Building on a Framework:
Using FG for More Flexibility
and Improved Performance in Parallel Programs

Elena Riccio Davidson *
Thomas H. Cormen†

Dartmouth College Department of Computer Science
{laney, thc}@cs.dartmouth.edu

Abstract

We describe new features of FG that are designed to improve performance and extend the range of computations that fit into its framework. FG (short for Framework Generator) is a programming environment for parallel programs running on clusters. It was originally designed to mitigate latency in accessing data by running a program as a series of asynchronous stages that operate on buffers in a linear pipeline. To improve performance, FG now allows stages to be replicated, either statically by the programmer or dynamically by FG itself. FG also now alters thread priorities to use resources more efficiently; again, this action may be initiated by either the programmer or FG. To extend the range of computations that fit into its framework, FG now incorporates fork-join and DAG structures. Not only do fork-join and DAG structures allow for more programs to be designed for FG, but they also can enable significant performance improvements over linear pipeline structures.

1. Introduction

FG is a C++ programming environment for asynchronous parallel programs running on clusters that fit into a pipeline framework [DC]. It reduces both the source code size and execution time of these programs. In this paper, we introduce optimizations to FG that improve performance even more, reducing execution times as much as approximately 4% for out-of-core parallel computations and as much as approximately 15% for in-core computations. We also introduce extensions to the framework that broaden the scope of FG beyond a linear pipeline. These extensions make FG more flexible and adaptable, and they have also yielded performance gains of up to 43% for some in-core computations.

FG arose from prior work in hiding the high latency that is inherent in many out-of-core programs. An out-of-core program operates on a dataset so large that it exceeds the capacity of main memory. The data, therefore, must reside on disks. Many out-of-core algorithms in the literature (e.g., [BC99, CCW01, CSW99, CWN97, Vit01, VS94]) share a similar structure: they make multiple passes over the data, and each pass typically includes high-latency operations such as disk I/O and, when run on a distributed-memory cluster, network communication. In order to amortize the cost of transferring data among levels of the memory hierarchy, these programs move data in blocks.

On its own, moving data in blocks is not enough to achieve the highest performance. The costs of reading, writing, and communicating often dominate the computation costs. The designer of such programs, therefore, must write code to overlap these high-latency operations with other tasks to create the asynchrony necessary for a high-performance implementation. This code, which we call "glue," is often the most difficult code to design, write, and debug. The FG framework eliminates the need for the programmer to write this code but retains the elements necessary for high-performance implementations.

FG assumes the dual tasks of managing blocks of data (buffers) and of using asynchrony to overlap work. In other words, FG provides the glue. All that is left to the programmer is to write a series of straightforward, synchronous functions. The framework transforms these functions into a pipeline of stages and maps each stage to a thread. FG also passes buffers through the pipeline, so that the stages of the pipeline work on different buffers concurrently. Many out-of-core algorithms fit neatly into a pipeline structure, and so we designed the heart of FG as an asynchronous pipeline. Figure 1 shows one pass of out-of-core columnsort [CC02, Cha04]. There are five stages in the pipeline, and at any one moment each of them may be working on a distinct buffer. FG makes such pipeline-structured programs smaller and faster. Indeed, our previous experimental results have shown that writing programs with

*Supported in part by DARPA Award W0133940 in collaboration with IBM.

†Contact author. Supported in part by DARPA Award W0133940 in collaboration with IBM and in part by National Science Foundation Grant IIS-0326155 in collaboration with the University of Connecticut.

FG affords significant reductions in both source-code size and running time [CD04].

This paper introduces extensions to FG that fall into two broad categories. First, we have built into FG optimizations to improve performance. Programs written with FG were already faster than programs written without FG. Improvements on the original FG model make some of them even faster. The optimizations compensate for stages in the pipeline that are not completing work at the same rate as other stages. FG may devote either fewer CPU timeslices to a pipeline stage that is progressing faster than other stages or more CPU timeslices to a stage that is not progressing fast enough. Second, we have expanded FG for programs that do not fit a simple, linear pipeline structure. FG accommodates programs that are configured as split pipelines or even as directed acyclic graphs (DAGs). We have implemented a fork-join construct that allows a pipeline to split into separate paths and later regather into a single pipeline. We have also incorporated a DAG construct, which enables programs that do not fit a sequential order. These extensions make the framework more general and adaptable.

The performance optimizations in FG reduce execution times for some programs. FG compensates for a stage that is progressing too quickly by lowering the priority of its associated thread, which our experimental results show reduces execution time by approximately 1% for out-of-core columnsort and 7% for in-core columnsort. FG compensates for a stage that is not progressing quickly enough, a *bottleneck stage*, by creating an additional copy of the stage. In our experiments, in which each node has two processors, this adjustment reduces execution time by approximately 2% for out-of-core columnsort and 14% for in-core columnsort. We can also combine the two optimizations for out-of-core columnsort, resulting in speedups of approximately 4%.

The structural extensions to FG yield speedups as well in some cases. In a linear pipeline, every buffer must travel to every stage. Replacing a linear pipeline with a fork-join construct or a DAG allows buffers to skip over stages that may be superfluous. In one experiment, replacing a linear pipeline with a fork-join construct reduced execution time for an in-core program by just under 6%. In an another experiment, replacing a linear pipeline with a DAG for a different program resulted in a dramatic performance improvement of approximately 43%.

The remainder of this paper is organized as follows. Section 2 outlines the basic structure of FG. Section 3 discusses the inclusion of stage replication and adjusting thread priorities as performance optimizations. Section 4 examines fork-join constructs and DAGs, the more flexible program configurations that broaden the scope of FG beyond the basic pipeline. Section 5 summarizes our experimental results. Finally, Section 6 offers some concluding remarks.

2. FG background

FG makes it easy for a programmer to transform a series of simple, synchronous C or C++ functions into an asynchronous pipeline. The programmer specifies the work that each stage of the pipeline performs. FG maps each stage to a thread in order to overlap work; we use standard POSIX threads in our implementation. FG also constructs and maintains the pipeline, so that each buffer traverses the stages of the pipeline in order. FG provides mechanisms for these pipeline-structured programs to run correctly and efficiently, each of which we discuss in this section: buffer management, source and sink stages, and program flexibility.

2.1. Buffer management

It is essential to use buffers to amortize the cost of passing data among levels of the memory hierarchy. A buffer's size corresponds to a block size in the memory hierarchy.

Without middleware such as FG, managing these buffers would place a significant burden on the designer of out-of-core parallel programs. Each buffer must traverse the pipeline from the first stage to the last. The programmer would have to ensure that each stage has information about its predecessor and successor stages in order to coordinate semaphores and move buffers through the pipeline. The buffers must remain in order relative to each other, and they must respect the linear order of the pipeline. The programmer also would be responsible for allocation and deallocation of buffers. Moreover, since it is not possible to allocate enough memory at one time to handle all of the data in an out-of-core program, the programmer would have to write code to reuse buffers that have traveled the entire pipeline.

FG takes over responsibility for the buffers. The programmer need only specify the number and size of buffers at the outset. FG allocates the buffers and passes them from stage to stage. As in UNIX pipes, no stage has direct knowledge of its neighbors; instead, FG ensures that buffers traverse the pipeline in order. Because the stages of a pipeline do not necessarily work on each buffer for the same amount of time, it is not sufficient for FG to convey buffers one at a time from a stage to its successor. Instead, FG installs FIFO queues between each pair of stages. Each stage repeatedly *accepts* a buffer from its incoming queue, operates on the buffer, and *conveys* the buffer to its outgoing queue. Operating on a buffer may entail interprocessor communication, in which some data in the buffer moves to other processors and is replaced by data received from other processors. Each time a buffer completes its journey through the pipeline, the program has completed one *round* of execution. Typically, there are far more rounds than buffers, and so FG reuses any buffer that has finished a round.

FG stores information about each buffer in a *thumbnail*. A thumbnail points to its corresponding buffer, and it also maintains some supplementary data. Each buffer always has a round number, so that FG and the stages can keep track of

Figure 1: The stages of the pipeline for one pass of out-of-core columnsort. Each stage accepts a buffer from its predecessor, operates on the buffer, and conveys the buffer to its successor. Stages work asynchronously so that each one may be working on a distinct buffer (indicated by a black rectangle attached to the stage) at any moment.

progress in the pipeline. The final buffer to enter the pipeline also has a flag, called the *caboose*, indicating that no additional buffers should follow that buffer in the pipeline. Finally, there may be user-defined parameters associated with a buffer, declared and used by the programmer but unused by FG. Thumbnails store all of this information.

Thumbnails serve an additional purpose: they allow the programmer to swap entire buffers. Buffer swapping avoids copying for a stage whose work cannot be done in place, such as a permutation. The permutation stage of columnsort, for example, accepts a buffer from its predecessor stage, and it permutes the buffer's data into another, secondary buffer. This stage can ask FG for the secondary buffer, a type of buffer we call an *auxiliary buffer*. Auxiliary buffers do not traverse the pipeline but are simply blocks of allocated memory that any stage may request. We store information about each auxiliary buffer in an *auxiliary thumbnail*. The permutation stage accepts an ordinary buffer, which we call a *pipeline buffer*, from its predecessor and requests an auxiliary buffer from FG. The permutation stage moves data from the pipeline buffer to the auxiliary buffer, rearranging it along the way. After the stage completes its work, the auxiliary buffer contains the result of the permutation. Hence, the stage should convey the auxiliary buffer, and not the pipeline buffer, to the next stage. Since each of these buffers belongs to a thumbnail, we can simply swap the buffer pointers stored in the two thumbnails. After the swap, the permutation stage simply conveys its pipeline thumbnail, which now points to what had been the auxiliary buffer. It is this buffer that FG conveys to the next stage in the pipeline.

2.2. Source and sink

FG bookends the programmer-defined pipeline with a source stage at the head of the pipeline and a sink stage at the tail. These two stages are collectively responsible for introducing buffers into the pipeline, recycling buffers that have traversed the entire pipeline, and shutting down the pipeline.

The source stage collects buffers and passes them to the first programmer-defined stage. Every time the source stage conveys a buffer, another round begins, and the source stage assigns the corresponding round number to that buffer's thumbnail.

The source stage and the sink stage work together to recycle buffers. Since there are many more rounds than buffers in a pipeline, FG recycles each buffer when it reaches the end of the pipeline by having the sink stage pass the buffer back to the beginning of the pipeline. The source stage then reassigns the recycled buffer a round number and emits it into the pipeline.

The sink stage has a secondary responsibility as well: when the final buffer—as indicated by the caboose—reaches the sink, the sink shuts down the pipeline. FG also takes on the burdensome job of shutting down the pipeline. The sink stage, upon seeing the final buffer, deallocates the buffers, kills the threads and semaphores, and cleanly shuts down each stage of the pipeline.

2.3. Flexibility

An inescapable part of writing parallel programs is tweaking the code—a programmer often makes frequent changes in order to discover small speedups. In the context of FG, such refinements might include altering the mapping of stages to threads (FG allows multiple stages to map to the same thread) or experimenting with different pipeline structures. Each of these changes requires rewriting the glue of the program, not the algorithm itself. Without FG, making these adjustments requires a significant amount of redesigning, programming, and debugging. Furthermore, the payoff from a given change may be relatively low compared to the effort required. With FG, a programmer rewrites only a few lines of code for the change to be complete. FG makes it easy to reconfigure a pipeline many different times, and in many different ways, to find an effective combination of pipeline structure, stages, threads, and buffers.

3. Improving performance

In this section, we discuss performance optimizations in FG. First, we present stage replication. When additional processing power is available, FG compensates for a bottleneck stage by making an additional copy of it. The two copies work on different buffers at the same time, thus passing buffers through the pipeline more quickly. Stage replication can be done statically by the user or dynamically by FG. Second, we discuss adjustment of thread priorities. FG compensates for a stage that is progressing faster than other stages in the pipeline by lowering the priority of its thread. Lowering priorities can also be done statically or dynamically. Our experimental results, presented in Section 5, show that both optimizations can yield performance improvements, and we see the greatest speedups from combining the two.

3.1. Static stage replication

It is often the case that a programmer knows ahead of time whether a pipeline contains a bottleneck stage. The programmer may specify that such a stage should be replicated before the pipeline begins. FG will spawn an additional copy of that stage so that it gets more CPU timeslices. We call this user-initialized method *static replication*.

There may be several rounds' worth of work before a stage emerges as a bottleneck. Given a pipeline with a user-replicated stage, however, FG spawns the additional copy of that stage immediately, rather than waiting for it to become a bottleneck. Moreover, the programmer can replicate a stage in the midst of execution, rather than just at the beginning.

Asking for a stage to be replicated requires only one additional line of code. FG spawns the additional thread and ensures that buffers traverse the pipeline in order. It is easy to try replicating a stage or two to see if the change speeds up the pipeline, and it is just as easy to eliminate the replication if it proves unfruitful.

3.2. Dynamic stage replication

The programmer need not have advance knowledge of performance bottlenecks in the program. FG constantly monitors the stage queues and replicates any stage that becomes a bottleneck in the midst of execution. We call this FG-initialized method *dynamic replication*. In our experimental results, dynamic replication fares almost as well as static when it comes to reducing execution time.

FG considers a stage to be a bottleneck if there are multiple buffers residing in its input queue. These buffers are not making progress through the pipeline because they are waiting for that stage to accept them. Any stage that is letting buffers build up in its input queue is slowing down the pipeline overall.

When FG discovers a bottleneck stage in the course of executing the pipeline, it replicates that stage. How many additional copies of the stage should FG run? Ideally, each copy would run on a separate processor to exploit parallelism. Of course, all of these processors must share memory so that the threads interact properly. In other words, the most obvious answer would be to set the number of replications to the number of processors per node. Making so many copies of a stage can introduce performance problems, however.

If a stage uses auxiliary buffers, replicating the stage may not improve performance at all. Let us consider a pipeline with only one auxiliary buffer, and suppose that a bottleneck stage requests the auxiliary buffer. FG replicates the stage when it becomes clear that the stage is a bottleneck. All copies of the stage request the auxiliary buffer, but only one copy at a time can get it. Therefore, the other copies of the stage are forced to wait until that one copy releases the auxiliary buffer. Replicating that stage, therefore, would not speed up the pipeline, even though it is a bottleneck stage. For this reason, FG allows the programmer to declare any stage as unreplicatable.

Using stage replication introduces the possibility of changing the order of buffers in the pipeline. In the columnsort pipeline shown in Figure 2, we replicate the sort and permute stages with sort′ and permute′, respectively. For the moment, let us turn our attention to the replicated sort stage. The sort stage is working on buffer 4 and the sort′ stage is working on buffer 5. It is possible that the sort′ stage completes before the sort stage, so that buffer 5 is ready to move on to the communicate stage before buffer 4. If the communicate stage accepts buffer 5 first, then the buffers traverse the pipeline out of order. If the communicate stage waits to accept buffer 5 until after the sort stage has conveyed buffer 4, the buffers traverse the pipeline in order but there could be a performance slowdown caused by waiting for the sort stage to finish.

Which order method do we choose for a replicated stage? FG leaves it up to the user. FG can convey buffers from a replicated stage in either first-come first-served (FCFS) or round-robin (RR) order. If the user chooses FCFS order, a replicated stage conveys its buffers immediately, but the buffers may traverse the remainder of the pipeline out of order. If, on the other hand, the programmer chooses RR order, the buffers always make progress in order, but the execution time of the pipeline may be slowed.

Whether the programmer chooses FCFS or RR order, FG always conveys buffers from the sort stage and the sort′ stage as soon as possible. The availability of a buffer to the communicate stage may be delayed in the case of RR order, but sort′ can move immediately to the next buffer—FG's buffer queues enable a stage to convey its buffers even if the successor stage is not ready to accept them.

3.3. Thread priorities

It is not only bottleneck stages whose CPU time allotment is imperfect. FG also compensates for stages that are progressing more quickly than other stages in the pipeline. FG uses thread priorities to devote fewer CPU timeslices to a stage that is finishing off buffers too quickly.

We use the POSIX thread library in Linux RedHat version 9.0. If the programmer attempts to increase a thread's priority by calling a function in the Linux thread library, no noticeable change results. We have found, however, that lowering a thread's priority when it is completing its work quickly, rather than raising a thread's priority when it becomes a bottleneck, can be effective.

FG can adjust thread priorities both statically and dynamically. If the programmer is aware ahead of time that a particular stage will operate on its buffers more quickly than other stages in the pipeline, the programmer may lower the thread's priority upon initialization of the pipeline. As with stage replication, FG also makes adjustments dynamically. FG monitors the queues that sit between stages. If a stage is passing buffers through the pipeline more quickly than other stages in the pipeline, FG lowers its priority automatically. We have found

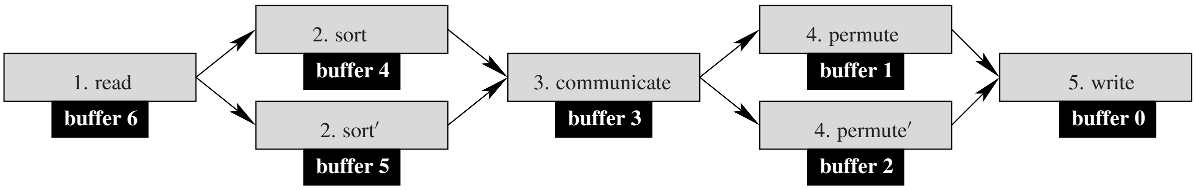

Figure 2: A pass of columnsort with two replicated stages. We replicate the sort stage with sort′, and we replicate the permute stage with permute′. Each coupling of a stage and its copy can work on different buffers concurrently. Hence, the sort and permute functions are allocated more CPU timeslices to speed overall execution.

that adjusting thread priorities both statically and dynamically yields modest performance gains for out-of-core and in-core columnsort.

4. Beyond linear pipelines

Some parallel programs do not quite fit the pipeline model. In this section, we introduce features of FG that incorporate additional computational structures. These structures allow different kinds of programs to be written with FG, and they can also improve performance for some pipeline-structured programs. There are two distinct extensions to the basic pipeline structure, and we will examine each in this section. First, we introduce the fork-join construct within a pipeline. Fork and join split a pipeline into multiple paths and then regather it into one pipeline. Second, we discuss DAG structures in FG. In a DAG, a stage may convey a thumbnail to a stage that is not its linear successor.

4.1. Fork-join

A linear pipeline may split into two—or even more—separate paths with a *fork*. Separate paths in turn may merge back into one pipeline with a *join*. A forked stage does not have one successor stage as it would in a simple, linear pipeline structure. Rather, a forked stage has many successor stages, one for each path. A fork in a pipeline always has a corresponding join. A join stage has multiple predecessor stages, rather than the single predecessor it would have in a linear pipeline.

Figure 3 shows a pipeline with a fork and join. Stage A is followed by a fork, and it has successor stages B and C. Stage A conveys buffers in round-robin order to stages B and C. Therefore, stage A conveys buffer 0 to stage B, buffer 1 to stage C, buffer 2 to stage B, and so on. Stage B conveys all of its buffers to its linear successor, as does stage C. Further down the pipeline, the split paths regather into a single pipeline. Stage T follows a join and has predecessor stages R and S. Stage T accepts buffer 0 from stage R, buffer 1 from stage S, buffer 2 from stage R, and so on.

How do we convey buffers from a join to its single successor? The principle is similar to the convey choices for stage replication: the user must choose between FCFS and RR order. For the pipeline shown in Figure 3, let us first consider FCFS order. Suppose that stage R is working on buffer X and stage S is working on buffer Y. If stage S finishes its computation before stage R, then FCFS order dictates that stage T accepts buffer Y from stage S as soon as the buffer is ready. Stage T, therefore, may accept its buffers out of order so as to avoid a performance slowdown incurred by waiting for buffers in order.

Now let us consider RR order in the same scenario. Assume that stage S has finished working on buffer Y and conveys it to stage T, but stage R is still working on buffer X. RR order dictates that stage T does not accept buffer Y until after it has accepted buffer X. Therefore, stage T waits for stage R to convey buffer X, accepts buffer X, works on buffer X, and conveys buffer X, before accepting buffer Y. With RR order, stage T is guaranteed to accept its buffers in consecutive order, but there may be a tradeoff in performance if its predecessor stages finish out of order.

In some cases, using a fork-join can make for more efficient buffer management than using a linear pipeline. Figures 4 and 5 illustrate the difference between a linear pipeline and a fork-join construct on a program that shears its input. For example, the Shearsort algorithm [Lei92, pp. 144–148], [SI86], [SSS86] shears rows of a mesh: it sorts every other row in ascending order and the remaining rows in descending order. In the context of FG, shearing a dataset entails reading in buffers, sorting them alternately into ascending or descending order, and writing out the sorted buffers in order. Therefore, the pipeline that performs a shearing operation includes a stage that sorts in ascending order and a stage that sorts in descending order.

The pipeline for our shearing program is set up as follows. The first stage reads data from a file. The sort-up stage performs a sort in ascending order. The sort-down stage performs a sort in descending order. The final stage writes the sorted buffers to disk.

Figure 4 shows the linear version of the shearing program. Buffers travel sequentially through each stage. When the sort-up stage accepts a buffer, it checks the round number attached to that buffer. If it is an even round, the stage performs the ascending sort. Otherwise, it conveys the buffer to its successor stage without sorting. The sort-down stage acts likewise: it accepts every buffer, but only sorts the buffers for odd rounds. The write stage simply writes out the buffers, which arrive alternately sorted into ascending and descending order.

Figure 5 shows the shearing program with a fork-join con-

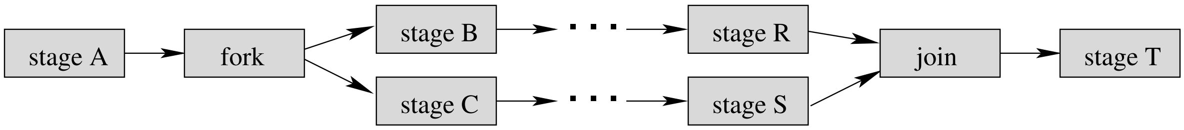

Figure 3: A pipeline with a fork-join construct. Following the fork, the pipeline splits into two separate paths. Following the join, the two pipelines regather into a single pipeline. Stage A conveys its buffers alternately to stage B and stage C. Stage T accepts buffers alternately from stage R and stage S.

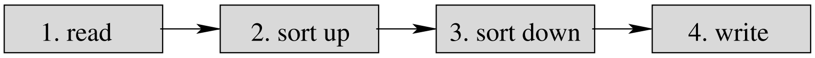

Figure 4: A linear pipeline for a shearing program. Every buffer travels to every stage. The first stage reads data from a file into its buffer. The sort-up stage sorts every other buffer in ascending order. The sort-down stage sorts every other buffer in descending order. The final stage writes the sorted buffers to disk.

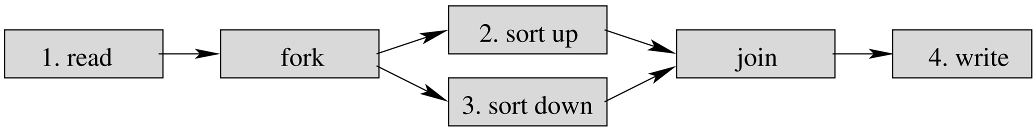

Figure 5: A pipeline that uses a fork-join to implement a shearing program. The first stage reads data from a file. The second stage introduces a fork, alternating buffers between sort up and sort down. The join collects the buffers from the two sort stages. The final stage of the pipeline writes the alternately sorted buffers to a file.

struct. Instead of traversing the pipeline in a linear fashion, different buffers take different paths. The fork ensures that buffers for even-numbered rounds travel to the sort-up stage and buffers for odd-numbered rounds travel to the sort-down stage. The join ensures that the write stage accepts every buffer and, by specifying RR order, the write stage accepts buffers in order.

In reality, FG does not introduce a fork stage and a join stage into the pipeline. Rather, it uses the fork and join placement as cues from the user to reassign the ordering of stages in the pipeline. FG alters the way buffers move from queue to queue, but it does not add new stages or threads to the pipeline.

4.2. DAG-structured computation

In addition to linear pipeline and fork-join setups, FG now applies to programs that fit a DAG structure. A stage may convey its buffers to any other stage rather than to only its linear successor, as long as the structure of the stages is acyclic. Moreover, stages can make these decisions on the fly, so that FG need not know in advance the paths that buffers take through the pipeline.

The ability to construct a DAG makes FG more flexible. Consider the pipeline for out-of-core columnsort shown in Figure 1. The pipeline comprises five stages and represents one pass of columnsort. In fact, columnsort requires four separate passes. Columnsort is an eight-step algorithm, and each pass of the implementation executes two of the eight steps, along with disk I/O and interprocessor communication. Each pass builds a new pipeline that resembles the one shown in Figure 1, but the passes are not quite identical to each other. Although each pass has the same basic structure, there are two different read stages, four sort stages, three communicate stages, one permute stage, and three write stages. The particular combination of read, sort, communicate, permute, and write stages changes in every pass.

With only a linear pipeline at our disposal, we build the four pipelines for columnsort one at a time: we run the pipeline for the first pass, dismantle it, build an entirely new pipeline for the second pass, dismantle that pipeline, and so on. Figure 6 illustrates how we can instead build only one pipeline that is able to run all four passes of columnsort. All 13 columnsort stages are included in the pipeline. We also introduce a new stage, initialize, which assigns each buffer to a particular path through the pipeline according to the pass number. We build the same DAG, with the same 13 stages, for each pass of columnsort, instead of building a new pipeline with different stages in each pass.

The DAG shown in Figure 6 bears a resemblance to a fork-join construct, but in fact it is different. With a fork-join, each buffer follows a pre-determined path: buffers predictably alternate between paths of the fork and then eventually all buffers enter the same single pipeline. With a DAG, we can choose each buffer's path on the fly. Moreover, the buffers need not travel each available path alternately. In columnsort, for example, passes 1–3 use the read 1 stage and pass 4 uses the read 2 stage. Consequently, more buffers traverse the read 1 stage than the read 2 stage.

We can also use the DAG from Figure 6 to circumvent the straightforward method of building a new pipeline for each pass. We build the DAG only once and execute all four passes of columnsort without altering its structure or building a new DAG. The initialize stage then tracks the overall round number as the program progresses. When the program has executed sufficient rounds for one pass of the program, the ini-

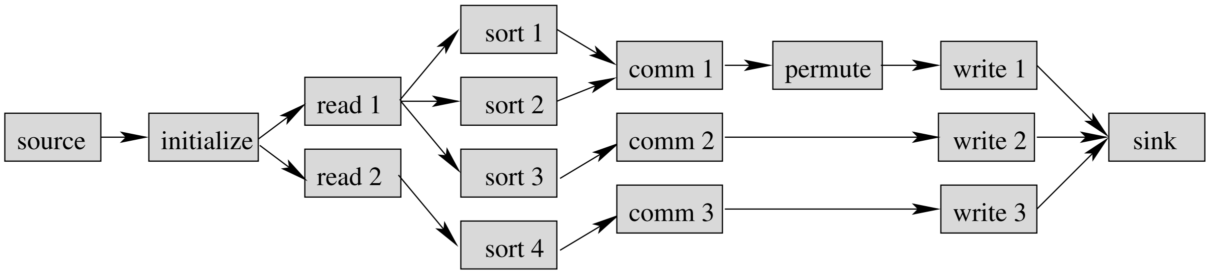

Figure 6: All passes of out-of-core columnsort, configured in one DAG structure. There are two distinct read stages, four sort stages, three communicate stages, one permute stage, and three write stages. Different buffers follow different paths according to the pass number. The initialize stage determines the pass number on the fly and assigns the corresponding path to each buffer.

tialize stage alters the path of its buffers to simulate the start of a new pass.

Using a DAG instead of a linear pipeline for out-of-core columnsort does not have a measurable effect on performance. We can use either the pass-by-pass DAG, which builds a new, identical DAG for each pass, or the single-pass DAG, which builds only one DAG and executes all four passes without tearing it down. The variation among the running times for all three possibilities is negligible.

Implied in the DAG feature is the ability to discard a buffer midway through the pipeline. FG allows a buffer to move from an arbitrary stage in the pipeline to the sink, rather than traversing the entire pipeline sequentially. If some stage discovers that a buffer contains no useful data, the stage can convey that buffer to the sink, bypassing the remaining pipeline stages. The sink processes the buffer as usual: it conveys the buffer to the source, where it gets recycled into the pipeline. This capability can also improve performance. By immediately reusing a buffer whose data is no longer needed, the programmer can avoid the overhead associated with passing that buffer through the pipeline. Furthermore, buffers can make progress more efficiently by keeping the pipeline full.

We can use a DAG effectively with an in-core, filtered FFT. An FFT program might have to filter out noisy data. If a buffer contains noise, our pipeline will not compute the FFT on it. We implemented two pipelined versions of an FFT that uses a filter. We assume that each buffer has already undergone a permutation, and so our pipeline consists of a read stage, one stage for every level of the butterfly network and a write stage. The first version is a simple, linear pipeline. Every buffer travels to every stage. Each stage performs its computation on the buffer only if it does not contain noise; otherwise, it conveys the buffer without touching its data. The second is a DAG implementation—the read stage determines whether a buffer contains noise, and if so, conveys it to the sink. We show in our experimental results that the DAG implementation can be much more efficient than the linear pipeline.

5. Experimental results

We present our experimental results for both of the performance optimizations we have introduced: stage replication and adjusting thread priorities. We have found that both of these optimizations reduce running time for pipeline-structured programs. For benchmarks, we use separate implementations of columnsort—one is out-of-core and one is in-core. For out-of-core columnsort, stage replication reduces running time by 1.70–2.99%. Adjusting thread priorities yields a more modest gain of 0.46–1.11%. We see the most signifcant improvements, however, by using both of the optimizations—up to 3.99%. For in-core columnsort, stage replication reduces running time by up to 14.84% and adjusting thread priorities reduces running time by up to 7.18%.

Additionally, this section presents experimental results that come from employing the more flexible FG structures when they are suitable. For benchmarks, we use an in-core shearing program and an in-core filtered FFT. We implemented each of these programs using a simple, linear pipeline that passes every buffer to every stage. We compare the running times for these linear pipelines with a fork-join construct for the shearing program and a DAG for FFT. Replacing a linear pipeline with a fork-join yields a performance improvement of up to 5.80% for shearing, and replacing a linear pipeline with a DAG yields an improvement of up to 43.36% for a filtered FFT with a discard rate of 1/2.

We ran our experiments on a Beowulf cluster with 16 dual 2.8-GHz nodes. Each node has 4 GB of RAM and an Ultra-320 36-GB hard drive. The nodes run RedHat Linux 9.0 and are connected with a 2-Gb/sec Myrinet network. We use the C stdio interface for disk I/O and the pthreads package of Linux. The out-of-core applications use the ChamMPIon/Pro implementation of MPI for interprocessor communication.

Out-of-core columnsort makes four separate passes over a dataset of 64-byte elements. It runs in parallel on all the nodes of our cluster, and it makes calls to MPI to pass data among processors; note that it is the user-defined functions, and not FG, that makes the MPI calls. We ran experiments using 4 GB and 8 GB of data per processor. Experiments that use fewer than 4 GB of data per processor are less meaningful because file-caching effects mask the out-of-core nature of the problem. We cannot use more than 8 GB of data per processor due to disk-space limitations. In-core columnsort sorts datasets that fit in memory—that is, each buffer represents a complete dataset to be sorted. We ran the program on one node of the

	4 GB/proc				8 GB/proc			
	static	impr.	dynamic	impr.	static	impr.	dynamic	impr.
No optimizations	933.69	—	933.69	—	1942.59	—	1942.59	—
Stage replication	905.78	2.99%	917.38	1.75%	1885.76	2.93%	1909.58	1.70%
Thread priorities	926.19	0.80%	928.39	0.57%	1921.05	1.11%	1933.62	0.46%
Both optimizations	896.46	3.99%	916.61	1.83%	1882.17	3.11%	1915.87	1.38%

Table 1: Running times (in seconds) for out-of-core columnsort with and without FG optimizations. We compare the original code without any optimizations to code that uses stage replication, thread priorities, and both optimizations together. Improvements are expressed as percentages of the unoptimized times. Because each optimization affects a different pass of columnsort, we are able to combine the two to see the best results. Each time shown is the average of three runs.

Beowulf cluster.

5.1. Stage replication and thread priorities

Table 1 shows running times, in seconds, for out-of-core columnsort. The first pass of columnsort is the most compute-bound: its sort and permute stages dominate the I/O and communicate stages. Therefore, the speedups for stage replication come from replicating the sort and permute stages in the first pass. Statically replicating those stages reduces execution time by 2.99% and 2.93% for 4 GB and 8 GB of data per processor, respectively. Relying on dynamic replication gives smaller improvements, of 1.75% and 1.70%, because several rounds must complete before the stages emerge as bottlenecks.

Adjusting thread priorities gives us little in terms of improving performance for out-of-core columnsort. The second and third passes of columnsort are I/O-bound, but the I/O cost is somewhat less dominant than in the fourth pass. Therefore, the speedups for thread priorities come from lowering the priorities of the sort and permute stages in those two passes, although the gains are negligible. For static thread priorities, we see a reduction of only 0.80% and 1.11% for 4 GB and 8 GB of data per processor. Relying on dynamic thread priorities gives us reductions of 0.57% and 0.46%.

Since columnsort benefits from stage replication and lowering thread priorities in different passes, it is easy to combine the two optimizations. We lower priorities in the second and third passes and replicate stages in the first pass to see the most dramatic results. Using both optimizations dynamically, we reduce execution time by 1.83% and 1.38% for 4 GB and 8 GB of data per processor, respectively. Using both optimizations statically, we reduce execution time by 3.99% and 3.11%.

Tables 2 and 3 show running times, in seconds, for two different versions of in-core columnsort. In-core columnsort performs a sort on each buffer, rather than on the collective data in the input file. Each buffer is 2 MB in size, fitting easily in memory. Because in-core sorting requires only one pass, we are not able to exploit both optimizations in the same program.

The first version, whose running times appear in Table 2, is compute-bound: the first stage reads data from disk, the second stage performs all the steps of columnsort, and the final stage writes the sorted buffer to disk. Its pipeline contains three stages altogether (in addition to the source and sink). It is compute-bound because we perform all computations in one single stage, which dominates the read and write stages. Without any performance optimizations, the compute-bound implementation runs in 15.22 seconds. Both static and dynamic stage replication yield speedups, with the most significant payoff occurring in the compute-bound version of columnsort. Compute-bound columnsort with static stage replication runs in 12.96 seconds, a reduction of 14.85%, and with dynamic stage replication it runs in 13.04 seconds, a reduction of 14.32%.

The I/O-bound version of in-core columnsort, whose running times appear in Table 3, performs each step of the sort in a separate stage. Its pipeline, therefore, contains ten stages (not counting source and sink): one for each step of the eight steps of columnsort and one each for read and write. It is I/O-bound because the cost of the read and write stages dominates the cost of each individual sorting stage. Without any performance optimizations, the I/O-bound version runs in 12.53 seconds. Lowering thread priorities gives us running times of 11.63 seconds for the static approach and 11.64 seconds for dynamic. These adjustments give more significant speedups of 7.18% and 7.10%.

5.2. Fork-join and DAG structures

Next we turn our attention to the structural extensions to FG. Our experimental results demonstrate that we can see performance improvements from using a fork and join or a DAG construct in appropriate circumstances. We implemented a shearing program to demonstrate the fork-join construct and a filtered FFT to demonstrate the DAG construct. The speedups result from more efficient buffer management by moving buffers directly to stages where they are needed rather than conveying every buffer to every stage.

Table 4 shows the running times for the shearing program. We compare a pipeline with a linear structure, shown in Figure 4, to a pipeline with a fork-join structure, show in Figure 5. With a linear pipeline, we see running times of 1.66, 3.28, and 6.72 seconds for 16, 32, and 64 MB of data, respectively. The corresponding times for the fork-join are 1.60, 3.11, and 6.33 seconds. The percentage improvements are 3.61%, 5.18%, and 5.80% for these data sizes.

	static	impr.	dynamic	impr.
No optimizations	15.22	—	15.22	—
Stage replication	12.96	14.85%	13.04	14.32%

Table 2: Running times (in seconds) and percentage improvements for the compute-bound implementation of in-core columnsort with and without stage replication. Running times are for 32 rounds, with a buffer size of 2 MB. Each time shown is the average of three runs.

	static	impr.	dynamic	impr.
No optimizations	12.53	—	12.53	—
Thread priorities	11.63	7.18%	11.64	7.10%

Table 3: Running times (in seconds) and percentage improvements for the I/O-bound implementation of in-core columnsort with and without thread priorities. Running times are for 32 rounds, with a buffer size of 2 MB. Each time shown is the average of three runs.

Data size (MB)	linear pipeline	fork-join	improvement
16	1.66	1.60	3.61%
32	3.28	3.11	5.18%
64	6.72	6.33	5.80%

Table 4: Running times (in seconds) for the shearing program, based on the total input size, with a linear pipeline and a fork-join construct. The improvement is for fork-join over the linear pipeline. Each time shown is the average of three runs.

	2048 rounds			4096 rounds			8192 rounds		
Discard rate	linear	DAG	improvement	linear	DAG	improvement	linear	DAG	improvement
1/10	3.69	3.61	2.17%	15.62	15.29	2.11%	66.46	65.07	2.09%
1/5	3.30	3.22	2.42%	13.93	13.57	2.58%	59.49	56.53	4.98%
1/2	2.06	1.87	9.22%	8.72	7.46	14.49%	36.99	20.95	43.36%

Table 5: Running times (in seconds) for a filtered FFT in a linear pipeline and using a DAG. A linear pipeline conveys each buffer through the entire pipeline, although it does not compute the FFT on noisy buffers. A DAG-structured pipeline discards a buffer immediately upon discovering that it is noisy. We compare 2048, 4096, and 8192 rounds; the number of elements in each buffer is identical to the number of rounds. Each time shown is the average of three runs.

Table 5 shows the running times for a pipeline that performs an FFT. We compare a pipeline with a linear structure to one built as a DAG. We assume that the data undergoes a permutation before the FFT begins computation, and so all that is left to the FFT pipeline is to compute the levels of the butterfly network. In the case of a buffer that contains noise instead of real data, we do not compute the FFT on the buffer. The linear version of the pipeline passes every buffer to every stage regardless of the content of the buffer. The DAG version conveys a buffer from the read stage to the sink stage if it contains noisy data. We compare running times for 2048, 4096, and 8192 rounds. The number of elements in each buffer is identical to the number of rounds: hence, by doubling the rounds we quadruple the total data size and, due to the structure of FFT, add one stage to the pipeline.

Using the DAG structure to discard a buffer affords a significant improvement over the linear pipeline. When we execute 2048 rounds, the DAG structure is faster than the linear pipeline by 2.17%, 2.42%, and 9.22%, for discard rates of 1/10, 1/5, and 1/2, respectively. When we perform 4096 rounds, the DAG structure beats the linear pipeline by 2.11%, 2.58%, and 14.49%. The improvement is most dramatic for 8192 rounds, resulting in 2.09%, 4.98%, and 43.36% speedups over a linear pipeline.

6. Conclusion

This paper introduced extensions to FG of two flavors: optimizations to improve performance time and enhancements to broaden the scope of FG. We used stage replication and thread priorities, both statically and dynamically, to improve running times for some programs written with FG. We added fork-join and DAG constructs to make FG more flexible, and we found that those extensions can improve running time as well.

The performance improvements we have seen throughout this paper are modest, but they come with little or no effort by the programmer. Statically replicating a stage or lowering the priority of a thread requires only one additional line of code; employing either optimization dynamically requires no additional code at all. Moreover, the benefits of these optimizations compound. The performance gains that come from fork-join and DAG constructs are similarly modest, but these extensions are structural. Although we introduced them to broaden the scope of FG, they also provide the benefit of running faster than linear pipelines.

6.1. Future work

We expect that future implementations of FG will use even more performance enhancements, such as dynamic buffering. The number and size of buffers in a pipeline can impact performance, and we envision a version of FG that dynamically determines the optimal combination of number and size and adjusts the buffers accordingly.

Any high-performance application that works with massive data and adheres to a pipeline, fork-join, or DAG structure would be well-suited to FG. We expect to implement in FG relevant applications from areas such as imaging, computational biology, and seismic modeling. We hope that porting such applications from their original implementations to FG will yield faster running times.

6.2. Acknowledgments

Geeta Chaudhry wrote the columnsort program and advised us when we converted it to use FG. Brunn Roysden, as part of his senior honors thesis, investigated thread priorities in Linux. Tim Tregubov set up our Beowulf cluster and answered myriad questions. Finally, we thank MPI Software Technology, Inc., for their assistance in installing and using ChaMPIon/Pro.

References

[BC99] Lauren M. Baptist and Thomas H. Cormen. Multidimensional, multiprocessor, out-of-core FFTs with distributed memory and parallel disks. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 242–250, June 1999.

[CC02] Geeta Chaudhry and Thomas H. Cormen. Getting more from out-of-core columnsort. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX 02)*, pages 143–154, January 2002.

[CCW01] Geeta Chaudhry, Thomas H. Cormen, and Leonard F. Wisniewski. Columnsort lives! An efficient out-of-core sorting program. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 2001.

[CD04] 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.

[Cha04] Geeta Chaudhry. *Parallel Out-of-Core Sorting: The Third Way*. PhD thesis, Dartmouth College, 2004.

[CSW99] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1):105–136, 1999.

[CWN97] Thomas H. Cormen, Jake Wegmann, and David M. Nicol. Multiprocessor out-of-core FFTs with distributed memory and parallel disks. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS '97)*, pages 68–78, November 1997.

[DC] Elena Riccio Davidson and Thomas 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/.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[SI86] Kazuhiro Sado and Yoshihide Igarashi. Some parallel sorts on a mesh-connected processor array and their time efficiency. *Journal of Parallel and Distributed Computing*, 3(3):398–410, September 1986.

[SSS86] Isaac D. Scherson, Sandeep Sen, and Adi Shamir. Shear sort: A true two-dimensional sorting technique for VLSI networks. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 903–908, August 1986.

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[VS94] 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.