, the main benefit of River is the use of a fast, switch-based network between application and disk. Before using River, the simulator accessed data via an NFS file server over an Ethernet shared network. However, because there is no parallelism in the application, distributed queues can not be used to provide robust performance.

4.2 Parallel External Sort

The next set of experiments involve a more complicated application, external sorting. In this case as with the next, the program has been written entirely within the River environment. Sort is a good benchmark for clustered systems because its performance is largely determined by disk, memory, and interconnect bandwidth. We compare an external sort built in the River framework to an "ideal" statically partitioned sort; by ideal we mean that the parallel sort reads in data from disk at full disk bandwidth, takes zero time to perform the in-memory sort, and then writes back to disk at full bandwidth, all with no overhead to parallelism. For the sake of simplicity, we only consider a single-pass sort (where the records are read into memory, sorted, and written to disk in a single pass); eventually, we plan to extend our work to include two-pass sorting, which places more severe memory management demands on the system.

Figure 10 presents the flow of data in the simple version of external sort in River (the flow is quite similar to NOW-Sort [2]). First, data begins as an unsorted parallel collection on a number of disks. Data is read in on each disk node via the disk read module (D_R), and then passed to a partitioning module (P). The partitioning modules perform a key-range partitioning of the data; thus, each partitioning module reads the top few bits of each record to determine which sorter (S) module should be sent a particular record. When a sorter module has received all of the input, it sorts the data, and begins streaming it to the disk write module (D_W), which proceeds to write the data out to disk as a stream (thus preserving order). Thus, the application proceeds in three phases: read/partition, sort, and write.

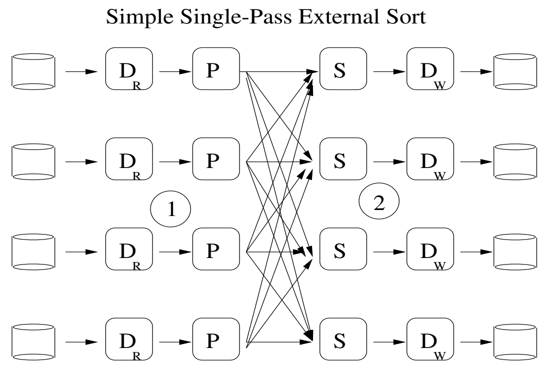

Figure 10: **Parallel External Sort in River.** *This figure depicts the logical data flow in a single-pass external sort. Data proceeds from disk into a set of static partitioners, which split data (based on a key range in each record) across a set of sort modules. When the data has been read in, the sort modules sort entirely in parallel, and hand data to the write modules, which send them to disk as an ordered stream. The two markers, a circled 1 and 2, indicate two points where the flow could be altered to add robustness, as discussed below.*

First, we discuss the scaling behavior of the sort. Figure 11 shows the result of scaling the River sort to 14 machines. The graph compares the River sort to the idealized, statically-partitioned parallel sort. The performance of the sort in the River framework begins at around 90% of peak efficiency, and drops slightly to 86% at 14 nodes. The majority of the inefficiency can be attributed to a poorly tuned in-memory sort, which contributes to 10% of the total elapsed time. Even with the un-tuned, in-memory, sort, we learn from this graph that it is relatively easy to build a high-performance, non-trivial application that does not lose much efficiency inside the framework. Further, the application does not have to write a single line of code to manage I/O. Qualitatively, all the application writer has to write is the partitioning module and the sort module; scaling to a parallel sort is then just a matter of constructing the proper flow.

However, as it stands, the simple River parallel sort is not robust to performance perturbations. With graduated declustering, the sort can tolerate read perturbation. Here, we focus on the read/partition and write phase of the sort, both of which have potential for performance robustness.

First, we examine whether we can perturb the partition modules and still achieve reasonable performance. To add a level of robustness to the partition phase, we insert a distributed queue between the disk read modules and the partitioners (labeled with a circled 1 in Figure 10). Because the sort is partitioning the data, there is no order yet imposed at this stage of the sort, and therefore inserting the distributed queue only changes the performance characteristics of the sort, not the correctness.

Figure 12 shows the result of perturbing the partitioner modules. In this experiment, the disk modules and sort modules are placed on one set of 14 machines, and the partitioners are placed on another set of 14 machines (28 total). When perturbations are applied to the partitioners, other partition modules take over the slack, until the system is overloaded, degrading slowly after 8 of the 14 partition nodes are perturbed.

The other location in the flow that can be modified to avoid

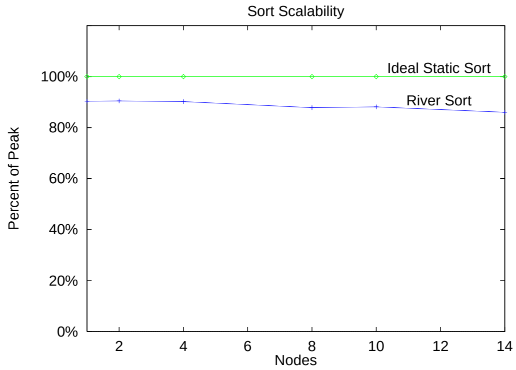

Figure 11: **Parallel External Sort Scaling.** *This figure shows the scaling behavior of the sort built in the River framework, as compared to an idealized statically-partitioned sort. The River sort scales well; its only deficiency is an under-tuned in-memory sort.*

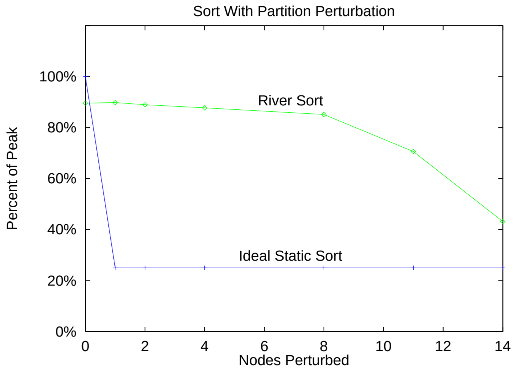

Figure 12: **Perturbing the Sort Partitioner.** *This figure shows the sort when the partition modules are perturbed. The disk and sort modules run on one set of 14 machines, and the partition modules run on another set. The River sort is compared to a "perfect" sort that is statically partitioned. Each perturbation steals 75 percent of the CPU.*

run-time perturbations is between the sort modules and the disk write modules, labeled with a 2 in Figure 10. Our desire is to tolerate one or more disks slowing down during the write phase. However, we can not simply move records arbitrarily among the different disks; we must preserve the set of sorted partitions as generated by each sort module. Thus, instead of balancing load across the disks at the record-level, we can balance load at a *higher level of granularity*, by dynamically deciding where to place each sorted partition.

In order to balance load among n consumers with data from n producers, there must be more than n data items produced. In its original form, the sort allocates a single sort module per producer (and thus per consumer); to remedy this, and allow for load balancing at the disks, we instead allocate $c \cdot n$ sort modules, where c is a small constant. Note that this produces a slightly modified output, in that there are more sorted partitions.

The performance of load balancing sorted-runs under disk perturbation is shown in Figure 13. As expected, by writing runs through the DQ, performance degrades much more gracefully than with the static allocation. However, under full perturbation, the performance is lower than expected; in this case, the overhead of the current implementation results in only 40% of peak performance, roughly 10% lower than expected.

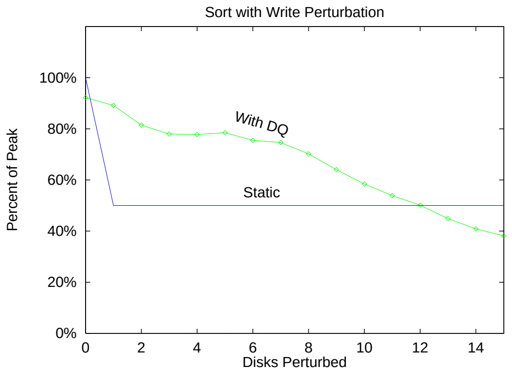

Figure 13: **Perturbing the Sort Writer.** *This figure shows performance when writers are perturbed during the write phase of the sort. The runs are on one set of 14 machines, and are writing to disks on a separate set of 14 machines. In this case, each perturbation is a competing write-stream to disk.*

4.3 Parallel Hash Join

Hash-join is another important database operation, and is used extensively in decision-support benchmarks such as TPC-D. Hash-join takes two collections of records as input, and outputs all pairs that have equal values on the join key. Both one-pass and two-pass variants exist [45, 28]: the one pass algorithm is suitable for use when the smaller collection fits into the aggregate cluster memory.

For simplicity, we discuss the one-pass hash join. Figure 14 shows the flow of data. In the first phase, the smaller collection (or building collection, because a hash table will be built over it) is read from disk, partitioned using a hash function across nodes, and internally hashed inside each join module (labeled J in the diagram) to prepare for the join phase. In the second phase, the second (probing) collection is read from disk, and partitioned across nodes via the same hash function. As records pass into the join module, matching records from the building collection are found, and the output proceeds immediately to disk. Thus, during this phase, both reading of the second collection and writing of the output will operate concurrently.

The addition of distributed queues in the hash join is quite similar to that of the sort. A queue can be placed between data sources and partitioners, allowing faster partitioners to partition more data. After the join is performed, if the output relation is not kept in hash form, another DQ can be inserted, easily balancing load across the disks. If the application wishes to keep the output records in hashed partitions, a situation similar to the balancing of sorted runs in the external sort could be employed.

More interestingly, the hash-join can avoid performance perturbations to the join modules by using replication. If each building collection is replicated to two or more nodes, each record that is partitioned during the probing phase can dynamically choose between sites via a DQ.

Because of some functionality limitations of our current infrastructure, we do not yet have performance numbers for hash-join at scale. However, our initial results (on only 4 machines) are promising.

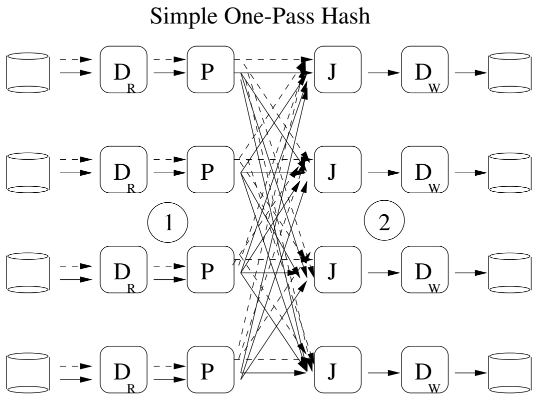

Figure 14: **Parallel External Hash in River.** *This figure depicts the logical data flow in a single-pass hash-join. The solid lines indicate the path of the first relation, from disks, into the hash-partition modules (P), and then into the hash-join modules (J). The dashed lines indicate the path of the second relation, similar to the path of the first relation. When the second relation passes through the hash-join modules, the join is performed, and the output relation is generated.*

5 Related Work

River relates to work from a number of often distinct areas: file systems, programming environments, and database research. In this section, we discuss work from the three areas.

5.1 Parallel File Systems

High-performance parallel file systems are abundant in the literature: PPFS [27], Galley [37], Vesta [16], Swift [10], CFS [38], SFS [33], and the SIO specification [6]. However, most assume performance-homogeneous devices; thus, performance is dictated by the slowest component in the system.

Further, devoid of a specific programming model, applications could be constructed in an single-program, multiple-data (SPMD)-like fashion; thus, even if the parallel file system could deliver consistent high-performance, it would go wasted inside of a rigidly-designed program.

More advanced parallel file systems have specified higher-level interfaces to data via collective I/O (a similar concept is expressed with two-phase I/O) [30, 13]. In the original paper, Kotz found that many scientific codes show tremendous improvement by aggregating I/O requests and then shipping them to the underlying I/O system; the I/O nodes can then schedule the requests, and often noticeably increase delivered bandwidth. However, because requests are made by and returned to specific consumers, load is not balanced across those consumers dynamically. Thus, though these types of systems provide more flexibility in the interface, they do not solve the problems we believe are common in today's clustered systems.

Finally, there has been recent file-system work extolling the virtue of "adaptive" systems [35, 44]. As hardware systems increase in complexity, it can be argued that more intelligent software systems are necessary to extract performance from the underlying machine architecture. Whereas some of these systems employ off-line reorganization to improve global performance [35], the goal of River is balance load on-line (at run-time). However, long-term adaptation could also be useful in our system.

5.2 Programming Environments

There are a number of popular parallel programming environments that support the SPMD programming style, including messaging passing environments such as MPI [47] and PVM [21], as well as explicit parallel languages, such as Split-C [18]. These packages all provide a simple model of parallelism to the user, thus allowing the ready construction of parallel applications. However, none provide any facility to avoid run-time perturbations or adapt to hardware devices of differing rates. Our own experience in writing a parallel, external sort in Split-C led us to realize some of the problems with the SPMD approach; while it was possible to run the sort well *once* (NOW-Sort broke the world record on two database-industry standard sorting benchmarks), it was difficult to attain a high-level of performance consistently [2, 3].

There have been many parallel programming environments that are aligned with our River design philosophy of run-time adaptivity. Some examples include Cilk [7], Lazy Threads [23], and Multipol [12]. All of these systems balance load across consumers in order to allow for highly-irregular, fine-grained parallel applications.

The main difference between River and the systems above is the granularity of communication. Because River limits itself to I/O workloads, data is pushed through the interconnect in large-sized blocks. All of these other systems are run-times for general-purpose parallel programming, with a focus on fine-grained or irregular applications. On today's clusters, latency to remote memory is much higher than latency to local memory, perhaps by two orders of magnitude (10 microseconds versus 100 nanoseconds). This forces locality to be the dominant issue in many of the systems. However, remote I/O bandwidth is no worse than local I/O bandwidth; hence, while difficult to hide remote memory latency, I/O data can be pushed through the system with little cost. Further, none of these systems attempt to deal with the problem of slow producers, which is important in our environment.

Perhaps more similar to the River environment is Linda, which provides a shared, globally-addressable, tuple-space to parallel programs [11, 22]. Applications can perform atomic actions on tuple-space, inserting tuples, and then querying the space to find records with certain attributes. Because of the generality of this model, high performance in distributed environments is difficult to achieve [4]. Thus, while the distributed aspects of River could be built on top of Linda, they would likely suffer from performance and scaling problems.

5.3 Databases

Perhaps most relevant to River is the large body of work on parallel databases. Data flow techniques are well-known in the database literature [19], as it stems quite naturally from the relational model [14].

One example of a system that takes advantage of unordered processing of records is the IBM DB2 for SMPs [32]. In this system, shared data pools are accessed by multiple threads, with faster threads acquiring more work. This is referred to as "the straw model", because each thread "slurps" on its data straw at a (potentially) different rate. Implementing such a system is quite natural on an SMP; a simple lock-protected queue will suffice, modulo performance concerns. With River, we argue that this same type of data distribution can be performed on a cluster, due to the bandwidth of the interconnect.

There are a number of parallel databases found in the literature, including Gamma [15], Volcano [24], and Bubba [20].

These systems all use similar techniques to distribute data among processes. Both the Gamma *split table*, Volcano *exchange operators*, and a generalized split table known as a "river" in [5], are used to move data between producers and consumers in a distributed memory machine; however, all use static data partitioning techniques, such as hash partitioning, range partitioning, or round robin. These functions all do not adapt at run-time to load variations among consumers.

Current commercial systems, such as the NCR TeraData machine, exclusively use hashing to partition work and achieve parallelism. A good hash function has the effect of dividing the work equally among processors, providing consistent performance and achieving good scaling properties. However, as Jim Gray recently said of the TeraData system, "The performance is bad, but it never gets worse" [25]. Consistency and scalability were the goals of the system, perhaps at the cost of getting the best use of the underlying hardware.

6 Future Work

In the future, there are many research areas which we wish to explore. The first three of these are enhancements to the system infrastructure, and will serve to move the system from the realm of a system for expert programmers to one more easily used.

- **Process and Data Placement**. Process placement and data placement are two important decisions which are currently determined entirely by the user in River. In the *ideal* system, such decisions would be automated, perhaps by a higher-level entity such as a compiler or query planner.
- **Process and Data Migration**. River currently moves data through the system quite effectively. Initial experience suggests the feasibility of code migration, which would also improve the dynamic performance properties of the system. Long-term data migration would also be useful; in this, short-term locally optimal placement decisions could be re-evaluated and perhaps result in data movement to optimize for current usage. This would especially be useful
- **Application Fault Tolerance.** The ultimate goal is to write applications to the River interface that not only have robust performance, but also can continue operation under machine failure, similar to work in other dynamic programming environments [8, 43]. Some form of automatic check-pointing may be the solution, as suggested in [5].

We also believe River is well-suited to a large class of external, distributed applications, including traditional scientific codes and perhaps multimedia programs as well. Some evidence for this exists in the literature about Volcano [51], where scientific data-intensive applications are programmed and optimized in the Volcano data-flow environment. We plan on exploring how to add robust performance features into these types of applications.

Finally, we are developing simple models of how various "performance faults" should affect the system. With well-developed analytical models, we will be able to easily compare the performance of our system versus the theoretical ideal in any given perturbation scenario.

7 Conclusions

As hardware and software systems spiral in size and complexity, systems that are designed for controlled environments will experience serious performance defects in real-world settings. This has long been realized in the area of wide-area networking, where the end-to-end argument [42] pervades the design methodology of protocol stacks such as TCP/IP. In such systems, it is clear that a globally-controlled, well-behaved environment is not attainable; therefore, applications in the system treat it as a *black box*, adjusting their behavior dynamically based on feedback from the system to achieve the best possible performance under the current circumstances.

Complexity has slowly grown beyond the point of manageability in smaller distributed systems as well. Comprised of largely autonomous, complicated, individual components, clusters exhibit many of the same properties (and hence, the same problems) of larger scale, wide-area systems. This problem is further exacerbated as clusters move towards serving as a general-purpose computational infrastructure for large organizations. As resources are pooled into a shared computing machine, with hundreds if not thousands of jobs and users present in the system, it is clearly difficult, if not impossible, to believe that the system will behave in an orderly fashion.

To address this increase in complexity and the corresponding decrease in predictability, we introduce River, a substrate for building I/O-intensive cluster applications. River is a confluence of a programming environment and an I/O system; by extending the notion of adaptivity and flexibility from the lowest levels of the system up into the application, River programs can reliably deliver high performance. Even when system resources are over-committed, performance of applications written in this style will degrade gracefully, avoiding sudden (and often frustrating) prolongations in expected run time.

From our initial study of applications, we found that avoiding perturbations among consumers is relatively straight-forward via distributed queues. One important issue in balancing load is the *granularity* of ordering required by the applications. The most fine-grained applications (those that can balance load on the level of the individual records) are the simplest to construct in a performance-robust manner. While distributed queues have proven excellent as load balancers, they do require the programmer to insert them where appropriate in the flow.

Avoiding perturbations at the producers is the other problem solved by River, with graduated declustering. By dynamically shifting load away from perturbed producers, the system delivers the proper proportion of available bandwidth to each client of the application.

For high-performance I/O in clusters, getting *consistent* performance is easy (it can always be bad); getting *peak* performance is a matter of persistence (one good run when everything is "just right"); getting both is the goal of the River I/O environment.

Source code is available upon request.

8 Acknowledgements

First and foremost, we would like to thank Jim Gray for all of his advice and encouragement. We'd also like to thank Alan Mainwaring for his work on and support of Active Messages; it is the *sine qua non* of our work. Many thanks to the anonymous reviewers for all of their helpful comments. Finally, thanks to Andrea Arpaci-Dusseau, Amin Vahdat, and the I-Store group at Berkeley for suggestions that improved the presentation and content of the paper.

This work was funded in part by DARPA F30602-95-C-0014, DARPA N00600-93-C-2481, NSF CDA 94-01156, NASA FDNAGW-5198, and the California State MICRO Program.

References

[1] Supercomputers: Plug and Play. *The Economist*, November 1998.

[2] 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 *SIGMOD '97*, May 1997.

[3] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *SPDT '98*, Aug. 1998.

[4] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar. 1992.

[5] T. Barclay, R. Barnes, J. Gray, and P. Sundaresan. Loading Databases Using Dataflow Parallelism. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(4):72–83, December 1994.

[6] B. Bershad, D. Black, D. DeWitt, G. Gibson, K. Li, L. Peterson, and M. Snir. Operating system support for high-performance parallel I/O systems. Technical Report CCSF-40, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, July 1995.

[8] R. D. Blumofe and P. A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In USENIX, editor, *1997 Annual Technical Conference, January 6–10, 1997. Anaheim, CA*, pages 133–147, Berkeley, CA, USA, Jan. 1997. USENIX.

[9] N. Boden, D. Cohen, R. E. Felderman, A. Kulawik, and C. Seitz. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, February 1995.

[10] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, Fall 1991.

[11] N. J. Carriero. *Implementation of tuple space*. PhD thesis, Department of Computer Science, Yale University, December 1987.

[12] S. Chakrabarti, E. Deprit, E.-J. Im, J. Jones, A. Krishnamurthy, C.-P. Wen, and K. Yelick. Multipol: A Distributed Data Structure Library. Technical Report CSD-95-879, University of California, Berkeley, July 1995.

[13] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[14] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970. Also published in/as: 'Readings in Database Systems, 3rd Edition', M. Stonebraker and J. Hellerstein, Morgan-Kaufmann, 1998, pp. 5–15.

[15] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):99–108, Sept. 1988.

[16] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[17] T. P. Council. TPC-D Individual Results, 1998. http://www.tpc.org/results/tpc_d.results.page.html.

[18] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.

[19] 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.

[20] D. J. DeWitt, S. Ghandeharizadeh, and D. Schneider. A Performance Analysis of the Gamma Database Machine. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):350–360, Sept. 1988.

[21] G. Geist and V. Sunderam. The Evolution of the PVM Concurrent Computing System. In *COMPCON*, February 1993.

[22] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in Linda. In D. Degroot, editor, *1985 International Conference on Parallel Processing*, pages 255–263, 1985.

[23] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, Aug. 1996.

[24] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):102–111, June 1990.

[25] J. Gray. What Happens When Processors Are Infinitely Fast And Storage Is Free? Invited Talk: 1997 IOPADS, November 1997.

[26] H.-I. Hsiao and D. DeWitt. Chained Declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.

[27] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

[28] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. GRACE: Relational algebra machine based on hash and sort — its design concepts. *Journal of the Information Processing Society of Japan*, 6(3):148–155, 1983.

[29] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner. Symmetric Multiprocessing in Solaris 2.0. In *Proceedings of COMPCON Spring '92*, 1992.

[30] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[31] S. Kubica, T. Robey, and C. Moorman. Data parallel programming with the Khoros Data Services Library. *Lecture Notes in Computer Science*, 1388:963–973, 1998.

[32] B. Lindsey. SMP Intra-Query Parallelism in DB2 UDB. Database Seminar at U.C. Berkeley, February 1998.

[33] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Technical Conference*, pages 291–305, 1993.

[34] A. Mainwaring and D. Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report CSD-96-918, University of California at Berkeley, October 1996.

[35] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 238–251, Saint-Malo, France, October5–8 1997. ACM SIGOPS, ACM Press.

[36] R. V. Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, Jan. 1997.

[37] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[38] B. Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[39] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*. ACM Press and IEEE Computer Society Press, 1995.

[40] G. Papadopolous. Untitled. Talk at Winter NOW Retreat, July 1997.

[41] D. M. Ritchie. A Stream Input-Output System. *BLTJ*, 63(8, Part 2):1897–1910, October 1984.

[42] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, pages 277–288, November 1984.

[43] D. Scales and M. Lam. Transparent Fault Tolerance for Parallel Applications on Networks of Workstaions. In *Proceedings of the 1996 USENIX Conference*, Jan. 1996.

[44] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Systems. In *Proceedings of the 1997 Workshop on Hot Topics on Operating Systems*, Chatham, MA, May 1997.

[45] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, Sept. 1986.

[46] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga:providing data management support for scientific visualization applications. In *International Conference On Very Large Data Bases (VLDB '93)*, pages 25–38, San Francisco, Ca., USA, Aug. 1993. Morgan Kaufmann Publishers, Inc.

[47] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883, November 1993.

[48] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 40–53, December 1995.

[49] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

[50] R. Winter and K. Auerbach. The Big Time: 1998 Winter VLDB Survey. *Database Programming and Design*, 1998.

[51] R. Wolniewicz and G. Graefe. Algebraic Optimization of Computations over Scientific Databases. In *VLDB '93*, pages 13–24, 1993.