Out-of-Core Distribution Sort in the FG Programming Environment

Priya Natarajan*
Thomas H. Cormen*
Dartmouth College
Department of Computer Science
{priya, thc}@cs.dartmouth.edu

Elena Riccio Strange†
A9.com
laneyd@gmail.com

***Abstract*—We describe the implementation of an out-of-core, distribution-based sorting program on a cluster using FG, a multithreaded programming framework. FG mitigates latency from disk-I/O and interprocessor communication by overlapping such high-latency operations with other operations. It does so by constructing and executing a coarse-grained software pipeline on each node of the cluster, where each stage of the pipeline runs in its own thread. The sorting program distributes data among the nodes to create sorted runs, and then it merges sorted runs on each node. When distributing data, the rates at which a node sends and receives data will differ. When merging sorted runs, each node will consume data from each of its sorted runs at varying rates. Under these conditions, a single pipeline running on each node is unwieldy to program and not necessarily efficient. We describe how we have extended FG to support multiple pipelines on each node in two forms. When a node might send and receive data at different rates during interprocessor communication, we use disjoint pipelines on each node: one pipeline to send and one pipeline to receive. 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 show that by using multiple pipelines, an out-of-core, distribution-based sorting program outperforms an out-of-core sorting program based on columnsort—taking approximately 75%–85% of the time—despite the advantages that the columnsort-based program holds.**

***Keywords*-multithreaded programming framework; out-of-core sorting; latency; FG.**

I. Introduction

In out-of-core problems, 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 [1]) have similar structures. They make multiple passes over the data, where each pass entails high-latency operations such as disk I/O and possibly interprocessor communication. In order to amortize the high cost of transferring data, these programs move data in blocks. In this paper, we consider the platform of a distributed-memory cluster in which each node can run multiple threads.

In order to process massive datasets, we often need to sort them. Various out-of-core sorting methods appear in the literature, and many of these methods are distribution based. Typically, distribution-based algorithms proceed in three or four phases. The first phase samples the input data and chooses $P-1$ splitters, where P is the number of nodes in the cluster. The second phase, given the splitters, partitions the input into P sets $S_0, S_1, \ldots, S_{P-1}$ such that each sort key[1] in S_i is less than or equal to all the sort keys in S_{i+1}; set S_i resides on the disk of node i of the cluster. The third phase sorts each partition, and an optional fourth phase load-balances the output among the P nodes of a cluster. Even in the presence of well selected splitters, a distribution-based algorithm generates I/O and communication patterns that vary depending on the data to be sorted. To improve efficiency, when we implement a distribution-based algorithm, we should overlap I/O, communication, and computation as much as possible. The FG programming framework makes it easy to overlap such operations. In this paper, we describe how we implemented a distribution-based sorting program using FG.

Brief introduction to FG: The FG[2] programming environment [2]–[6] uses software pipelines to mitigate the high latency inherent in interprocessor communication and in accessing the outer levels of the memory hierarchy. As Figure 1 illustrates, FG advances buffers, corresponding to blocks, through the pipeline. While high-latency operations are in progress, CPUs are often free to perform other functions, so that operations may overlap. With FG's pipeline structure, when one stage of a pipeline yields to perform a high-latency operation on one buffer, other pipeline stages can gain the CPU to work on different buffers.

Each stage in an FG pipeline runs in its own thread, and so stages run asynchronously to overlap their work with other stages. When a stage needs a buffer to work on, it *accepts* a buffer from its predecessor in the pipeline. When it is done working on the buffer, the stage *conveys* the buffer to its successor in the pipeline. The typical parallel program that uses FG runs one copy of a single pipeline on each node in a cluster.

Extensions to FG: In some programs, a pipeline stage that performs interprocessor communication accepts and

*This work was supported in part by National Science Foundation Grant IIS-0326155 in collaboration with the University of Connecticut and in part by National Science Foundation Grant EIA-98-02068.

†Work performed while Elena Riccio Strange was at Dartmouth College. Previous work appears under the name Elena Riccio Davidson.

[1]We assume that we are sorting *records*, where each record consists of a *sort key* and possibly some additional data.

[2]FG is short for ABCDEFG, which is in turn short for Asynchronous Buffered Computation Design and Engineering Framework Generator. We pronounce FG as "effigy."

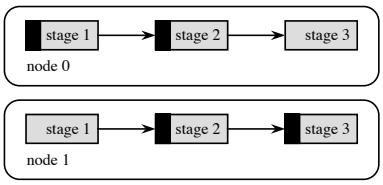

Figure 1. A copy of a software pipeline running on each of two nodes of a cluster. The pipeline consists of three stages and processes data in two buffers, indicated by black rectangles. At the moment shown, the copy on node 0 has buffers in stages 1 and 2, and the copy on node 1 has buffers in stages 2 and 3.

conveys buffers at the same rate. Such a 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 communication is *balanced*, then the amount of data this stage sends equals the amount of data it receives every time. With balanced communication, 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, communication is *unbalanced*: the rate at which data is sent does not always equal the rate at which data is received. With unbalanced communication, a pipeline stage would have to accept and convey buffers at different rates. For example, if a communication stage 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 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. Multiple disjoint pipelines help us to overlap communication with other operations, thereby mitigating latency, when communication is unbalanced. Multiple intersecting pipelines help us to overlap operations, again mitigating latency, when data is consumed from different streams at differing rates, such as when merging streams of data. 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 [7], which was based on out-of-core columnsort, all interprocessor communication was balanced, and so the single-pipeline model sufficed. In contrast, the distribution-based sort has unbalanced communication, and it merges sorted runs.

The distribution-based algorithm has one advantage and two disadvantages compared with the columnsort-based algorithm:

- 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 algorithm makes only two passes to the columnsort-based algorithm's three, and so the columnsort-based algorithm performs approximately 50% more disk I/O.
- In each pass of the distribution-based algorithm, some nodes might read or write differing volumes of data, and therefore some nodes might read or write more than the average volume of data. In the columnsort-based algorithm, all nodes read and write exactly the same volume of data. Thus, the I/O time consumed by the most heavily used disk in a pass might be greater in the distribution-based algorithm than in the columnsort-based algorithm.
- The distribution-based algorithm's I/O and communication operations are determined only dynamically, as the algorithm's execution unfolds, thereby making it difficult to prefetch data. The columnsort-based algorithm's I/O and communication operations are known in advance. Thus, the columnsort-based algorithm is more amenable to overlapping I/O, communication, and computation than is the distribution-based algorithm.

The question is whether the disadvantages of the distribution-based algorithm are enough to outweigh its lesser I/O volume. Experimental results on a cluster show that the distribution-based algorithm runs faster than the columnsort-based algorithm in most cases. Therefore, we can conclude that the disadvantages of the distribution-based algorithm do not dominate its one advantage.

The remainder of this paper is organized as follows. Section II introduces linear pipelines in FG, and Section III summarizes the columnsort-based algorithm, which uses only a single linear pipeline, copied on each node. Section IV describes how we extended FG to support multiple disjoint pipelines and multiple intersecting pipelines, which provide the structure for the implementation of the distribution-based sort outlined in Section V. Section VI presents experimental results on a cluster, showing that with the new features of FG, the distribution-based sort fares well compared with the columnsort-based method. Section VII discusses related work, and Section VIII offers some concluding remarks.

II. Single linear pipelines in FG

FG's original release restricted all programs that used FG to be written using only a copy of a single linear pipeline

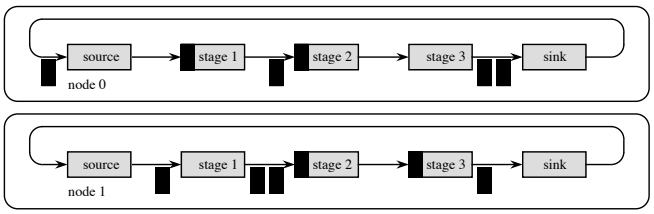

Figure 2. The pipeline of Figure 1, expanded to a standard FG pipeline comprising a source, a sink, and three other stages running on two nodes. Where a buffer, indicated by a black rectangle, appears inside a stage, the stage is currently working on that buffer. Buffers in queues appear below the arrows between stages. The arrow from the sink to the source represents how FG recycles buffers.

on each node. In this section, we introduce some concepts associated with an FG pipeline, and we discuss some details about programming in FG.

Pipelines in FG: As Figure 2 shows, an FG pipeline is composed of stages. FG maps each stage to its own thread. Doing so allows the stages to run asynchronously, so that stages performing high-latency operations can overlap their work with other stages, especially on multicore processors. Because the buffers that traverse the pipeline correspond to blocks for transferring data, the buffer size typically equals the block size for data transfer (as in I/O or communication).

FG places a queue of buffers between each pair of consecutive stages in the pipeline, so that a stage conveys a buffer to its successor by placing the buffer 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 that buffer. If the queue feeding into a stage is empty and the stage tries to accept a buffer from its predecessor, the accept operation blocks and this stage's thread will yield. Once a buffer enters this queue, the thread for this stage will once again become ready to run.

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. Each time that the source emits a buffer into the pipeline, it begins a new *round*. The number of rounds a computation requires can greatly exceed the number of buffers. Therefore, the sink conveys each buffer that reaches it back to the source stage so that the buffer can be recycled. In this way, only a small pool containing a fixed number of buffers needs to be allocated, and the total memory consumed by buffers fits within the physical RAM.

As Figure 2 shows, the typical parallel program that uses FG runs one copy of a single pipeline on each node in a cluster. Figure 2 and the figures that follow show pipelines on only two nodes, but in general we can have any number of nodes, each running its own copy of the pipeline.

Programming with FG: FG transforms a series of programmer-defined functions written in C/C++ into one or more pipelines of asynchronous stages that operate on buffers. To implement each pipeline stage, the programmer writes a straightforward function containing only synchronous calls. FG runs the stages asynchronously, via calls to standard POSIX pthreads functions [8], and it manages the buffers. Although FG was originally designed for clusters, we can obtain additional parallelism when threads can run concurrently on multiple cores. An early paper on FG [2] shows that programs that use FG can be as fast as, or even faster than, hand-tuned programs that call pthreads functions directly.

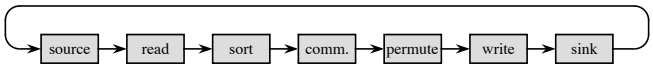

Figure 3. The pipeline structure of the four-pass version of out-of-core columnsort. Each node runs a copy of this pipeline during each pass. Buffers are not shown. The exact operation of the sort, communicate, and permute stages varies among the four passes.

Although FG does provide a framework to overlap high-latency operations with other operations, it does not provide the mechanisms that carry out high-latency operations. The programmer chooses these mechanisms from among those available. Because a program using FG runs multiple threads, these mechanisms must be thread safe. For example, if any stage calls MPI functions for interprocessor communication, the programmer should link in a thread-safe implementation of MPI.

III. Out-of-core columnsort

Using only the features present in the original release of FG, we were able to implement an out-of-core sorting algorithm, described by Chaudhry and Cormen [7] and based on Leighton's columnsort algorithm [9].

Our implementation of out-of-core columnsort mirrors the earlier treatment [7], which we summarize here. Columnsort configures its records as a tall, thin matrix, and it sorts the records into column-major order. Columnsort takes eight steps, where each odd-numbered step sorts every column individually. Each even-numbered step performs a specific, fixed permutation on the matrix.

A relatively simple four-pass implementation of out-of-core columnsort groups together each pair of consecutive steps into a single 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 sorts internally on each node to accomplish an odd-numbered step. The communicate and permute stages accomplish the corresponding 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, steps 5–8 of columnsort, which map to the latter two passes in the four-pass implementation, coalesce 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 the height of half a column. 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 three important properties. First, as mentioned in Section I, 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, every communication step is balanced. That is because the communication steps correspond to the highly regular permutations of the even-numbered steps, such as transposing a matrix or shifting down by half a column. Indeed, all interprocessor communication in the implementation occurs via the MPI calls `MPI_Sendrecv_replace`, `MPI_Alltoall`, and matching pairs of `MPI_Send` and `MPI_Recv` with equal data sizes specified. Third, in each pass, each node reads the same volume of data from disk and writes the same volume of data to disk. Thus, all nodes read and write exactly the average volume of data.

IV. How FG supports multiple pipelines

In order to accommodate stages that accept and convey buffers at different rates, such as when unbalanced communication occurs, we have extended FG. The extensions support multiple pipelines that are disjoint and multiple pipelines that intersect. This section discusses these new FG features in more detail.

Managing multiple disjoint pipelines: 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 I. Each node has two distinct 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 send and receive pipelines interact is that the send pipeline may send data via interprocessor communication to the receive pipeline. Each of the two pipelines on a node has its own source and sink, its own number of buffers, and its own buffer size. Although data

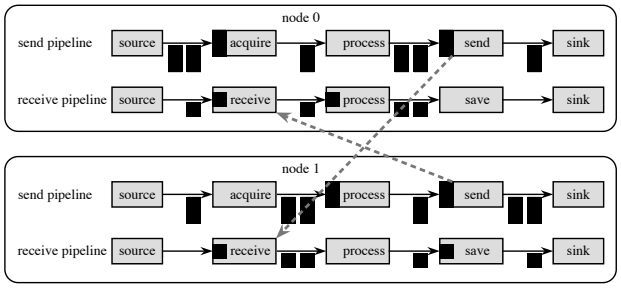

Figure 4. Disjoint pipelines running on each node, shown for just two nodes. The send pipeline has a send stage, which sends data from buffers to various nodes. The receive pipeline has a receive stage, which receives data sent by the various send stages. The number of buffers and their sizes (represented 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.

may move from a buffer within a send pipeline to a buffer within a receive pipeline, 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. If FG 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, in order to accept a buffer, it must specify which pipeline to accept from. Every buffer is tied to a particular pipeline; buffers cannot jump from one pipeline to another.

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 could differ between the vertical and horizontal pipelines. Buffers in the vertical pipelines might be relatively small, since there may be many of them. Although the sorted runs in the vertical pipelines are small compared with 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

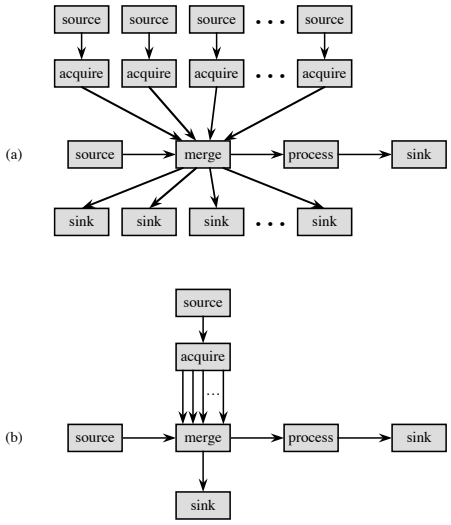

Figure 5. Multiple intersecting pipelines on a given node. These pipelines merge several small, sorted runs of data traveling the vertical pipelines into a single, large sorted sequence traveling the horizontal pipeline. Buffers and arrows connecting the respective sink and source stages are not shown. 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, and each arrow represents a buffer queue.

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, 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 vertical pipelines, and the horizontal pipeline, operates at its own rate. Although the vertical pipelines all have similar structure, they need not work in lockstep with each other. The rate at which a given vertical pipeline operates depends on the rate at which the merge stage consumes data from that vertical pipeline's buffer, which in turn depends on how the merging proceeds. The merge stage also fills each horizontal pipeline buffer at a rate that likely differs 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. A given node could be asked to run hundreds of such vertical pipelines. Because FG creates one thread per stage, including the source and sink, FG would try to create hundreds or even thousands of threads per node when building these intersecting pipelines.

Alas, 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.

FG economizes 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.

V. Out-of-core distribution sort

This section describes how we designed and implemented an out-of-core distribution sort using FG's multiple disjoint and multiple intersecting pipelines. We call this program "dsort." As mentioned in Section I, dsort entails two passes over the data, following a preprocessing phase. The preprocessing phase determines 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 [10]. The records reside in fixed-size blocks, which are assigned in round-robin order to the disks in the cluster. Both dsort and csort create striped output.

Selecting splitters: The first pass partitions N records to be sorted among the P nodes such that each record in node i has a sort key less than or equal to the keys of all records in node $i+1$, for $i = 0, 1, \ldots, P-2$. In order to decide which node 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,

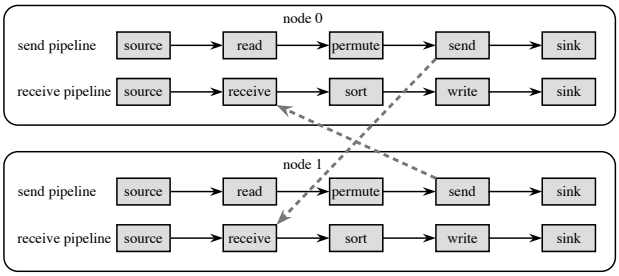

Figure 6. The pipeline structure of pass 1 of dsort, shown for two nodes. The structure is similar to that of Figure 4. Buffers and sink-to-source connections are not shown.

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. [11] and by Seshadri and Naughton [12]. To guard against heavily unbalanced partition sizes when keys are equal, we extend them to make each key unique while deciding where to send each record; the extended keys never actually become part of any record. In our experiments, all partition sizes were at most 10% greater than the average.

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. Each node distributes its records using 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.

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 (a feature that FG provides) 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 the respective target nodes.

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 it conveys them to the write stage, which writes the buffer to disk. Each buffer written contains a sorted run.

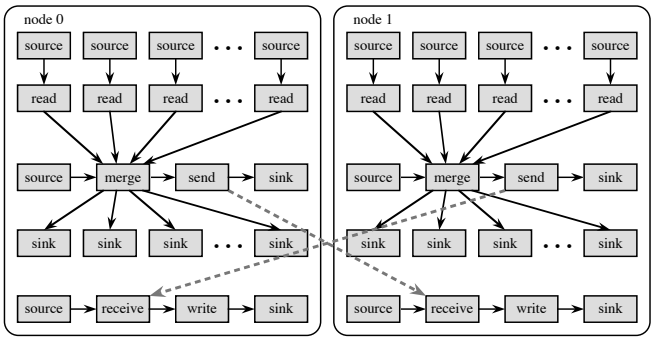

Figure 7. The pipeline structure of pass 2 of dsort, shown for two nodes. The structure is a combination of those in Figures 4 and 5. Buffers and sink-to-source connections are not shown.

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.

VI. Experimental results

In this section, we summarize our experiments with dsort on a cluster, comparing dsort's performance to that of csort.

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. For interprocessor communication, we use ChaMPIon/Pro, a thread-safe, commercial MPI implementation. (We could have instead chosen MPICH2 [13].)

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 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, standard normal, and

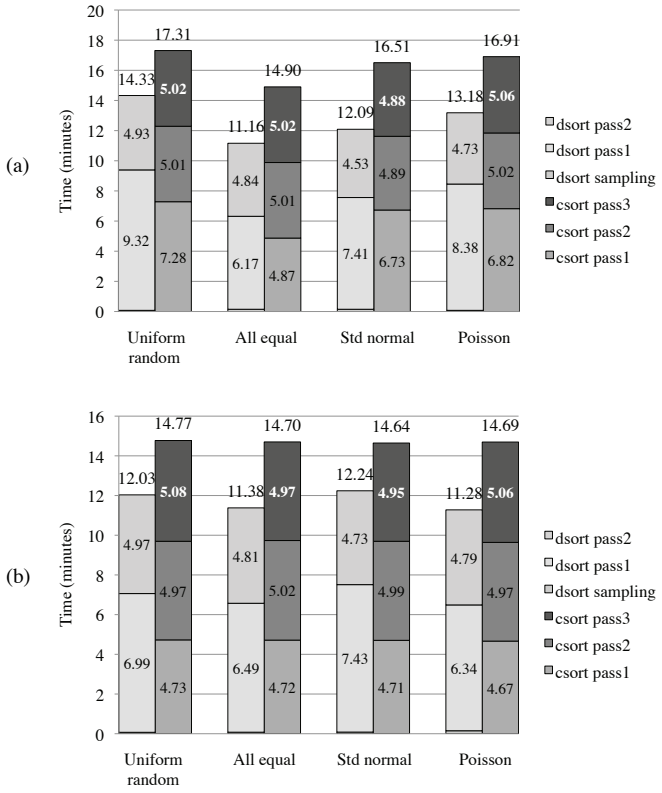

Figure 8. Total and per-pass running times of dsort and csort on various input distributions of 64 GB of data on 16 nodes for **(a)** 16-byte records and **(b)** 64-byte records. For each distribution, the left column represents data for dsort and the right column represents data for csort, with pass numbers going bottom to top. The number within each rectangle shows the running time for that pass, and the number at the top of each column gives the total time for all passes. Because these amounts are negligible, numbers corresponding to dsort's sampling phase are not shown.

Poisson with $\lambda = 1$. As Figure 8 shows, dsort beat csort in each case, taking time in the range 74.26%–85.06% of csort's time. The figure also illustrates that dsort's advantage of having one fewer pass outweighs its disadvantages of having unbalanced I/O and communication patterns. We also compared dsort and csort for input distributions designed to elicit highly unbalanced communication in pass 1 of dsort, and even under these conditions, dsort fared well; space does not permit us to detail these results here.

Unfortunately, we were restricted to input file sizes of only 4 GB per node due to available disk space in our cluster. Although a dataset of size 64 GB might seem "small," given many modern cluster configurations, we believe that our results would also extend to larger inputs. In other words, the dataset size is a limitation of neither dsort nor FG. Indeed, we are arranging to run dsort and csort on a cluster with nodes that have more disk space available.

VII. Related work

In this section, we discuss other programming models that are similar to FG, and we compare dsort with some distribution-sort implementations.

Programming models related to FG: FG came about as a framework for implementing algorithms for the Parallel Disk Model. Its closest relatives are the TPIE project and the `<stxxl>` library.

TPIE [14] provides a set of structures, implemented as C++ classes, for manipulating streams of disk-resident data. Although TPIE does provide classes for merging and distribution (the inverse of merging), it cannot support pipeline structures as general as the multiple pipelines that FG supports, and TPIE programs cannot run in parallel on a cluster or even with shared memory.

`<stxxl>` [15] replicates some of the functionality of the C++ Standard Template Library [16, Part III]. It provides classes for vectors, stacks, and queues, as well as a small number of algorithms, including sorting, all with built-in support for data that resides on parallel disks on a single node. `<stxxl>` now also supports pipelining [17] and allows constructs that resemble FG's fork-join and intersecting pipelines. Asynchronous nodes within a pipeline seem to provide a buffer-passing mechanism between successive nodes.

To some degree, FG resembles dataflow programming (see, for example, the survey by Johnston, Hanna, and Millar [18]), where stages in FG correspond to dataflow operations connected by queues holding data. FG and traditional dataflow programming differ primarily in granularity: dataflow programming tends to be fine grained, compared with FG's coarse-grained approach. Whereas dataflow programming applies to data already in memory, FG is designed for high-latency access to data from other nodes or from the outer levels of the memory hierarchy.

Other distribution-sort implementations: One of the pioneering and successful works in the area of out-of-core, distribution-based sorting is NOW-Sort [19]. Although dsort and NOW-Sort share the same two-pass design, NOW-Sort assumes that the splitters are known in advance and does not output the final sorted result in PDM ordering. Moreover, current cluster configurations render the results reported by NOW-Sort obsolete.

In more recent work, Rahn, Sanders, and Singler [20] describe CANONICALMERGESORT, an `<stxxl>`-based distributed-memory implementation of the parallel multiway merging approach described by Varman et al. [21]. CANONICALMERGESORT achieves perfect load-balancing after partitioning the data, but it does not stripe the final output across the nodes of the cluster. Furthermore, depending on the input and machine parameters, the algorithm might require more than two passes. In their paper, Rahn, Sanders, and Singler report results of using CANONICALMERGESORT to sort many gigabytes and terabytes of data, using hardware that differs significantly from the hardware that we used to run dsort. We have contacted the authors for their implementation so that we may be able to make a fair comparison.

VIII. Conclusions

When we started this project, we expected results in line with the earlier results [7], in which csort prevailed due to the advantages that we listed in Section I. Much to our surprise, dsort ran faster.

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 the way in which dsort depends on key values. Multiple disjoint pipelines support unbalanced communication, 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 assembling them properly.

An obvious question would be how much faster dsort runs with multiple pipelines on each node compared with an implementation restricted to single, linear pipelines on each node. Although we have not investigated this issue yet, we are developing such an implementation. The design of dsort using only linear pipelines entails extensive bookkeeping on the programmer's part for stages that perform interprocessor communication, as well as the merge stage. The results will help us test the claims about FG's multiple pipelines that we made in the previous paragraph in terms of both dsort's performance and that of the programmer's coding burden.

As mentioned in Section VI, we would like to try running dsort on larger inputs. Furthermore, we believe that the extensions to FG that we have discussed in this paper would be suitable for the design of out-of-core algorithms other than sorting; we are actively soliciting suggestions for such algorithms.

Acknowledgments

We thank Tim Tregubov for his assistance in setting up the cluster that we used for our experiments. Doug McIlroy, Laura Toma, and anonymous referees provided valuable comments and suggestions.

References

[1] J. S. Vitter, "External memory algorithms and data structures: Dealing with MASSIVE DATA," *ACM Computing Surveys*, vol. 33, no. 2, pp. 209–271, Jun. 2001.

[2] T. H. Cormen and E. R. 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)*, Sep. 2004, pp. 137–144.

[3] E. R. Davidson, "FG: Improving parallel programs and parallel programming since 2003," Ph.D. dissertation, Dartmouth College Department of Computer Science, Aug. 2006.

[4] E. R. Davidson and T. H. Cormen, *Asynchronous Buffered Computation Design and Engineering Framework Generator (ABCDEFG): Tutorial and Reference*, Dartmouth College Department of Computer Science, available at http://www.cs.dartmouth.edu/FG/.

[5] ——, "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)*, Apr. 2005.

[6] ——, "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)*, Feb. 2005.

[7] G. Chaudhry and T. H. Cormen, "Oblivious vs. distribution-based sorting: An experimental evaluation," in *13th Annual European Symposium on Algorithms (ESA 2005)*, ser. Lecture Notes in Computer Science, vol. 3669. Springer, Oct. 2005, pp. 317–328.

[8] IEEE, "Standard 1003.1-2001, Portable operating system interface," http://www.unix.org/version3/ieee_std.html, 2001.

[9] T. Leighton, "Tight bounds on the complexity of parallel sorting," *IEEE Transactions on Computers*, vol. C-34, no. 4, pp. 344–354, Apr. 1985.

[10] J. S. Vitter and E. A. M. Shriver, "Algorithms for parallel memory I: Two-level memories," *Algorithmica*, vol. 12, no. 2/3, pp. 110–147, Aug. and Sep. 1994.

[11] 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*, vol. 31, pp. 135–167, 1998.

[12] S. Seshadri and J. F. Naughton, "Sampling issues in parallel database systems," in *3rd International Conference on Extending Database Technology (EDBT '92)*, ser. Lecture Notes in Computer Science, A. Pirotte, C. Delobel, and G. Gottlob, Eds., vol. 580. Springer-Verlag, March 1992, pp. 328–343.

[13] MPICH2 home page. http://www.mcs.anl.gov/research/projects/mpich2/.

[14] TPIE home page. http://www.madalgo.au.dk/Trac-tpie/.

[15] R. Dementiev, STXXL *Tutorial*, Jun. 2006, available from stxxl.sourceforge.net.

[16] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley, 1997.

[17] A. Beckmann, R. Dementiev, and J. Singler, "Building a parallel pipelined external memory algorithm library," in *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009)*, May 2009.

[18] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1–34, Mar. 2004.

[19] 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*, 1997.

[20] M. Rahn, P. Sanders, and J. Singler, "Scalable distributed-memory external sorting," 2009, available at http://arxiv.org/abs/0910.2582.

[21] P. J. Varman, S. D. Scheufler, B. R. Iyer, and G. R. Ricard, "Merging multiple lists on hierarchical-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 171–177, Jun. 1991.