

# When One Pipeline Is Not Enough

Dartmouth Computer Science Technical Report TR2007-596

Thomas H. Cormen

Elena Riccio Davidson\*

Priya Natarajan

H5 Technologies

Dartmouth College Department of Computer Science

laneyd@gmail.com

{thc, priya}@cs.dartmouth.edu

## Abstract

Pipelines that operate on buffers often work well to mitigate the high latency inherent in interprocessor communication and in accessing data on disk. Running a single pipeline on each node works well when each pipeline stage consumes and produces data at the same rate. If a stage might consume data faster or slower than it produces data, a single pipeline becomes unwieldy.

We describe how we have extended the FG programming environment to support multiple pipelines in two forms. When a node might send and receive data at different rates during interprocessor communication, we use disjoint pipelines that send and receive on each node. When a node consumes and produces data from different streams on the node, we use multiple pipelines that intersect at a particular stage. Experimental results for two out-of-core sorting algorithms—one based on columnsort and the other a distribution-based sort—demonstrate the value of multiple pipelines.

---

\*Work performed while Elena Riccio Davidson was at Dartmouth College.

# 1 Introduction

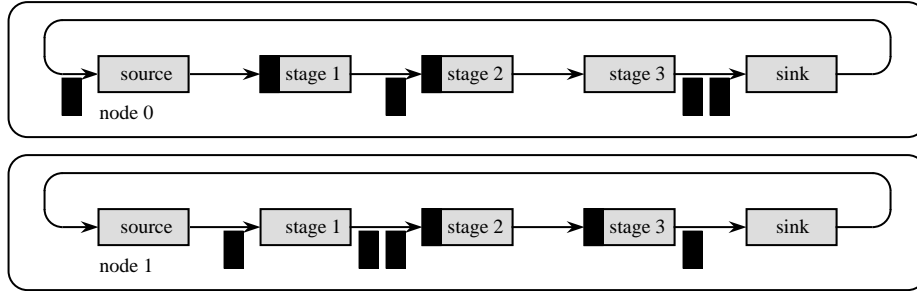
The FG programming environment [3, 4, 5, 6, 7, 8] uses pipelines to mitigate the high latency inherent in interprocessor communication and in accessing the outer levels of the memory hierarchy. For example, in out-of-core programs, the dataset is so large that it exceeds the size of the main memory, and so it resides on one or more disks. Several algorithms in the literature for solving out-of-core problems (see the survey by Vitter [12]) have similar structures. They make multiple passes over the data, where each pass might entail high-latency operations such as disk I/O or interprocessor communication. These programs move data in blocks in order to amortize the high cost of transferring data. FG's developers claim that it makes such programs smaller, faster, and quicker to develop [4, 5].

FG uses software pipelines in order to make it easy to overlap operations. While high-latency operations are in progress, CPUs are often free to perform other functions. With FG's pipeline structure, one stage of a pipeline can perform a high-latency operation on one block of data while other pipeline stages work on different blocks of data, much like how hardware pipelines work. FG advances buffers, corresponding to blocks, through the pipeline.

**FG background.** An FG pipeline is composed of stages. By mapping each stage to its own thread, FG allows the stages to run asynchronously so that stages performing high-latency operations can overlap their work with other stages. Because buffers correspond to blocks for transferring data, the buffer size typically equals the block size. Each buffer is described by a small object called a *thumbnail*, which contains, among other information, a pointer to the buffer. FG places a queue of thumbnails between each pair of consecutive stages in the pipeline, so that a stage *conveys* a buffer to its successor by placing the buffer's thumbnail into the queue between the stage and its successor. The stage can then immediately *accept* the next buffer from the queue between it and its predecessor, and then it can start working on the buffer. Queues actually contain pointers to thumbnails, and each thumbnail contains a pointer to its corresponding buffer. In this way, no copying need occur as a buffer progresses through the pipeline.

FG adds two stages to every pipeline: a source stage at the start and a sink stage at the end. The source stage injects buffers into the pipeline by conveying them to the first stage following the source. The sink recycles each buffer that reaches it back to the source stage, so that only a small pool containing a fixed number of buffers needs to be allocated. The sink also shuts down the pipeline when the last buffer (the *caboose*) reaches it.

The typical parallel program that uses FG runs one copy of a single pipeline on each node in a cluster. See Figure 1

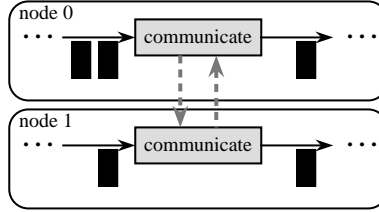


**Figure 1:** A standard FG pipeline comprising a source, a sink, and three other stages running on two nodes. Each black rectangle represents a buffer. Where a buffer appears inside a stage, the stage is currently working on that buffer. Buffers in queues appear below the arrows between stages. The arrow from the sink to the source represents how buffers are recycled.

for an example of a standard FG pipeline running on two nodes. Note that in Figure 1, and in all the figures that follow, pipelines are shown on two nodes only for illustration purposes. In general, we can have any number of nodes, each running its own copy of the pipeline.

**Our extensions to FG.** In some programs, a given pipeline stage accepts and conveys buffers at the same rate, even when the stage performs interprocessor communication. Consider, for example, the stage shown in Figure 2. This stage repeatedly accepts a buffer from its predecessor stage as input, performs interprocessor communication, and conveys the buffer to its successor stage in the pipeline. The communication stage sends data from the buffer and receives data into the buffer. If the amount of data this stage sends equals the amount of data it receives every time, then the rate at which data enters the stage from its predecessor stage equals the rate at which data exits the stage to its successor stage. In this case, we can convey to the successor the same buffer that the stage accepted from its predecessor.

Sometimes, however, a given pipeline stage needs to accept and convey buffers at different rates. For example, if the communication stage in Figure 2 sends more data from a buffer than it receives into the buffer, then this stage will need to convey buffers to its successor at a slower rate than it accepts buffers from its predecessor. Conceptually, buffers begin to pile up within the stage. Conversely, if this stage sends less data than it receives, then this stage will need to convey buffers at a faster rate than it accepts them. Conceptually, this stage needs to acquire new buffers in



**Figure 2:** A pipeline stage that performs interprocessor communication. A copy of the pipeline runs on each node, with individual buffers represented by black rectangles. The source, sink, and other stages involved in the pipeline are not shown. The stage repeatedly accepts a buffer from its predecessor stage in the pipeline, sends data from that buffer to its counterpart stages in other nodes (communication is indicated by dashed gray arrows), receives data from its counterparts into the buffer, and then sends the buffer to its successor stage in the pipeline.

---

order to have some to convey. In either case, it would be difficult to convey to the successor stage the same buffer that entered.

This paper describes how we extended FG to support pipelines in which stages accept and convey buffers at different rates. We augmented FG in two ways: support for multiple disjoint pipelines and support for multiple pipelines that intersect at a particular stage. These extensions to FG arose from the design requirements of an out-of-core distribution-based sorting algorithm for a cluster. In a prior sorting algorithm for a cluster [2], which was based on out-of-core columnsort, all interprocessor communication sent and received the same amount of data every time, and so the single-pipeline model sufficed. In contrast, in the distribution phase of a distribution-based sort, each node can send more or less data than it receives during interprocessor communication. In the merging phase, a distribution-based sort consumes data from sorted runs at different rates, which also differ from the rate at which the merging phase produces sorted data.

Experimental results on a cluster show that our distribution-based sorting algorithm runs faster than the columnsort-based algorithm in most cases. The columnsort-based algorithm has the advantage that its disk-I/O and communication patterns are oblivious to the data values, and so all disk-I/O and communication operations are predetermined. In the distribution-based sort, on the other hand, the exact disk-I/O and communication operations depend heavily on the

data being sorted. The distribution-based sort is able to run faster because it performs fewer disk-I/O operations. Both algorithms make multiple passes over the data, where each *pass* reads each record to be sorted once from one of the disks in the cluster and writes each record once to one of the cluster’s disks. The distribution-based sort requires only two passes over the data (plus a little bit more to select splitter values), whereas the column-sort-based algorithm makes three passes.

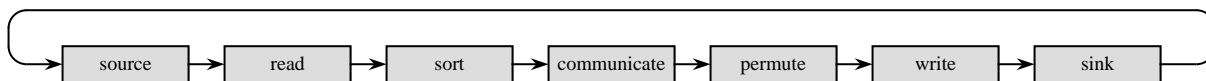
Therefore, we see that with our extensions, FG successfully hides latency when stages accept and convey buffers at different rates and even when the sequence of high-latency operations is dynamically determined.

The remainder of this paper is organized as follows. Section 2 summarizes the column-sort-based algorithm, which uses only one pipeline per node. Section 3 shows how we extended FG to support multiple disjoint pipelines and multiple intersecting pipelines. The extensions to FG provide the structure for the implementation of the distribution-based sort described in Section 4. Section 5 presents experimental results on a cluster, showing that with our extensions to FG, the distribution-based sort fares well compared with the column-sort-based method. Finally, Section 6 offers some concluding remarks.

## 2 Out-of-core column-sort

Without our extensions to FG, all programs that use FG are restricted to only a single pipeline per node. FG allows several variations on simple, linear pipelines—such as fork-join constructs and even arbitrary DAG configurations—but all of these additional features work only within a context of one pipeline on each node. We implemented in FG an out-of-core sorting algorithm, described in [2] and based on Leighton’s column-sort algorithm [10], using only a single, linear pipeline on each node.

Our implementation of out-of-core column-sort mirrors the treatment in [2], which we summarize here. Column-sort sorts  $N$  records, which are considered to be in an  $r \times s$  matrix, where  $N = rs$ . There are additional restrictions on the height  $r$  and width  $s$  of the matrix:  $r$  must be even,  $s$  must divide  $r$ , and  $r \geq 2s^2$ . At completion, the matrix is sorted in column-major order. Column-sort takes eight steps, where each odd-numbered step sorts every column individually. Each even-numbered step performs a specific, fixed permutation on the matrix. Step 2 transposes the matrix and reshapes it back into an  $r \times s$  shape. Step 4 performs the inverse permutation of step 2. Step 6 shifts each



**Figure 3:** The pipeline structure of the four-pass version of out-of-core columnsort. Each node runs a copy of this pipeline during each pass. Buffers are not shown. The exact operation of the sort, communicate, and permute stages varies among the four passes.

---

column down by  $r/2$  positions: it moves the bottom half of each column to the top half of the next column, and the top half of each column into the bottom half of the same column. The top half of the leftmost column is filled with  $-\infty$  values, and a new rightmost column receives the bottom half of the last column in its top half, with the bottom half filled with  $\infty$  values. Step 8 performs the inverse permutation of step 6.

A relatively simple four-pass implementation of out-of-core columnsort groups together steps 1 and 2 into a first pass, steps 3 and 4 into a second pass, steps 5 and 6 into a third pass, and steps 7 and 8 into a fourth pass. In each pass, each node of the cluster runs a copy of the same pipeline. Figure 3 shows the pipeline’s structure, which is similar across all four passes. As each buffer traverses the pipeline, the read stage reads a column of the matrix from the disk into the buffer, and the sort stage accomplishes the appropriate odd-numbered step. The communicate and permute stages accomplish the appropriate even-numbered step, and the write stage writes a column to the disk. The exact nature of the sort, communicate, and permute stages varies from pass to pass, according to known characteristics of the columns entering the pass and to the permutation performed in the even-numbered columnsort step.

In order to achieve a three-pass implementation, we can combine steps 5–8 of columnsort, which map to the latter two passes in the four-pass implementation, into a single pass. The key observation is that in the four-pass implementation, the communicate, permute, and write stages of the third pass, together with the read stage of the fourth pass, just shift each column down by  $r/2$  positions, as described above. By replacing these four stages by a single communicate stage, we can eliminate one pass.

The three-pass implementation, which we call “csort” from here on, has two important properties. First, as mentioned in Section 1, the disk-I/O and communication patterns are predetermined. That is because csort is oblivious to the data values, except for steps that sort internally within each node. Thus, it is relatively easy to structure the implementation to overlap disk I/O, communication, and computation. Second, in every communication step, each

node receives exactly as much data as it sends. That is because the communication steps correspond to highly regular permutations such as transposing a matrix or sending half of each node's data to the next node. Indeed, all interprocessor communication in the implementation is via the MPI calls `MPI_Sendrecv_replace`, `MPI_Alltoall`, and matching pairs of `MPI_Send` and `MPI_Recv` with equal data sizes specified.

The `csort` algorithm resulted from extensive engineering, using only off-the-shelf software, applied to Leighton's columnsort algorithm. Mechanisms such as active messages and memory-mapped I/O were specifically avoided.

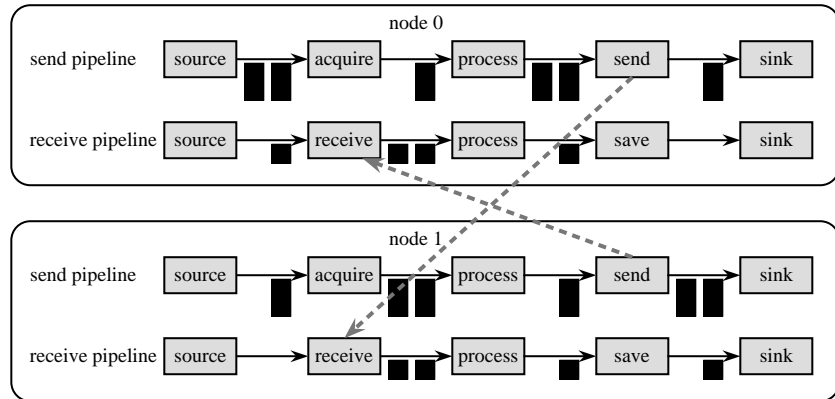
### 3 FG support for multiple pipelines

We have extended FG to support multiple pipelines that are disjoint and multiple pipelines that intersect. This section discusses these new features in more detail.

**FG basics.** FG transforms a series of programmer-defined C/C++ functions into one or more pipelines of asynchronous stages that operate on buffers. The programmer writes a straightforward function that implements each pipeline stage, containing only synchronous calls. FG runs the stages asynchronously, via calls to standard POSIX pthreads functions [9], and it manages the buffers. An early paper on FG [3] shows that programs that use FG can be as fast as, or even faster than, hand-tuned programs that call pthreads functions directly.

FG defines classes for its basic building blocks: stages, threads, and pipelines. To create a single pipeline, the programmer first creates an object for each stage and, optionally, an object for each thread. After associating a function with each stage object, the programmer then creates an array of pointers to stage objects and passes that array to a function that creates a pipeline with these stages, in order. After creating the pipeline, the programmer sets its attributes (e.g., the number and size of the buffers), and finally fixes the pipeline's final configuration and runs it.

As mentioned in Section 1, FG adds source and sink stages to every pipeline so that buffers can be reused. The buffer queues that FG maintains between successive stages allow for stages to accept and convey buffers as soon as they are ready to do so. In this way, each stage can convey a buffer to its successor regardless of whether the successor is ready to accept the buffer at that time.



**Figure 4:** Disjoint pipelines running in each node, shown for just two nodes. The send pipeline has a send stage, which sends data from buffers to various nodes. The receive pipeline has a receive stage, which receives data sent by the various send stages. Within a node, each pipeline progresses at its own pace, which is determined in part by the amount of data sent and received. The number of buffers and their sizes (represented by different-sized black rectangles) can differ between the two pipelines on each node. The sink recycling buffers back to the source is not shown explicitly in this figure.

**Managing multiple disjoint pipelines.** To support multiple pipelines, we added a pipeline-manager class to FG. Now the programmer may create several pipeline objects and then an array of pointers to these pipeline objects. The programmer then creates a single pipeline-manager object, passing the array of pipeline-object pointers to the constructor, and calls a pipeline-manager function to fix and run all the pipelines belonging to that manager.

When the multiple pipelines are disjoint, each stage belongs to one and only one pipeline. Because the pipelines are linear, each stage has a unique predecessor and a unique successor. Thus, when a stage accepts or conveys a buffer, it is always unambiguous which stage it is accepting from or conveying to.

Figure 4 abstracts how we use multiple disjoint pipelines to solve the problem of a stage sending more or less data than it receives, as discussed in Section 1. Each node has two pipelines. The *send pipeline* acquires data into a buffer, processes the buffer, and then sends the data in the buffer to the various nodes of the cluster. The *receive pipeline* receives into a buffer the data sent by each of the send stages running on the nodes, processes the buffer, and then saves the data in the buffer. The details of the acquire, process, and save stages are unimportant here. What matters is

that the pace at which buffers progress through the two pipelines in the same node may differ, according to the rate at which each node sends data and the rate at which each node receives data.

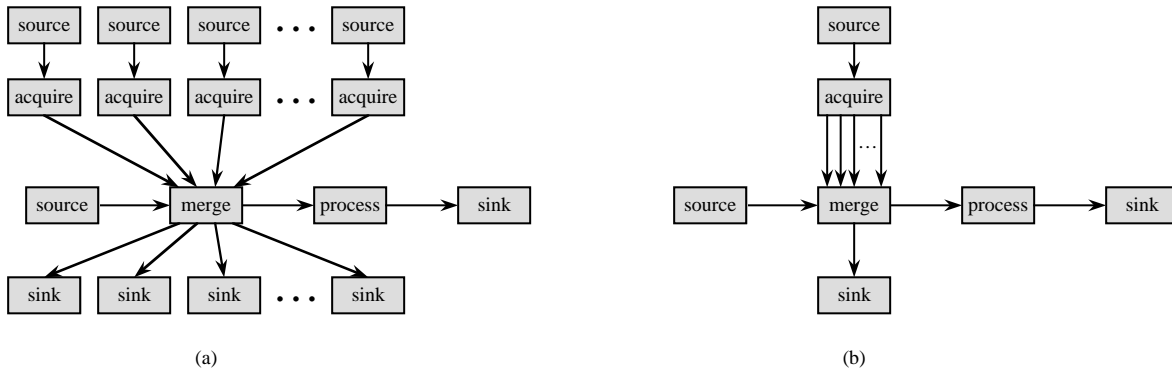
The only way in which the two pipelines interact is that one may send data via interprocessor communication to the other. Each of the two pipelines has its own source and sink, its own number of buffers, and its own buffer size. Although data may move from send pipelines to receive pipelines via interprocessor communication, each buffer is tied to a specific pipeline.

**Managing multiple intersecting pipelines.** The situation for multiple pipelines that intersect at a *common stage* is more complicated than for disjoint pipelines. As with disjoint pipelines, the programmer creates multiple pipeline objects and a single pipeline-manager object. We have extended FG so that if it determines that a particular stage object belongs to more than one pipeline, then it treats these pipelines as intersecting at that common stage. FG creates only one thread for the common stage, which can accept buffers from any of its predecessor stages on any of the pipelines it belongs to.

Because the common stage has multiple predecessors, when accepting a buffer, it must specify which pipeline to accept from. The function in the stage class that accepts buffers is overloaded to allow the programmer to specify the pipeline. Every buffer is tied to a particular pipeline; buffers cannot jump from one pipeline to another. Each buffer's thumbnail contains a pipeline identifier. Thus, when a common stage conveys a buffer, it does not specify which pipeline to convey the buffer along: this information is gleaned from the buffer's thumbnail.

Figure 5(a) abstracts how we use multiple intersecting pipelines on a node to merge several small, sorted runs of data into a single large, sorted output sequence. Input comes into the merge stage—the common stage—from the vertical pipelines, and output goes from the merge stage into the horizontal pipeline. Each vertical pipeline has a stage that acquires a buffer containing a section of an individual sorted run, feeding into the merge stage. The merge stage accepts empty buffers from the horizontal pipeline's source stage, filling them with data that came in along the vertical pipelines, and once it has filled a buffer, it sends the buffer along the horizontal pipeline for further processing.

As with multiple disjoint pipelines, pipelines that intersect can have differing numbers and sizes of buffers. For example, in Figure 5(a) the buffer sizes differ between the vertical and horizontal pipelines. Buffers in the vertical pipelines are relatively small, since there might be many of them. Although the sorted runs in the vertical pipelines are



**Figure 5:** Multiple intersecting pipelines on a given node. These pipelines merge several small, sorted runs of data traveling the vertical pipelines into a single, large sorted sequence traveling the horizontal pipeline. To reduce clutter, buffers and arrows connecting the respective source and sink stages are not shown. The merge stage, which is the common stage in the intersecting pipelines, accepts small buffers from the vertical pipelines and merges them into larger buffers along the horizontal pipeline, where the buffers undergo further processing. **(a)** The conceptual view with multiple vertical pipelines. **(b)** How the vertical pipelines are implemented when the acquire stages are designated as virtual. Each box represents a single thread. All the acquire stages in the vertical pipelines share a common thread, as do all the source stages and all the sink stages. Each arrow is associated with a buffer queue, so that only one queue feeds into the acquire and sink stages, but several queues feed into the common merge stage. We omit the feedback arrows from the sink to source stages, but each source stage has only one incoming queue.

small compared to the output sequence, each sorted run is many times the size of the vertical pipeline buffers. There is only one horizontal pipeline, however, and so its buffers can be much larger than those in the vertical pipelines.

The merge stage operates as follows. It repeatedly chooses the smallest value not yet chosen from any of the buffers that it has accepted along a vertical pipeline. It then copies this value into the next available position in the output buffer that it has accepted along the horizontal pipeline. Once an output buffer fills, the merge stage conveys it along the horizontal pipeline for further processing, and then the merge stage accepts a new, empty buffer from the horizontal pipeline's source stage. Whenever the merge stage has consumed all the values from an input buffer along a vertical pipeline, it conveys this spent buffer to the sink along that particular vertical pipeline, where it will be recycled back to the source. The merge stage then accepts the next buffer from the same sorted run along the same vertical

pipeline. Of course, once a vertical pipeline sends its last buffer (i.e., its caboose), the merge stage should no longer try to accept a buffer along that pipeline, nor should it consider values from that pipeline's run when making decisions about merging.

Each of the pipelines operates at its own rate. The merge stage consumes data from each vertical pipeline buffer according to how the merging proceeds, and it fills each horizontal pipeline buffer at a rate that is likely to be different from the rate at which it consumes any of the vertical pipeline buffers.

**Virtual stages and virtual pipelines.** Observe that in Figure 5(a), each of the vertical pipelines has the same structure. There could easily be hundreds of such vertical pipelines on a given node. Because FG normally creates one thread per stage, including the source and sink, FG would try to create hundreds or even thousands of threads per node when executing these intersecting pipelines.

Most current systems cannot handle hundreds of threads. They either grind to a halt or simply disallow more threads to be created after reaching a limit.

We overcame this problem with *virtual stages*. The programmer can designate multiple, identical stages in separate pipelines as virtual. Instead of creating one thread for each of these stages, FG creates a thread for only one of the stages. The remainder of the corresponding stages share this thread. Pipelines containing any such stage are *virtual pipelines*.

In the example of Figure 5(a), the programmer could designate the acquire stages as virtual. Figure 5(b) shows the result: if there are  $k$  vertical pipelines, FG will create only one thread for the acquire stages rather than  $k$  threads.

We extended FG to economize in other ways with virtual pipelines. Whenever FG detects that pipelines are virtual, it automatically makes the source and sink stages for these pipelines virtual as well. Furthermore, as Figure 5(b) shows, if  $k$  identical stages are designated as virtual, then instead of creating  $k$  individual queues feeding into these stages, FG creates just one queue.

How do the stages of virtual pipelines differ from a common stage that belongs to multiple pipelines? For example, let us consider the acquire stages and the merge stage of Figure 5(b). The acquire stages all have the same stage function, which accepts a buffer, places data into the buffer, and conveys the buffer. The acquire function accepts any buffer that is ready, regardless of which pipeline that buffer belongs to. Because they are virtual, all of the acquire

stages share one buffer queue. The first buffer to be ready in that queue could be from any one of the virtual pipelines. The acquire stage processes the buffers in the order in which they arrive in the queue.

The merge stage, on the other hand, accepts a buffer from a specific pipeline. If there are  $k$  vertical pipelines, all virtual, then  $k + 1$  buffer queues feed into the merge stage: one from each of the  $k$  vertical pipelines, and one from the horizontal pipeline. Each time the merge stage accepts a buffer, it must specify which pipeline to accept from. FG waits until a buffer appears in the queue on that pipeline, regardless of how many buffers are available in the other  $k$  queues.

## 4 Out-of-core distribution sort

This section describes how we designed and implemented an out-of-core distribution sort using the extensions described in Section 3. We call this program “dsort.” As mentioned in Section 1, dsort entails two passes over the data, following a preprocessing phase. The preprocessing phase selects *splitters*, which define how the first pass partitions the data among the nodes. After the first pass, each node contains several sorted runs. The second pass merges the sorted runs to create a single sorted sequence and then permutes the sorted records across the cluster to perform load balancing and to create striped output.

By “striped output,” we mean that it appears in the order defined in the Parallel Disk Model [13]. The records reside in fixed-size blocks, which are assigned in round-robin order to the disks in the cluster. Assuming that each node has one disk and that each block contains  $B$  records, therefore, the first  $B$  records reside on node 0’s disk, the next  $B$  records on node 1’s disk, and so on. If the cluster has  $P$  nodes, then a block on node  $P - 1$  is followed by a block on node 0. Both dsort and csort create striped output.

**Selecting splitters.** The first pass partitions the  $N$  records among the  $P$  nodes such that each record in node  $i$  has a key less than or equal to the keys of all records in node  $i + 1$ , for  $i = 0, 1, \dots, P - 2$ . In order to decide which processor each record belongs to, we need to select a set of  $P - 1$  key values, known as *splitters*. Splitters are the multiway analogue of the pivot value when partitioning during quicksort. Ideally, the splitters should partition the  $N$  records into  $P$  partitions of  $N/P$  records each. In practice, we do not achieve such perfectly balanced partitions, but we can get close almost all the time.

The preprocessing step finds the splitters using the technique of *oversampling*, as done by Blelloch et al. [1] and by Seshadri and Naughton [11]. Each node selects a uniform random sample of  $s$  *candidates* from among its  $N/P$  records, where the parameter  $s$  is known as the *oversampling ratio*. Each node then sends its set of candidates to node 0, which gathers the  $sP$  candidates and sorts them. The  $P - 1$  final splitters are the records at ranks  $s, 2s, 3s, \dots, (P - 1)s$  in the sorted list of candidates. Node 0 then broadcasts the  $P - 1$  splitters to all other nodes.

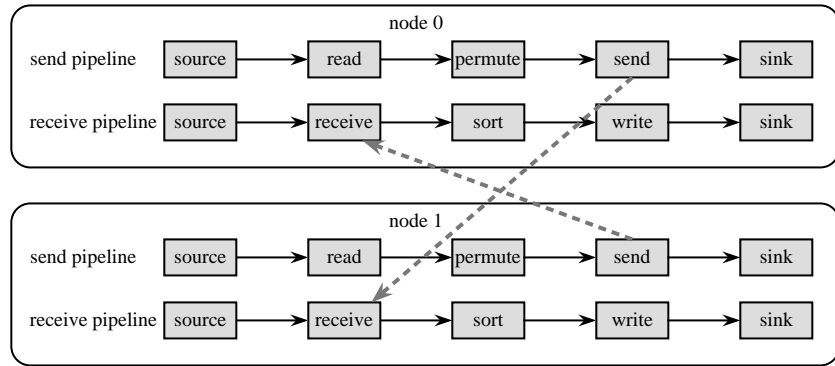
This method of selecting splitters almost always works well if all the input keys are distinct, but it can lead to unbalanced partition sizes if many keys happen to have the same value. In order to circumvent this problem, we extend each candidate record's key with two additional fields to break ties: the number of the node that the candidate comes from (0 to  $P - 1$ ), and the candidate's offset within the node (0 to  $N/P - 1$ ). With these extensions, all keys are unique. When deciding where to send each record, the first pass of dsort extends each record's key in the same manner and compares each record with the splitters using the extended keys. Only the original keys, and not the extended parts, are used afterward.

If the oversampling ratio  $s$  is large enough, then the partitions are reasonably well balanced with very high probability. We let  $\rho$  denote the ratio between the maximum partition size and the average partition size  $N/P$ . Ideally, we would have  $\rho = 1$ . Seshadri and Naughton [11] show that

$$\Pr \{ \rho > \alpha \} \leq P \alpha^s \left( \frac{P - \alpha}{P - 1} \right)^{s(P-1)}.$$

In our experiments, we used an oversampling ratio of  $s = 1000$  and  $P = 16$  nodes. With  $N = 2^{30}$ , the probability that some partition size exceeds the average by more than  $\alpha$  is less than  $1/N$  for all  $\alpha \geq 1.23$ , and it is less than  $1/N^2$  for all  $\alpha \geq 1.32$ . In fact, over several dozen runs, we never observed a value of  $\rho$  above 1.10.

**Pass 1: Partitioning and distribution.** Pass 1 partitions and distributes the records among the nodes according to the splitters that have been selected and broadcast. Figure 6 shows the pipeline structure of pass 1. Distribution requires interprocessor communication, and the number of records that a node sends at any one time almost certainly differs from how many records it receives. Hence, we use separate send and receive pipelines, as in Figure 4, but with the following stages renamed: acquire becomes read, process in the send pipeline becomes permute, process in the receive pipeline becomes sort, and save becomes write. In each case, we have just made the action more specific. Buffer sizes in the send and receive pipelines are equal.



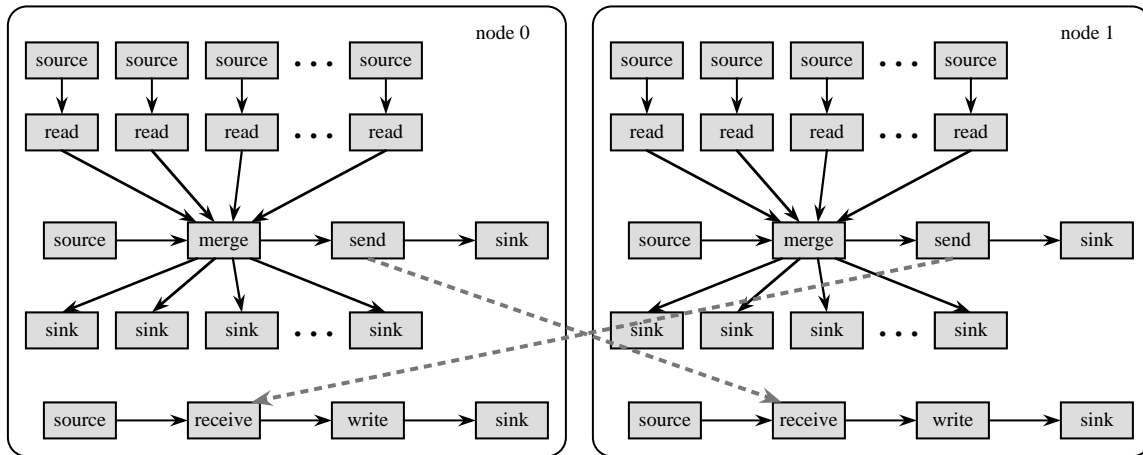
**Figure 6:** The pipeline structure of pass 1 of dsort, shown for two nodes. The structure is similar to that of Figure 4. Buffers and sink-to-source connections are not shown.

The send pipeline works as follows. The read stage reads a buffer of records from the disk, which it then conveys to the permute stage. The permute stage uses the splitters and extended keys to rearrange the records in the buffer so that all records belonging to the same partition are contiguous; it uses an auxiliary buffer so that the permutation need not be performed in-place. The buffer then travels to the send stage, which doles out the records of each partition to their target node.

In the receive pipeline, the receive stage repeatedly receives records sent by send stages into a temporary buffer. It copies received records into a pipeline buffer until the pipeline buffer fills, at which time it conveys the pipeline buffer to the sort stage and accepts a new, empty pipeline buffer. The sort stage simply sorts the records (according to the original, non-extended keys) using an auxiliary buffer, and conveys them to the write stage, which writes the buffer to disk. Each buffer written contains a sorted run.

The send stage actually sends three types of messages to receive stages, and each receive stage keeps track of its communication with the send stage of every node. One type of message simply tells the receive stage the number of records about to be sent, and another type of message contains the records. The third type of message tells the receiving node that the caboose has gone through this node's send pipeline, and hence this sending node is finished sending records in pass 1. Once each receive stage receives  $P$  such messages, it can shut down its pipeline.

It is important to assign the right number of buffers to each pipeline. Too few and stages may starve for buffers,



**Figure 7:** The pipeline structure of pass 2 of dsort, shown for two nodes. The structure is a combination of those in Figures 4 and 5. Buffers and sink-to-source connections are not shown.

thereby limiting how much we can overlap disk I/O, communication, and computation. Too many, and buffers may spend too much time sitting in queues between stages, waiting to be processed. We found that three pipeline buffers and one auxiliary buffer per pipeline worked best.

**Pass 2: Merging, load balancing, and striping.** At the end of pass 1, we have sorted runs of records, which we need to merge into longer sorted output sequences. If that was all we had to do, then the structure in Figure 5 would suffice, with the stage labeled “process” writing the sorted sequences out to local disks.

We need to do more, however. Because the partition sizes created during pass 1 are not necessarily all equal to  $N/P$ , we need to load balance the records across nodes. Furthermore, we need to stripe the output when writing to disk. We omit the details of how we compute which node each record goes to after merging, and we focus instead on how we perform the interprocessor communication. Figure 7 shows how. After the merge stage fills a buffer, that buffer travels to a send stage, which disperses the records in the buffer to various nodes. Just as in pass 1, the rate at which each node sends records differs from the rate at which the node receives records. Hence, we use separate pipelines for sending and receiving. As in pass 1, the pipeline that receives records copies them into a pipeline buffer

until that pipeline buffer fills. The buffer then travels to the write stage, which simply writes out the sorted block. The send and receive stages coordinate using the same three types of messages as in pass 1.

We might use two buffer sizes for the three types of pipelines. Both of the pipelines drawn horizontally in Figure 7—that is, the pipeline containing the send stage and the pipeline containing the receive stage—use buffers equal to the block size for striping. The vertical pipelines, because there are so many of them (we have approximately 32 of them when sorting 64 gigabytes on 16 nodes using 1 gigabyte of RAM per node), may require buffer sizes smaller than those used in the horizontal pipelines.

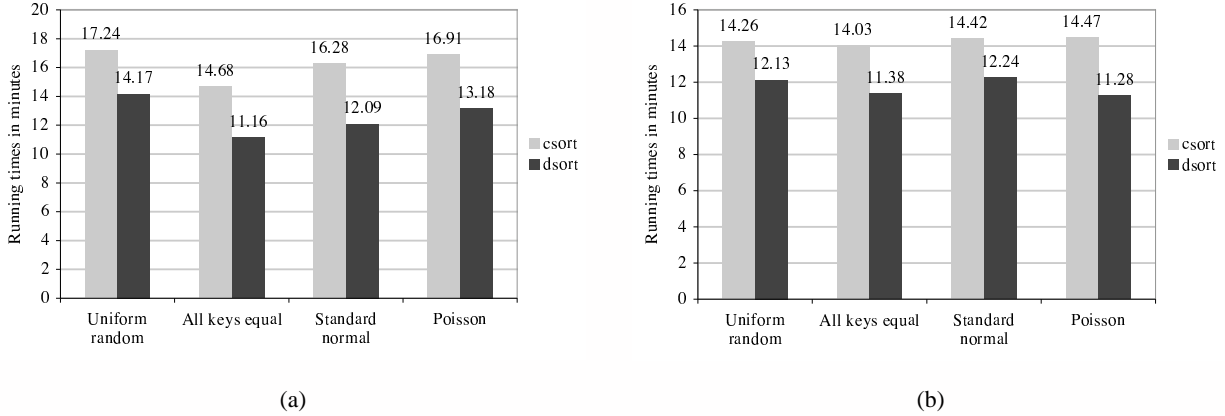
As in pass 1, we need to assign the right number of buffers to each pipeline. We want each vertical pipeline to always have a buffer ready to feed the merge stage; the horizontal send pipeline to always have a buffer ready to merge into and another buffer being scattered to the various nodes; and the horizontal receive pipeline to always have a buffer in which to receive data and another buffer being written to disk. We found that two buffers per vertical pipeline, two buffers per horizontal send pipeline, and 32 buffers per horizontal receive pipeline (two buffers per sending node, at 16 nodes) worked best.

## 5 Experimental results

In this section, we summarize our experiments with `dsort` on a cluster. We compare `dsort`'s performance to that of `csort` using FG. (We found, incidentally, that `csort` using FG is slightly faster than the non-FG version reported in [2].)

The cluster is a Beowulf-class system in which we used 16 nodes. Each node has two 2.8-GHz Intel Xeon processors, 4 GB of RAM, and an Ultra-320 SCSI hard drive. The nodes run RedHat Linux 9.0 and are connected by a 2-Gb/sec Myrinet network. We use the C stdio interface for disk I/O, calling `fread_unlocked` and `fwrite_unlocked`. We also call `fread`, `fwrite`, `flock`, and `funlock` because our programs are multi-threaded, with multiple threads reading from the same file in pass 2 of `dsort`. We use the ChaMPIon/Pro for interprocessor communication because this particular MPI implementation is thread safe.

Each experiment sorts a total of 64 gigabytes of data, distributed evenly among the 16 nodes. We ran experiments with two different record sizes: 16 bytes for a total of 4 gigarecords, and 64 bytes for a total of 1 gigarecord. All



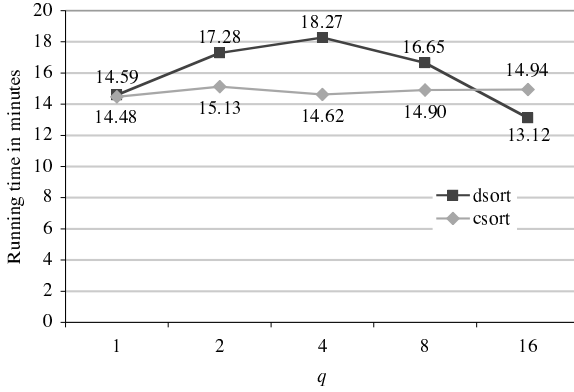
**Figure 8:** Running times of dsort and csort on various input distributions of 64 GB of data on 16 nodes. **(a)** 16-byte records. **(b)** 64-byte records.

results reported here are for the best choices of buffer sizes. Each result represents the average of three runs; running times varied only slightly within each group of three.

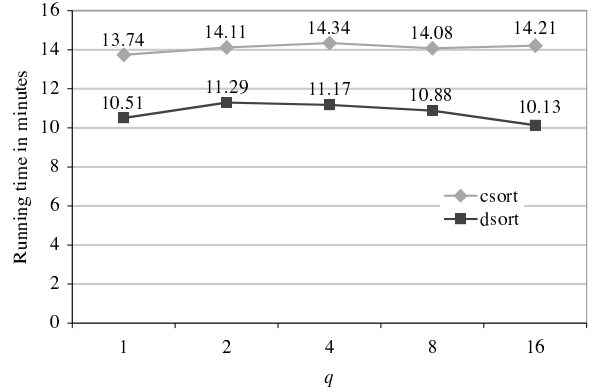
We compared dsort and csort with various key distributions: uniform random, all keys equal (but extended as described in Section 4), standard normal, Poisson with  $\lambda = 1$ , and a distribution designed to elicit poor performance in dsort. In the latter distribution, we designed the keys so that communication is unbalanced in pass 1 of dsort. Each time communication occurs in pass 1, almost all the records go to only  $q$  out of the 16 nodes, with the remaining  $16 - q$  nodes receiving very few records. Which nodes comprise the set of  $q$  varies over time, so that eventually each node receives its full complement of records. For low values of  $q$ , communication is highly unbalanced. We designed input distributions for  $q = 1, 2, 4, 8$ , and 16.

Figure 8 shows the results for the uniform random (averaged over three distinct input datasets), all-keys-equal, standard normal, and Poisson distributions. In each case, dsort beat csort, taking time in the range 74.26%–85.06% of csort’s time.

As Figure 9 shows, the distributions designed to foil dsort did not always succeed. Indeed, for 64-byte records, dsort beat csort by a comfortable margin for all values of  $q$ . For 16-byte records, however, dsort was slower for  $q = 1, 2, 4$ , and 8 and faster only for  $q = 16$ . These results differ greatly from those for the versions of dsort and csort that



(a)



(b)

**Figure 9:** Running times of dsort and csort for 64-GB datasets designed as bad inputs to dsort. In each communication of pass 1, almost all the records go to only  $q$  out of the 16 nodes. **(a)** 16-byte records. **(b)** 64-byte records.

do not use FG in [2]. For 64-byte records and without FG, csort beat dsort for  $q = 1, 2,$  and  $4,$  and dsort was faster for  $q = 8$  and  $16.$  Without FG, dsort took over twice as long as csort for  $q = 1,$  but with FG, the difference is under 1%.

## 6 Conclusion

When we started this project, we expected results in line with those in [2], in which csort prevailed because it is oblivious to the key values (except for the internal sorting steps). We thought that because the disk-I/O and communication patterns of dsort depend so heavily on the key values, it would have higher latencies in these operations. Although dsort might take longer to perform disk-I/O and communication, it turned out that the advantage of taking one fewer pass than csort usually prevailed.

It was our extensions to FG that accounted for the surprise. By extending FG to support situations in which data is consumed and produced at different rates, we were able to overlap disk I/O, communication, and computation sufficiently to overcome dsort’s dependence on key values. Multiple disjoint pipelines support communication in which nodes send and receive different amounts of data, and multiple intersecting pipelines support stages that consume data from one or more pipelines and emit data into one or more pipelines at varying rates. Moreover, each stage and each pipeline is fairly simple to program; FG assumes the burden of gluing them together properly.

## References

- [1] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31:135–167, 1998.
- [2] Geeta Chaudhry and Thomas H. Cormen. Oblivious vs. distribution-based sorting: An experimental evaluation. In *13th Annual European Symposium on Algorithms (ESA 2005)*, volume 3669 of *Lecture Notes in Computer Science*, pages 317–328. Springer, October 2005.
- [3] Thomas H. Cormen and Elena Riccio Davidson. FG: A framework generator for hiding latency in parallel programs running on clusters. In *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004)*, pages 137–144, September 2004.
- [4] Elena Riccio Davidson. The FG programming environment: Good and good for you. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, page 115, July 2006. Brief announcement.
- [5] Elena Riccio Davidson. Improving running time and programmer productivity in pipeline-structured applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2007. Poster.
- [6] Elena Riccio Davidson and Thomas H. Cormen. *Asynchronous Buffered Computation Design and Engineering Framework Generator (ABCDEFGF): Tutorial and Reference*. Dartmouth College Department of Computer Science. Available at <http://www.cs.dartmouth.edu/FG/>.
- [7] Elena Riccio Davidson and Thomas H. Cormen. Building on a framework: Using FG for more flexibility and improved performance in parallel programs. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, April 2005. To appear.
- [8] Elena Riccio Davidson and Thomas H. Cormen. The FG programming environment: Reducing source code size for parallel programs running on clusters. In *Second Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2005.

- [9] IEEE. Standard 1003.1-2001, Portable operating system interface. [http://www.unix.org/version3/ieee\\_std.html](http://www.unix.org/version3/ieee_std.html), 2001.
- [10] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.
- [11] S. Seshadri and Jeffrey F. Naughton. Sampling issues in parallel database systems. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *3rd International Conference on Extending Database Technology (EDBT '92)*, volume 580 of *Lecture Notes in Computer Science*, pages 328–343. Springer-Verlag, March 1992.
- [12] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.
- [13] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.