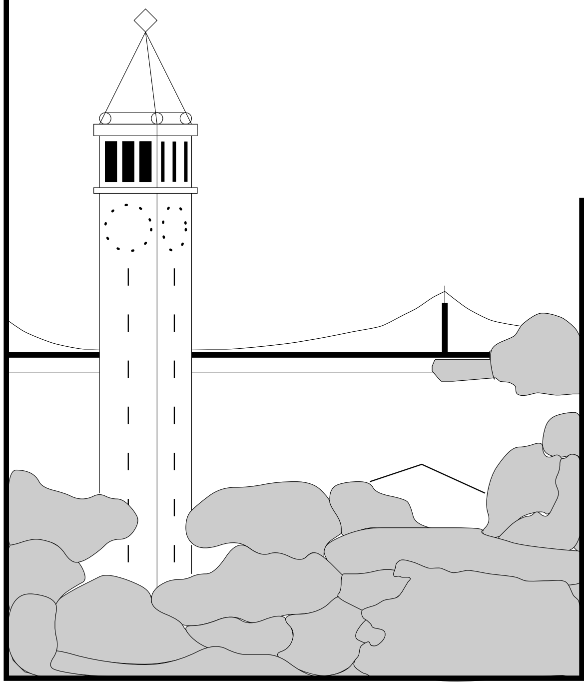
Enhancing Graduated Declustering for Better Performance Availability on Clusters

Noah Treuhaft



Report No. UCB/CSD-00-1118

November 16, 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

We present three enhancements to graduated declustering, a mechanism for improving the performance robustness of I/O-intensive parallel programs. These enhancements are called write support, primary copies, and logical partitioning. They serve to increase the number of programs to which we can successfully apply graduated declustering. We describe their need, present their design and implementation, and evaluate their performance.

1 Introduction

Clusters of commodity servers, workstations, or personal computers are an excellent platform for I/O-intensive software. They offer good scalability for processor, memory, communication, and especially storage resources, all of which can increase linearly with the number of nodes in the system. Clusters also offer incremental scalability: given scalable software, they accommodate increasing workloads with a simple addition of more nodes and switches. Finally, clusters offer a good cost-to-performance ratio, benefiting from the continually-improving technologies and economies of scale for components such as microprocessors, DRAMs, and magnetic disk drives.

In this work, our interest is in identifying techniques that we can apply when building high-performance I/O-intensive parallel programs for clusters. Typically, writing software that attains the underlying hardware's peak performance is a difficult task, requiring an in-depth understanding of both application algorithms and hardware performance characteristics. Moreover, the cluster's parallel environment complicates both software and hardware.

Despite the challenges, prior work in this area has achieved great successes. Shared-nothing parallel relational database systems (which run on clusters) are available from a number of vendors, and they currently hold the highest rankings for the Transaction Processing Performance Council's three database benchmarks (TPC-C, TPC-H at 1000 GB, and TPC-R) [8]. And beginning with NOW-Sort [1] in 1997, parallel sorting programs running on clusters have continually held the MinuteSort sorting-benchmark record [6].

Experience gained from building and tuning NOW-Sort [2, 4] helped Arpaci-Dusseau [3] to identify an interesting problem on clusters: *performance faults*. Performance faults are unexpected fluctuations in the performance of the components that comprise the system. Parallel programs that assume uniform performance from system components will run afoul of these faults, experiencing unpredictable load imbalance and, in turn, unexpectedly-low performance.

In response to the problem of performance faults, Arpaci-Dusseau presented the concept of *performance availability*. A performance-available parallel application is robust to performance faults; its performance degrades gracefully in their presence. With the thesis that modern parallel and distributed systems must provide the proper primitives to support performance availability, two performance-availability mechanisms — the *distributed queue* and *graduated declustering* — were presented in the context of River, a data-flow programming environment for the construction of performance-robust I/O-intensive applications. These mechanisms were shown to be sufficient for constructing a variety of database primitives that are robust to disk performance faults. (River, the distributed queue, and graduated declustering were initially described elsewhere by Arpaci-Dusseau et al. [5])

In this report, we present three enhancements to the previous implementations of graduated declustering [3, 5], which we refer to as *basic graduated declustering*. These three enhancements are

- write support,
- primary copies, and
- logical partitioning.

They serve to expand graduated declustering's usefulness as a performance-robustness mechanism. In addition to describing their design and implementation, we present performance results that compare graduated declustering with our enhancements to basic graduated declustering, and also to a non-robust parallel program.

The remainder of this report proceeds as follows. Section 2 provides background information on our software model, on performance robustness, and on performance-robustness mechanisms. Section 3 presents the design and implementation of our three enhancements to basic graduated declustering, and Section 4 evaluates their performance. Finally, Section 5 discusses related issues and future directions, and Section 6 concludes.

2 Background

This section provides background information. We first present our software model, describing the type of I/O-intensive parallel software that this work addresses. We also look at performance robustness as it applies to these programs. Finally, we describe two performance-robustness mechanisms: the distributed queue and basic graduated declustering.

2.1 Software Characteristics

We are interested in SPMD (single program multiple data) parallel programs that spend a significant fraction of their running time in one or more I/O-intensive phases. These SPMD programs consist of a number of operating-system processes distributed across some or all of the cluster's nodes, with at most one process per node. During the I/O-intensive phases, all processes read or write large, contiguous blocks of data. (Random-access workloads are not our focus.) These phases are *I/O-intensive* because, for their duration, all processes are nearly or completely I/O-bound, so that a slowdown in any node's I/O performance will create a bottleneck, slowing the process on that node. Each I/O phase ends with a barrier, requiring that every process completes the current phase before any process proceeds to the next phase. As a result, slowdown of even a single process will result in global program slowdown.

All data accessed by all of the processes during a given phase should be thought of as belonging to a single parallel file, which is to say that the data comprise a single dataset that has been *partitioned* (or *declustered*, in database terminology) across some or all cluster nodes, and is stored on the disk drives attached to those nodes. The pieces into which a dataset is partitioned are called *fragments*, and there is (at most) one fragment stored per node. Partitioning is performed according to application criteria, so fragments need not be equally sized, but we assume that each is large — a hundred megabytes or more. During a single I/O phase, each process in a parallel program accesses one fragment, so in a sense each process has responsibility for one contiguous range of bytes within the parallel file.

This software model is similar to that of NOW-Sort, and also reflects the execution of some query operators in parallel relational database management systems.

2.2 Performance Robustness

Any cluster is likely to exhibit a variety of performance faults. This work focuses on *disk performance faults* [3], which affect the performance of disk I/O. Such faults arise from a number of sources, including unpredictability of system activity and of data layout on disk drives.

As explained in the previous section, a slowdown in any node's I/O system will lead to global program slowdown. The significance of this is clear given the presence of disk performance faults, which will cause that exact problem. In building performance-robust parallel programs, we seek to minimize these slowdowns. The goal of the robustness mechanisms discussed in the next section is just that: to minimize the effect of disk performance faults and thereby avoid global program slowdown.

2.3 Two Performance Robustness Mechanisms

Prior work [3, 5] presents two mechanisms that we can employ to add disk-performance-fault robustness to the I/O-intensive parallel programs described above in Section 2.1. In this section, we outline those mechanisms, pointing out key details of their design, implementation, and operation. Although Arpaci-Dusseau [3] presents them as general producer-to-consumer data transfer mechanisms, we simplify our discussion by considering them solely in the context of disk I/O.

Both mechanisms transfer data between disks and processes in units that we call *blocks*. Blocks may be fixed or variable in size, as appropriate for the application's needs and the system's performance characteristics. (We explore the latter in Section 4.) Both mechanisms also rely on *performance redundancy* to provide performance availability, in much the same way that fault-tolerant systems rely on redundancy to provide availability. And as a basic strategy for achieving performance robustness, they both perform dynamic load balancing for I/O.

The overall goal of both mechanisms is the same: to prevent disk performance faults from significantly slowing any process in a parallel program relative to its peers. In this way they seek to provide graceful degradation of program performance in the face of disk performance faults. But as we shall see in the next two sections, their dynamic load-balancing strategies for achieving this goal are very different.

2.3.1 The Distributed Queue: Write Robustness

The distributed queue provides performance robustness to parallel programs as they write parallel files. It accomplishes this goal by combining flexibility in application-level data-ordering requirements with credit-based flow control. The specific strategy of the distributed queue is to ensure that all processes writing to a parallel file avoid slower disks, thereby sending more data to faster disks. The essence of the algorithm behind the distributed queue is to monitor the number of write requests queued at each disk and to make load-balancing decisions, which are enabled by flexibility in data ordering, based on that information.

The distributed queue requires that a parallel program's semantics does not require ordering to be maintained between blocks of data written by different processes, although it does maintain ordering between blocks written by the same process. This leaves it free to write any block to any fragment.

This flexibility provides the performance redundancy necessary for performance availability. Each process can write to any disk, and thus all processes can avoid slow disks. Ensuring that processes avoid slow disks is the distributed queue's basic strategy. The dynamic load-balancing algorithm that implements it is as follows.

Each process begins by allocating an equal number of credits to each disk. Then, independently, each process repeatedly selects a disk at random and, provided that the disk has at least one credit, sends a block to (and subtracts a credit from) the selected disk. If the selected disk has no credits, another is randomly selected. The requests sent to a given disk are queued at that disk; when serviced, they generate a reply that returns one credit to the corresponding process.

In this fashion, the number of requests enqueued by all processes at any one disk is limited, and faster disks will tend to receive and service more requests than slower disks. If any disk should suffer a performance fault, it will receive fewer requests and in turn be charged with writing fewer bytes. Probabilistically, the amount of data handled by any disk is proportional to its performance (i.e., its delivered write bandwidth).

2.3.2 Basic Graduated Declustering: Read Robustness

Graduated declustering provides performance-robustness to parallel programs as they read parallel files. It accomplishes this goal by combining data replication with dynamic load balancing based on progress. The specific strategy of graduated declustering is to ensure that all processes reading from a parallel file make progress at the same rate. The essence of the algorithm behind graduated declustering is to monitor the progress of each process and to make load-balancing decisions, which are enabled by replication, based on that information.

Graduated declustering assumes that each process reads one fragment in its entirety. More significantly, graduated declustering assumes replication of the dataset. The primary motivation for this replication is often availability of the traditional sort, but performance availability comes as a welcome side benefit.

In particular, graduated declustering assumes that the dataset has been replicated according to the *chained-declustering* strategy [7]. Under this approach, each fragment is replicated, and the two replicas — called the *primary fragment copy* and the *backup fragment copy*— are laid out as follows. With N nodes and N fragments, both numbered $0 \dots N-1$, the primary copy of fragment i is stored on node i and the backup copy of fragment i is stored on node $(i+1) \bmod N$.

This replication strategy provides the performance redundancy that is necessary for performance availability. Each process can read its fragment simultaneously from two disks, and (with the proper mechanism in place) it can expect more bandwidth from one disk if the other should suffer a performance fault.

As mentioned, graduated declustering's strategy is to ensure that all processes make progress at the same rate. Thus at any time during an I/O phase, each process should have read the same fraction of its fragment. Since fragments may vary in size, progress is measured by the relative fraction read, rather than the absolute amount read. The load-balancing algorithm that implements this strategy is as follows.

Each process reads from both of its fragment copies, so that each disk has two readers. If a disk observes that one of its readers has fallen behind the other, then it gives preference to the requests of the slow reader, serving its requests until the slow reader has caught up. Clearly, processes must inform disks of their progress; they do so by packaging that information with each read request.

If either (or even both) of the disks storing a process's fragment copies should suffer a performance fault, this algorithm will effectively steal bandwidth from the other two processes whose fragment copies are located on the same disks. As a result, all readers will tend to make progress at approximately the same rate in spite of disk performance faults.

As discussed by Arpaci-Dusseau [3], there are limits to graduated declustering's tolerance of performance faults because only two replicas are available. Too-numerous or too-severe performance faults will cause uncorrectable load imbalance. Still, graduated declustering robustly accommodates many common disk performance fault scenarios. And although we do not explore this option, even-greater robustness to performance faults can be achieved by generalizing graduated declustering to use three or more replicas.

As an implementation note, we point out that the functionality of graduated declustering resides in a runtime software layer that lies beneath the parallel program, between it and the operating system's file-access interfaces. This layer must be aware of the chained-declustering fragment layout, and must handle the routing of requests to the appropriate disks, the tracking each process's progress, the transmission of that progress information to the disks, and the scheduling of requests at the disks. Rather than accessing file data through operating-system interfaces, parallel programs access file data through the interface to this graduated-declustering layer. The implementation of this layer then accesses file data through operating-system interfaces on the program's behalf.

3 Three Enhancements to Basic Graduated Declustering

This section presents our three enhancements to basic graduated declustering: write support, primary copies, and logical partitioning. We describe them in turn, covering the motivation, design, implementation, and benefits of each.

3.1 Write Support

Write support extends basic graduated declustering to handle writing data as well as reading it, permitting the use of graduated declustering in place of the distributed queue. Graduated declustering with the write-support enhancement has two advantages over writing with the distributed queue.

First, equal progress for each process is an explicit goal with graduated declustering, but not for the distributed queue. This will lead to better robustness (and better overall performance) for graduated declustering under some circumstances, such when the cluster interconnect is congested [3, pp. 86–7].

Second, graduated declustering offers data placement that is far more predictable. Data from a given process will be routed to one of only two disks (those that hold the fragment's primary and backup copies), whereas the distributed queue may route data to any disk in the system. An important benefit of this difference is that ordering can easily be maintained between blocks written by different processes. This ability to maintain ordering expands the range of programs to which we can add performance robustness.

The addition of write support to graduated declustering is based on the simple observation that the basic graduated-declustering algorithm can work just as well for writing as it does for reading, provided that each process knows in advance how much data it will write. Performance redundancy takes a different form here. Rather than having two fragment copies from which a process may read each block, the write-support enhancement has two fragment copies to which a process may write each block. (In fact, the term fragment copy is a misnomer in this context because each one collects only part of the data from the process that writes to it. They might more accurately be called fragment portions. However, for consistency we will continue to refer to them as fragment copies.)

The strategy is then to ensure that all processes writing to a parallel file make progress at the same rate, and it essentially achieved by a simple reversal the flow of data in graduated declustering. That is, data is transfered from process to disk, rather than from disk to process. The only other noteworthy difference from basic graduated declustering is that we must maintain metadata regarding the ultimate destination (primary or backup fragment copy) of each block if we wish to locate particular blocks later on.

Note that the write-support enhancement does not provide replication when used with the chained-declustering fragment layout. Replication requires that that each byte be written to both its primary and backup fragment copies, thus eliminating the source of performance redundancy for writing. If replication is required, it can be postponed until the parallel program has completed, when missing bytes from the primary fragment copy can be copied from the backup fragment copy and vice-versa. Or it can be achieved with a different fragment layout that provides for writing a second copy of each process's output. Compared to the chained-declustering fragment layout, this approach requires twice the number of disk drives for equivalent performance.

3.2 Primary Copies

The primary-copies enhancement minimizes graduated declustering's usage of the backup fragment copies. In the absence of performance faults (or, more accurately, when all disks offer the same performance), graduated declustering with the primary-copies enhancement will not send any requests to the backup fragment copies. Contrast this with basic graduated declustering, which will send half of all requests to the backup fragment copies. This change in behavior relative to basic graduated declustering provides three significant benefits.

First, in the absence of performance faults, the disk drives will not be multiplexed: they will serve requests only for the primary fragment copy that they store. In contrast, under basic graduated declustering, all disk drives must constantly serve requests for both the primary and backup fragment copies that they store. When fragment copies are laid out contiguously on disk, the disk-head seeks generated by this multiplexing eat into delivered disk bandwidth, a potentially serious problem for I/O-intensive programs.

Arpaci-Dusseau notes this overhead and minimizes it to 5% (i.e., only 5% of the disks' peak bandwidth is lost) by selecting a large block size [3, p. 113]. We observe that his approach is based on the performance of the two particular disk-drive models used in the experiment, which have peak transfer bandwidths of about 5 MB/s and 9 MB/s. As technology improvements increase transfer bandwidth — current drives are reaching t0 MB/s, and the growth trend is about 40% per year — this approach requires that block size scale accordingly. But, as also noted, excessively large blocks limit graduated declustering's ability to achieved its goal of equal progress for all processes [3, p. 126].

Second, both graduated declustering and the distributed queue assume that node I/O buses and the cluster interconnect offer sufficient excess bandwidth to support the mechanisms' requisite data transfers. Notably, in the absence of any performance faults, graduated declustering still transmits across the network half of all data read from disk, and the distributed queue probabilistically transmits $(N-1)/N$ of all data written to disk, where N is the number of processes in the parallel program. Each transfer consumes bandwidth on the sender's I/O bus, on the network links and switches, and on the receiver's I/O bus [4]. If a program's intrinsic demand for these resources does not leave sufficient headroom, performance-robustness mechanisms may cause them to become oversubscribed, in turn causing program performance to suffer.

Third, as we shall see below in Section 4.2.2, the primary-copies enhancement reduces the use of backup fragment copies (and improves performance) even in the presence of performance faults.

The primary-copies enhancement is a straightforward change to the basic graduated-declustering algorithm, in which a process always reads from both its primary and backup fragment copies. In contrast with the basic algorithm, a process begins to read from its backup fragment copy only when it falls behind the process whose primary fragment copy resides on the same disk.

Specifically, let there be N nodes, processes, and fragments, all numbered $0 \ldots N-1$; let process i reside on node i; let the primary and backup fragment copies reside on nodes i and $i+1 \bmod N$, respectively; and let $P(i) \in [0,1]$ represent the progress of process i (which can be thought of as the fraction process i's I/O completed so far). Then process i begins reading from its backup fragment copy only when $P(i + 1 \bmod N) - P(i) > high\_watermark$. Likewise, it stops reading from its backup fragment copy when $P(i + 1 \bmod N) - P(i) < low\_watermark$. By setting the $low\_watermark$ to be less than $high\_watermark$, we allow the difference in progress between two adjacent (i.e., consecutively numbered, modulo N) processes to wander within an acceptable range, as determined by the watermarks. Values for the watermarks should be chosen to meet program needs.

One key to implementing this enhancement is keeping each process i apprised of the quantity $P(i + 1 \bmod N)$, the value of which is known to the disk that holds process i's backup fragment copy since it also holds the primary fragment copy for process $i+1 \bmod N$. When a process is accessing its backup fragment copy, this information can be included in messages sent between the process and the backup-copy disk. On the other hand, when a process is not accessing its backup fragment copy, periodic progress update messages must be sent to it by the backup-copy disk.

3.3 Logical Partitioning

Logical partitioning accommodates parallel programs whose processes do not wish to read exactly one fragment of a parallel file. An example is a parallel sorting program running on a heterogeneous cluster, in which some nodes have significantly faster processors or significantly more RAM than other nodes. Here, the best performance is obtained by performing more work on the more-powerful nodes. And this requires that those nodes read or write more data than the others.

To accomplish this, the logical-partitioning enhancement allows programs to create a *logical partitioning* of the parallel file, in which each process specifies an offset and range that together describe the portion of the parallel file that is of interest to it. The graduated-declustering implementation is modified to use a process's progress in reading or writing its logical partition when making load balancing decisions, rather than measuring progress relative to a single fragment, as the basic graduated-declustering algorithm does. Additionally, at runtime the graduated-declustering layer must determine which disk can handle a given access by comparing its offset and length against metadata that describes the size, ordering, and location of the parallel file's fragment copies.

4 Evaluation

In this section we evaluate the performance of the three enhancements described in the previous section. We begin with the primary-copies enhancement, and then move on to the logical-partitioning enhancement. Because it seems natural to compare read performance with write performance, results for the write-support enhancement are given along the way. Before beginning, though, we first describe our experimental setup.

4.1 Experimental Setup

All experiments were performed on a cluster of 16 Sun Ultra 1 Model 170 workstations. Each workstation contains the following hardware:

- a 167 MHz UltraSPARC-I processor with separate 16 KB first-level instruction and data caches and a 512 KB second-level cache;
- 128 MB of RAM;
- two Seagate ST32430WC (Hawk 2LP, 2.15 GB, 5400 RPM) disk drives, both attached to the workstation's internal fast-narrow (10 MB/s) SCSI bus and have outer and inner track bandwidths measured at 5.45 MB/s and 3.18 MB/s, respectively;
- a Myrinet network interface card attached to the SBus, capable of transfers at around 40 MB/s; and
- a builtin 10 Mb/s Ethernet network interface.

Each node runs the Solaris 2.6 operating environment, with additional drivers for the Myrinet network interface card to provide the Active Messages protocol, as well as Internet Protocol (IP) over Active Messages. Job startup is performed over the Ethernet network, but all other communication takes place over the Myrinet network using TCP sockets transported on IP over Active Messages. One of the two disk drives is reserved for operating system files and swap space, and the other disk drive is used for our experiments.

In many experiments, disk *perturbations* are introduced to simulate disk performance faults. A perturbation is a process that repeatedly reads from a disk file. Perturbations consume roughly 50% of a disk's bandwidth. When perturbations are applied to more than one disk, the perturbations are distributed in the "best" possible fashion, with (as much as possible) even spacing between them, to prevent uncorrectable load imbalance.

The data in all graphs reflect the results of five trials. Points are placed at mean values, and error bars extend one standard deviation above and below the mean. Note that error bars are present at every data point, but the variance of the data is frequently so small that the error bars are too short to be visible.

4.2 Primary Copies and Write Support

In this section, we look at the performance of graduated declustering with the primary-copies and write-support enhancements. To this end, we compare three different programs.

Non-Robust incorporates no performance-robustness mechanism, instead accessing data directly through `read` and `write` system calls.

Basic GD uses an implementation of basic graduated declustering with the addition of the write-support enhancement.

Primary-Copies GD uses an implementation of graduated declustering with both the primary-copies and write-support enhancements.

In each experiment, we run all three programs on 16 nodes, with one process per node. Each process reads or writes 150 MB of data, so the program as a whole accesses 2.4 GB. The 2.4 GB of data are partitioned into 16 fragments of 150 MB, stored one per node in a single file. The fragments are replicated according to the chained-declustering strategy for use by Basic GD and Primary-Copies GD. We used the Solaris `directio` system call to disable file-system caching for all fragment copies.

4.2.1 Performance in the Absence of Faults

Our first set of experiments considers the effect of block size on the performance of our three programs in the absence of any performance faults. The goal of these experiments is to gain insight into the overhead of the two graduated-declustering implementations.

The upper plot in Figure 1 shows read performance in the absence of faults as transfer size increases from 4 KB to 4096 KB. Specifically, it shows the bandwidth attained by the slowest process (the last to complete) in the program, which is representative of the performance of a disk-bound program absent faults.

Non-Robust attains peak disk bandwidth (approximately 5.45 MB/s) with 8 KB transfers. Transfers of 4 KB are too small to amortize the overhead of each disk request.

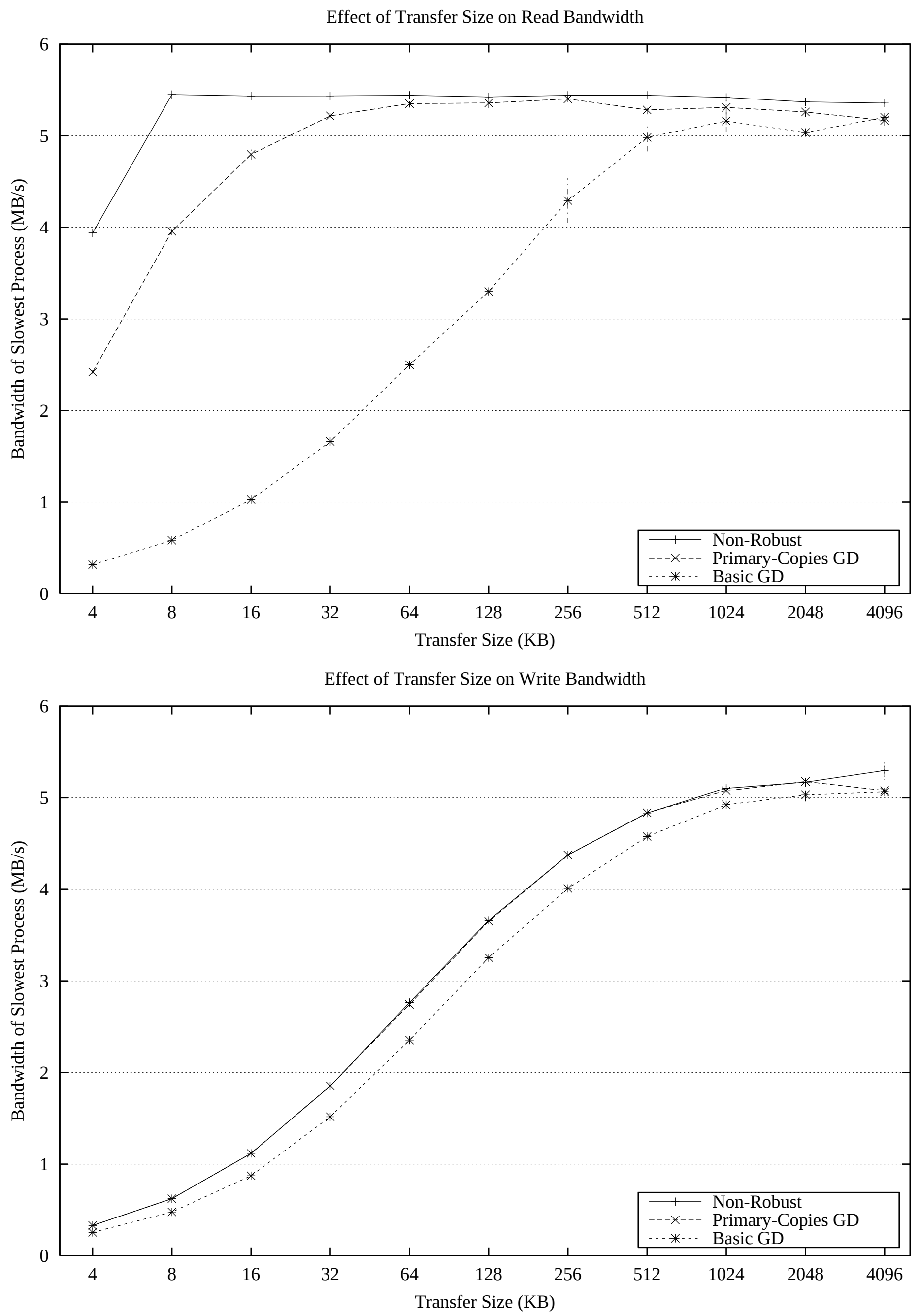


Figure 1: Effect of transfer size on performance. The bandwidth attained by the slowest of the 16 processes is shown. Each process accesses 150 MB. The block size is varied from 4 KB to 4096 KB. No faults are introduced.

Basic GD performs substantially worse than Non-Robust with transfers smaller than 512 KB. This slowdown is due to Basic GD's continual disk multiplexing. As transfer size grows, the cost of multiplexing is amortized over larger transfers (each of which completes without interruption), and performance improves accordingly. As Arpaci-Dusseau found with the same experimental environment [3, p. 113], 1024 KB transfers bring Basic GD's performance to within 5% of the system's peak (which is represented here by Non-Robust).

Primary-Copies GD tracks Non-Robust's performance closely with transfers of 32 KB and larger because, absent performance faults, it performs little or no disk multiplexing. However, it does perform noticeably slower than Non-Robust with transfers smaller than 32 KB. Such transfers are too small to hide the overhead of the graduated-declustering software, which (in our implementation) performs socket-based interprocess communication (IPC) to move data from the fragment copies into the processes. This IPC proceeds through the operating-system kernel, resulting in significant processor overhead when transfers are small. Small-transfer performance can be improved by changing the implementation to avoid the operating system during IPC operations, perhaps through the use of shared memory IPC or by restructuring the software to eliminate the need for per-transfer IPC entirely.

The lower plot in Figure 1 shows write performance in the absence of faults as transfer size increases from 4 KB to 4096 KB in the absence of performance faults. Relative to reading, writing exhibits a dramatic drop in small-transfer performance for both Non-Robust and Primary-Copies GD. The performance of both programs now tracks that of Basic GD rather closely. Further comparison of the two plots in Figure 1 reveals that Basic GD's performance is highly similar in both.

The moderate performance difference between Basic GD and the other two programs when writing is attributable to the disk-head seeks caused by Basic GD's disk multiplexing. The interesting conclusion to be drawn here is that the performance difference when reading is due *to a similar amount* to the disk-head seeks caused by multiplexing, and is due *to a much larger amount* to the loss of read prefetching in the disk drives (performed by on-drive control logic, not by the operating system) that multiplexing causes. Only by comparing multiplexing and non-multiplexing programs both when reading and when writing (where prefetching does not occur) do we see how significant the effects of prefetching and multiplexing on small-transfer read performance are.

Before moving on, we point out that better small-transfer write performance can be achieved by batching writes before issuing them to the disks, as is typically done by file system buffer caches. Solaris's buffer cache performs this optimization, resulting in substantial performance improvements in small-transfer write performance.

4.2.2 Performance in the Presence of Faults

Our second set of experiments considers the effect of performance faults on the performance of our three programs. The setup for these experiments is similar to that of Section 4.2.1, but with two differences:

1. the transfer size is fixed and

2. perturbations are introduced.

We vary the number of perturbed disks from zero to 16. As previously stated, our perturbations are designed to reduce a disk's performance by about 50%.

To help us evaluate how well each program copes with performance faults in an absolute sense, each graph in this section includes a line labeled *Ideal*, which is meant to represent the behavior of an ideal performance-robustness mechanism in the presence of performance faults. Its left endpoint is the mean bandwidth of all 16 Non-Robust processes when there are no perturbations, and its right endpoint is the mean bandwidth of all 16 Non-Robust processes when there are 16 perturbations. (Both are averaged over all five trials.)

Figure 2 shows the performance of all three programs with 64 KB transfers when reading (upper plot) and when writing (lower plot).

For reading, Non-Robust's performance performance is halved by the introduction of just a single fault — the canonical argument in favor of performance-robustness mechanisms.

Primary-Copies GD's performance suffers less, dropping by only 25% in the presence of one fault, and then degrading smoothly as more faults are added. The magnitude of the performance drop caused by a single fault is explained by the disk multiplexing that Primary-Copies GD performs in response to a performance fault.

As we saw in Section 4.2.1, Basic GD's performance with such small transfers is relatively poor in the absence of faults. Low as performance is to start, the introduction of a fault has little impact, and performance degrades smoothly as further faults are added. Note that Primary-Copies GD performs better than Basic GD even with faults.

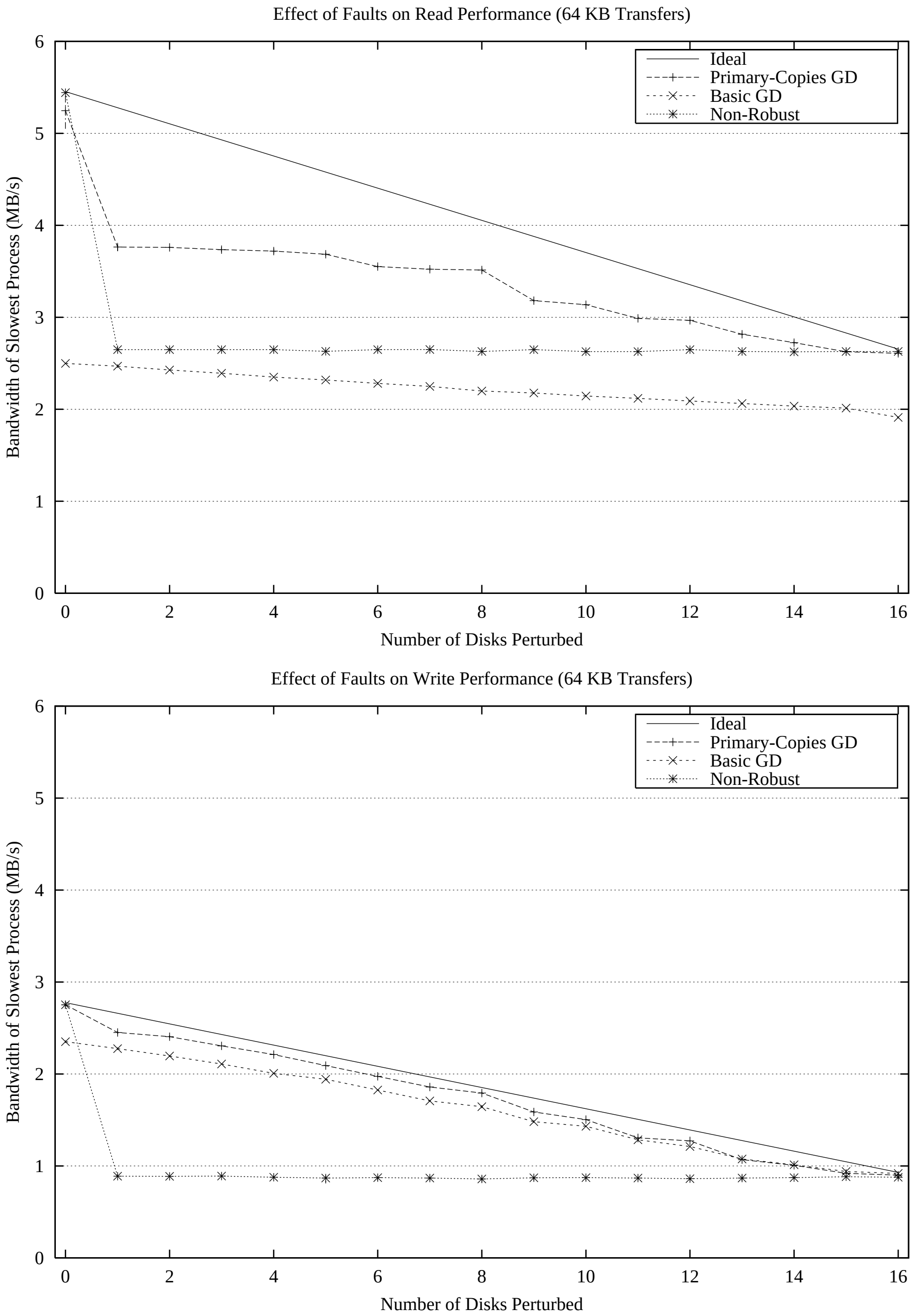
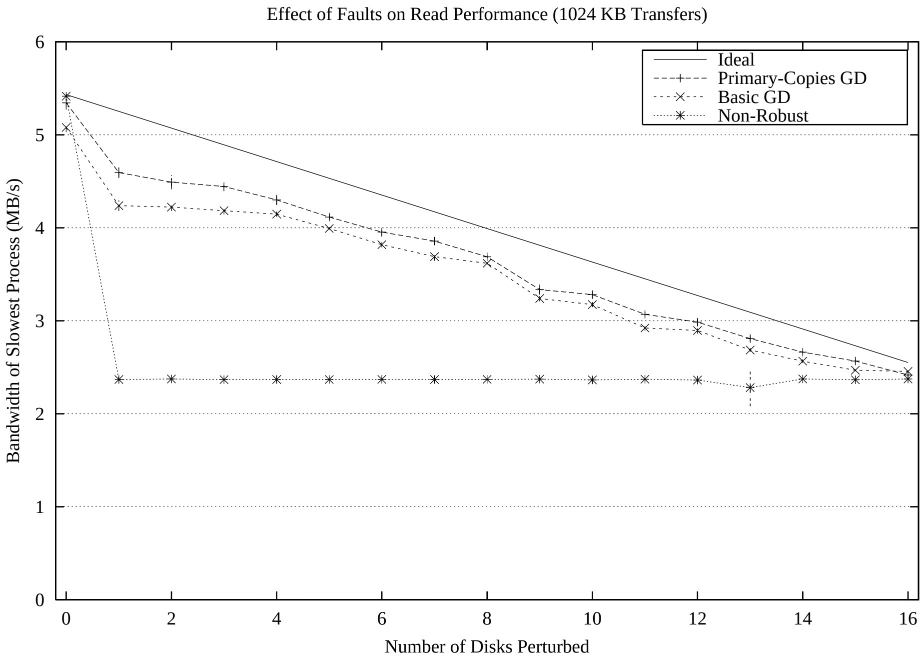


Figure 2: Effect of faults on performance. The bandwidth attained by the slowest of the 16 processes is shown. Each process accesses 150 MB. The block size is fixed at 64 KB. The number of faults is varied from zero to 16.



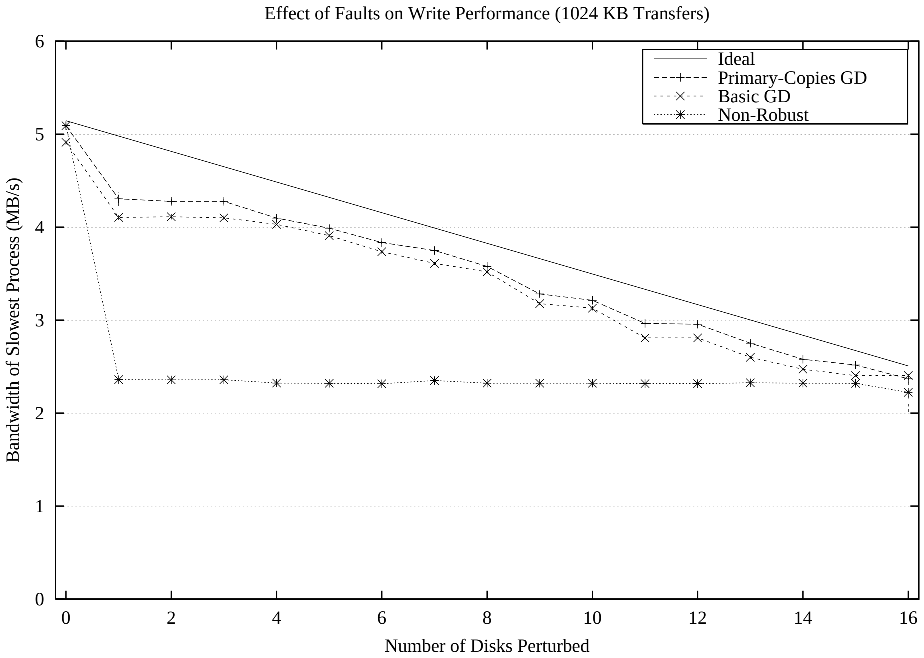


Figure 3: Effect of faults on performance. The bandwidth attained by the slowest of the 16 processes is shown. Each process accesses 150 MB. The block size is fixed at 1024 KB. The number of faults is varied from zero to 16.

For writing, we see the same poor small-transfer performance of Section 4.2.1, affecting all three programs. As a result, Primary-Copies GD and Basic GD offer similar performance, which is substantially better than Non-Robust's in the face of performance faults.

Figure 3 shows the performance with 1024 KB transfers when reading (upper plot) and when writing (lower plot). As can be seen, such large transfers are highly effective in improving the performance of Basic GD, which now runs only 5% to 10% slower than Primary-Copies GD, even in the absence of faults. They also bring both GD implementations close to the ideal performance.

4.3 Logical Partitioning

Our final experiment examines the performance of the logical-partitioning enhancement. Our interest here is in learning whether or not graduated declustering works as well with logical partitioning as it does without it.

To invoke the logical-partitioning enhancement, we vary the amount of data read and written by the processes. As in our previous experiments, the program as a whole handles 2.4 GB of data. Here, though, we change the ratio of data handled by the evenly-numbered and oddly-numbered processes, setting it at 1:1, 1:1.5, and 1:2. Thus with a 1:1 ratio, each process handles 150 MB; with a 1:1.5 ratio, evenly-numbered processes handle 120 MB while oddly-numbered processes handle 180 MB; and with a 1:2 ratio, evenly-numbered process handle 100 MB while oddly-numbered processes handle 200 MB.

These ratios might represent, say, the difference in memory size or processor speed between two groups of machines in a heterogeneous cluster. By alternating the assignment of the larger and smaller datasets on an even-odd basis, we eliminate some potential for uncorrectable load imbalance.

Figure 5 shows the performance of Basic GD and Primary-Copies GD with logical partitioning on 16 nodes when reading (upper plot) and writing (lower plot). Specifically, it shows the average per-process bandwidth computed based on the completion time for the slowest process in each program. The transfer size is 1024 KB. The number of perturbations varies from zero to 16. (In fact, the data for the 1:1 ratio are exactly those that were used to produce Figure 3.)

Changes in the data ratio have essentially no effect on performance, indicating that the logical-partitioning enhancement works well with both Basic GD and Primary-Copies GD..

Figure 4 shows the performance with 64 KB transfers. For both reading

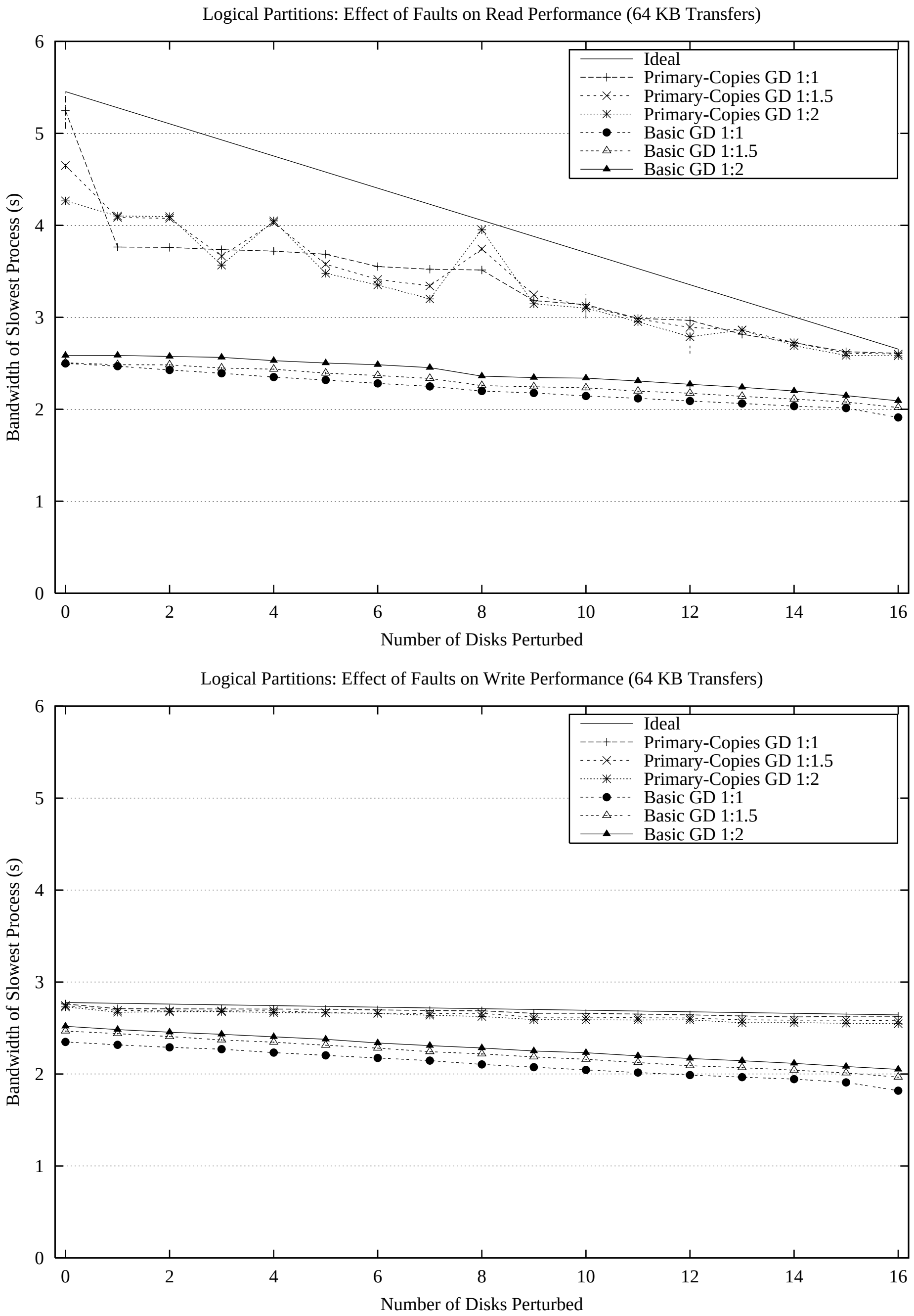
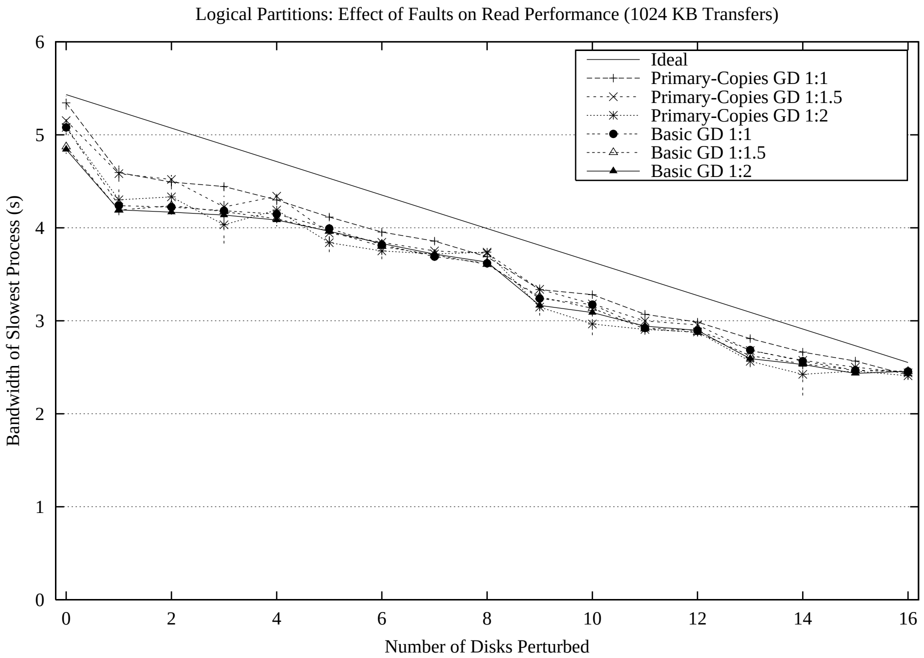


Figure 4: Logical partitioning: effect of faults on performance . The average per-process bandwidth (based on the completion time of the slowest process) is shown. The ratio of data accessed by evenly- and oddly-numbered processes is varied. The block size is fixed at 64 KB. The number of faults is varied from zero to 16.



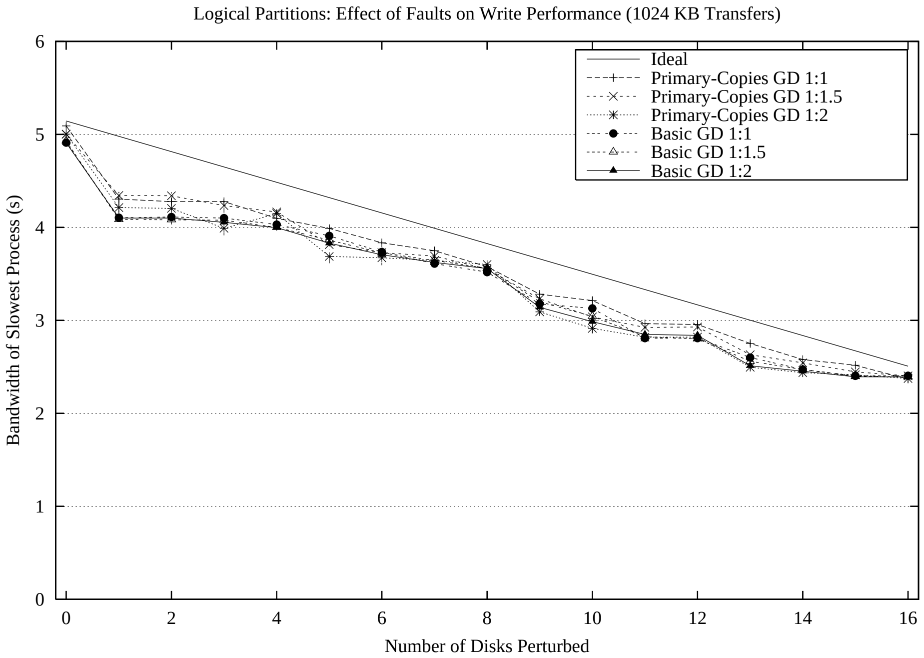


Figure 5: Logical partitioning: effect of faults on performance. The average per-process bandwidth (based on the completion time of the slowest process) is shown. The ratio of data accessed by evenly- and oddly-numbered processes is varied. The block size is fixed at 1024 KB. The number of faults is varied from zero to 16.

and writing, Primary-Copies GD's performance improvement over Basic GD is evident. We also notice somewhat greater variation in the read performance of Primary-Copies GD when the data ratio changes. However, we can still comfortably state that logical partitioning works well with graduated declustering at this transfer size.

5 Discussion

As we have shown, graduated declustering has a great deal of flexibility above and beyond its original implementations. Not surprisingly, there are further enhancements to be made.

One possibility is to adaptively decrease each process's primary-copies *high_watermark* parameter as completion nears. This will allow a larger initial value for more play in the system early on, reducing multiplexing and improving disk utilization. Later on in the program's execution, as progress is made, the reduction in the *high_watermark* parameter enables the system to avoid skew in the key program performance metric: the slowest-process's completion time.

Another possibility is an implementation of graduated declustering for applications with loose read-ordering requirements, such as parallel sorting. In its initial phase, a parallel sorting program repartitions its entire dataset. The range partitioning performed in that step has no dependence on the initial ordering or placement of the dataset. This enables the construction of a graduated-declustering implementation that requires no network transfers of disk data, thus bringing the processor, I/O bus, and network overheads to their bare minimums. (It is still necessary transmit control messages across the network, but these transfers are limited in both size and number.) The key to such a design is recognizing that the sort program does not care which bytes of a parallel file are read by a given processes so long as each byte is read by some process. In this case, graduated declustering can influence the progress of a process by adjusting the amount of data that it must read from the local disk, rather than by adjusting how quickly its must read data for that process.

As a final thought, we consider the effect of current hardware trends on the effectiveness and the importance of graduated declustering. 1.2 GHz processors and 50 MB/second disk drives are currently available, and the performance of those components, as well as that of networks, will continue to improve in the future. Not only is disk performance growing rapidly, but so is disk capacity (73 GB disks are now available), and capacity has been increasing more rapidly than performance. With more data to read from each disk, transfer bandwidth will continue to be a bottleneck resource in clusters that hope to achieve good performance for I/O-intensive applications. As a result, graduated declustering should continue to be a useful mechanism for achieving consistently-good performance. Moreover, rapidly-growing disk capacity increases the attractiveness of replication, and in turn graduated declustering. Processor speed increases should continue to easily handle the overhead of graduated declustering. And while networks may bog down under graduated declustering if disk transfer performance continues to improve so rapidly, the primary-copies mechanism helps to address that concern.

6 Conclusion

We have presented three enhancements to graduated declustering, a mechanism for use in the construction of I/O-intensive parallel programs that are robust to disk performance faults. These three enhancements are write support, primary copies, and logical partitioning. We described them, and gave details of their design, implementation, and benefits. We evaluated their performance, finding it to be good. Notably, the primary-‘copies enhancement offers consistently better performance than the original graduated declustering designs. Finally, we presented ideas for further enhancements of graduated declustering and discussed the implications of current hardware trends.

References

[1] Andrea C. Arpaci-Dusseau, Remzi Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, May 1997.

[2] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *Proceedings of the Second SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[3] Remzi H. Arpaci-Dusseau. *Performance Availability for Networks of Workstations.* PhD thesis, University of California, Berkeley, 1999.

[4] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David 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*, February 1998.

[5] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, Katherine Yelick. Cluster I/O with River: Making the Fast Case Common. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[6] Jim Gray, Microsoft Corporation, 301 Howard St., #830, San Francisco, CA 94105. http://research.microsoft.com/barc/SortBenchmark/

[7] Hui-I Hsiao and David J. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proceedings of the Sixth International Conference on Data Engineering*, February 1990.

[8] Transaction Processing Performance Council, 650 N. Winchester Blvd., Suite 1, San Jose, CA 95128. http://www.tpc.org/