FG: A Framework Generator for Hiding Latency in Parallel Programs Running on Clusters

Thomas H. Cormen*
Elena Riccio Davidson †

Dartmouth College Department of Computer Science
6211 Sudikoff Laboratory
Hanover, NH 03755
{thc, laney}@cs.dartmouth.edu

Abstract

FG is a programming environment for asynchronous programs that run on clusters and fit into a pipeline framework. It enables the programmer to write a series of synchronous functions and represents them as stages of an asynchronous pipeline. FG mitigates the high latency inherent in interprocessor communication and accessing the outer levels of the memory hierarchy. It overlaps separate pipeline stages that perform communication, computation, and I/O by running the stages asynchronously. Each stage maps to a thread. Buffers, whose sizes correspond to block sizes in the memory hierarchy, traverse the pipeline. FG makes such pipeline-structured parallel programs easier to write, smaller, and faster.

FG offers several advantages over statically scheduled overlapping and dynamically scheduled overlapping via explicit calls to thread functions. First, it reduces coding and debugging time. Second, we find that it reduces code size by approximately 15–26%. Third, according to experimental results, it improves performance. Compared with programs that use static scheduling, FG-generated programs run approximately 61–69% faster on a 16-node Beowulf cluster. Compared with programs that make explicit calls for dynamically scheduled threads, FG-generated programs run slightly faster. Fourth, FG offers various design options and makes it easy for the programmer to explore different pipeline configurations.

*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.

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

1 Introduction

Latency is a significant impediment to producing high-performance parallel programs. Storage components farther from the processor have significantly higher capacities, but their access latencies increase greatly. Additionally, latency results from interprocessor communication on a cluster. This paper presents FG, a framework generator for pipeline-structured computations on clusters that mitigates latency—that is, it hides a portion of the latency by overlapping high-latency operations with other operations—and also reduces the burden on the programmer.

FG arose from prior work in out-of-core programming on clusters. In out-of-core programs, the amount of data exceeds the capacity of main memory and so must reside on disks. Disk accesses are 4–5 orders of magnitude slower than main-memory accesses, and interprocessor communication is also quite a bit slower than in-processor computation. Hence, moving data between memory and disk and between two different memories in the cluster are high-latency operations. In order to achieve high performance, therefore, out-of-core programs must accomplish two goals: (1) overlap communication, computation, and I/O; and (2) amortize the high latency of memory transfers and communication by moving data in blocks.

Several out-of-core algorithms in the literature (e.g., [3, 10, 11, 14, 15]) make multiple passes over the data. In many of these algorithms, each pass falls into a common framework: a pipeline of stages that may be run asynchronously. Buffers, corresponding to blocks of data in the memory hierarchy, traverse the pipeline. For example, in one implementation of out-of-core columnsort [4], the pipeline for one pass has the five stages shown in Figure 1. A buffer enters the pipeline and then, in order, the stages read into it from disk, sort it, perform interprocessor communication with it, permute it, and finally write it back to disk. Because each stage works on a single buffer, each

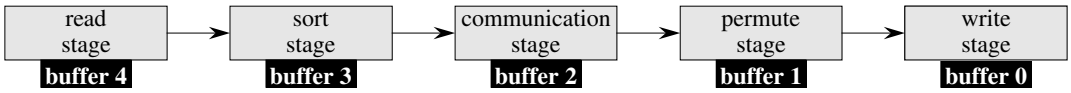

Figure 1: The stages of the pipeline for one pass of out-of-core columnsort. Each stage accepts a buffer from its predecessor, operates on the buffer, and conveys it to its successor. Stages work asynchronously so that each one may be working on a distinct buffer (indicated by a black rectangle attached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

individual stage may be working on a distinct buffer at any moment in time. Thus arises the opportunity to overlap the actions of the individual stages, and in doing so, to mitigate the latency of the read, communicate, and write stages.

Producing asynchronous programs for the pipeline framework requires a great deal of time and effort on the part of the programmer and a substantial body of code. One way to achieve asynchrony is to use explicit asynchronous calls for disk I/O and interprocessor communication. Writing programs with asynchronous calls is particularly burdensome, due to the programmer's having to preschedule work to perform while waiting for I/O or communication and then inserting blocking waits in the appropriate places. Moreover, because the scheduling is static, it cannot adapt to changing workloads in the various stages.

Another way to achieve asynchrony in the pipeline framework is to locate each pipeline stage within its own thread. Even more parallelism results if the cluster has an SMP in each node, as we see in many high-end clusters; then the threads can run in parallel. With explicit threaded code, however, the programmer must write the code to create, run, and kill the threads; to create, use, and destroy the semaphores needed to coordinate the threads; and to allocate, manage, and deallocate the buffers that traverse the pipeline. The unfortunate consequence is that a sizable portion of code is *glue*, which is unrelated to the algorithm that the programmer is implementing. We estimate that glue accounts for approximately 15–25% of the code for a typical out-of-core program. The glue is often the most difficult code to write and debug.

We have designed a programming environment that mitigates latency by means of overlapping communication, computation, and I/O for programs with a pipeline structure that run on clusters. The stages of the pipeline run asynchronously in threads, and they operate on buffers corresponding to blocks of data. In this environment, the programmer specifies the stages of the pipeline framework in order, and the environment generates the glue and runs the pipeline. Hence, we call this environment Asynchronous Buffered Computation Design and Engineering Framework Generator (ABCDEFG), or FG for short. In our experience, FG reduces source code size by approximately 15–26%. In addition, FG allows for easy experimentation. There are several factors that may influence the performance of a pipeline-structured program, and tuning these factors is simple in FG, often entailing a line or two of code changes rather than a complete redesign. A further benefit of FG is that it improves performance.

The remainder of this paper is organized as follows. In Section 2, we present an overview of FG. Section 3 describes some additional features of FG beyond its basic structure. Section 4 presents experimental results that show how FG both reduces code size and improves performance. Finally, Section 5 offers some concluding remarks.

2 FG's basic features

FG makes it easy for the programmer to define and run a pipeline of asynchronous stages. The programmer provides the individual stages, and FG provides the glue and manages buffers. This section describes the basic functionality of FG.

Stages and buffers

FG's paradigm is a pipeline of stages. The programmer writes each stage as a C or C++ function, and FG runs the stages asynchronously to overlap communication, computation, and I/O. FG manages buffers as they traverse the pipeline, and it also allocates and deallocates the buffers. FG accomplishes asynchrony and mitigates latency by mapping each stage to a thread.

With FG, the programmer is relieved of most of the burden of making an out-of-core program run asynchronously, including making explicit calls to functions in the pthreads package. The programmer writes each stage as a synchronous function. That is, no asynchronous calls for I/O, communication, or anything else are needed. The programmer defines a simple, opaque object for each stage and each thread and then associates each stage object with a thread object. FG takes care of the rest. It creates and spawns each thread, executes the function associated with each stage, and manages semaphores for buffer transmission. FG also performs all the work associated with the cleanup of a program, such as joining and killing threads, destroying semaphores, and deallocating buffers. Without FG, all this work would be the programmer's burden.

FG manages buffers as they traverse the pipeline, alleviating the programmer of the associated work. FG allocates buffers of a fixed size at the start of execution. Each buffer corresponds to the block size for the outermost level that will be accessed in the memory hierarchy. Queues hold buffers as they progress between stages; that is, a separate queue sits between each pair of consecutive stages. A stage conveys a buffer by placing it in a queue, and its successor stage accepts the buffer by removing it from the same queue. The queue implementation is lock-free. Any given stage need not know what its predecessor and successor stages are, since FG will ensure that the buffers move appropriately.

FG keeps the information about a buffer in a *thumbnail*. We shall see some of the information stored in thumbnails later on, but for now we mention that a thumbnail contains a pointer to its corresponding buffer. That is, there is a one-to-one mapping of thumbnails and buffers. It is actually the thumbnails that populate the queues.

Source and sink stages

In addition to the basic pipeline, FG adds some elements to make programs work properly. To help manage buffers, FG includes a source stage at the head of the pipeline and a sink stage at the tail. The source stage collects the buffers at the start of execution and passes them along to the first programmer-defined stage. Each time the source stage emits a buffer, we say that another *round* starts, and the source stage places the next round number into the buffer's thumbnail. A stage may need to refer to a thumbnail's round number; for example, a stage that reads blocks from disk may perform a seek based on the round number and buffer size.

The sink stage works with the source stage to recycle buffers. The number of rounds is typically far greater than the number of stages in the pipeline. Although we could allocate each buffer in the source stage and deallocate it in the sink stage, doing so could cause performance problems in the interaction between allocation and deallocation. Instead, when a buffer reaches the end of the pipeline, the sink stages recycles it back to the source stage, which then resends the buffer back into the pipeline but now with a new round number.

Figure 2 shows how FG puts all of the above notions together to overlap stages in a pipeline. Here, we have three programmer-defined stages plus the source and sink. For simplicity, this example assumes that each stage takes the same amount of time. Once the pipeline is in full swing, multiple stages can execute concurrently but on different buffers. The sink recycles thumbnails and buffers back to the source, where they re-enter the pipeline but with new round numbers in the thumbnails.

step	source	stage 1	stage 2	stage 3	sink
1	buffer 0 round 0	–	–	–	–
2	buffer 1 round 1	buffer 0 round 0	–	–	–
3	buffer 2 round 2	buffer 1 round 1	buffer 0 round 0	–	–
4	–	buffer 2 round 2	buffer 1 round 1	buffer 0 round 0	–
5	–	–	buffer 2 round 2	buffer 1 round 1	buffer 0 round 0
6	buffer 0 round 3	–	–	buffer 2 round 2	buffer 1 round 1
7	buffer 1 round 4	buffer 0 round 3	–	–	buffer 2 round 2
8	buffer 2 round 5	buffer 1 round 4	buffer 0 round 3	–	–

Figure 2: The progression of three buffers through a pipeline over time. The pipeline has three programmer-defined stages—stage 1, stage 2, and stage 3—as well as the FG-supplied source and sink. Each buffer's thumbnail has the number of a round, and the sink recycles buffers back to the source. Here, we assume that each stage takes the same amount of time. Multiple stages can execute concurrently.

Multistage threads

FG allows a thread to contain multiple stages, a design that we call a *multistage thread*. We implemented this feature for two reasons. First, if there are fewer buffers than threads, there would always be some thread making no progress because it is waiting for a buffer to arrive. If some thread has many stages, we can build a pipeline with the same number of stages but fewer threads. Thus, without increasing the number of buffers, we can keep all threads busy. Second, there may be stages that cannot run simultaneously. For example, suppose that a pipeline has one stage that reads from disk and another stage that writes to the same disk. Of course, we cannot read and write the same disk at the same time. A programmer might, therefore, choose to map the read stage and the write stage to the same thread. This setup means that the read and write stages will run synchronously without the overhead of spawning two separate threads and context-switching between them.

The initial design for FG allowed for the stages of a multistage thread to execute in any order, even one that is randomly chosen. We quickly found a serious problem that arose from this flexibility. Suppose that two stages of a multistage thread perform interprocessor communication, such as a broadcast or a scatter. Let the processors be A and B, and let the stages be X and Y. Suppose that pro-

step	source	read	sort	write	sink
1	0	–	–	–	–
2	1	0	–	–	–
3	1	–	0	–	–
4	1	–	–	0	–
5	2	1	–	–	0
6	2	–	1	–	–
7	2	–	–	1	–
8	3	2	–	–	1

(a)

step	source	read	sort	write	sink
1	0	–	–	–	–
2	1	0	–	–	–
3	2	1	0	–	–
4	2	–	1	0	–
5	2	–	–	1	0
6	3	2	–	–	1

(b)

Figure 3: Multistage repeat. The read and write stages are in the same thread. The figure shows which rounds are in which stages at each step. Shaded boxes indicate which of the read and write stages runs next. **(a)** A multistage repeat of 1. The read stage executes once and does not execute again until the write stage finishes with the most recent buffer read. The buffer for round 1 does not get out of the source stage until step 5, and altogether it takes 8 steps to get two buffers through the pipeline. **(b)** A multistage repeat of 2. The buffer for round 1 leaves the source stage at step 3, and altogether it takes only 6 steps to get two buffers through the pipeline.

cessor A chooses to execute stage X and that processor B chooses to execute stage Y. Stage X does not complete on any processor until all processors involved in the communication run stage X, and the same is true for stage Y. Therefore, processor A waits for processor B to run stage X, and processor B waits for processor A to run stage Y. In other words, the two processors deadlock.

Rather than implementing mechanisms for deadlock detection and recovery, FG requires that the stages of a multistage thread run in the same order on all processors. The most straightforward scheme would be to run each stage once in succession. That is, if stage X precedes stage Y in our previous example, then our multistage thread would run X then Y on one buffer, then it would run X then Y on the next buffer, and so on, until the pipeline completes.

This restricted approach may slow down the pipeline, however. Consider the pipeline in Figure 3(a). This pipeline has a read stage as its first stage, a write stage as its last, and these two stages reside in the same thread. As Figure 3(a) shows, after the read stage has read into the buffer for round 0, it cannot start to read into the buffer for round 1 until round 0's buffer has gone the entire length of the pipeline. It takes 8 steps to get two buffers all the way through the pipeline, and the buffer for round 1 does not get out of the source stage until step 5. This scheme results in a pipeline that is often underfull and consequently doesn't overlap work as much as it could. Therefore we introduce the notion of multistage repeat.

Multistage repeat means that each stage of a multistage thread may run several times before another stage in the thread gets to run. Figure 3(b) shows the same pipeline as in Figure 3(a), but with a multistage repeat of 2. Observe that now the buffer for round 1 can leave the source stage in step 3, and altogether it takes only 6 steps to get two buffers all the way through the pipeline.

Flexibility

An important feature of FG is that it allows the programmer to explore changes in the design of the pipeline. For example, the programmer might wish to put two stages in a multistage thread, split a multistage thread into separate threads, or increase the multistage repeat to determine whether it yields a performance improvement. There are also design changes that affect the structure of the pipeline, such as adding or removing a stage. Moreover, the programmer may wish to change the number or size of the buffers. Without FG, implementing these changes means quite a bit of redesigning and debugging for the programmer. Furthermore, the payoff from a given change may be relatively low compared to the effort required, though after exploring enough options, the programmer may find a particularly good configuration. Since FG provides the glue, however, the programmer need rewrite only a few lines of code to try out each option.

3 Additional features

The structure of FG outlined above does not tell the whole story. In this section we present two additional features of FG. First we present buffer swapping, which simplifies stages that cannot do work in-place. Then we examine pipeline macros, which allow one pipeline to be plugged into another. Due to space limitations, we omit discussion of some other FG features: the caboose (a special flag attached to the last buffer to go through the pipeline), thread initialization and cleanup functions, and barriers.

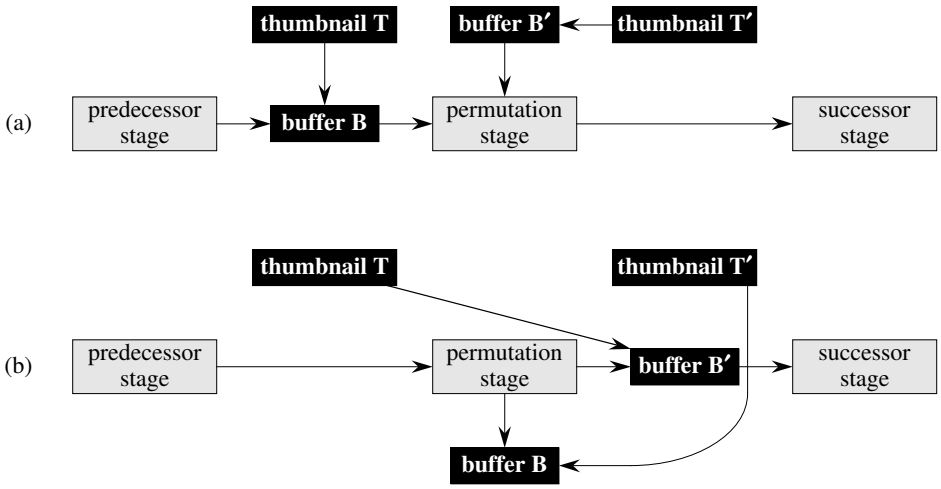

Figure 4: Buffer swapping. **(a)** A permutation stage receives from its predecessor a pipeline buffer B pointed to by thumbnail T, and it requests auxiliary buffer B′ pointed to by auxiliary thumbnail T′. **(b)** The permutation stage permutes the contents of B into B′, and it swaps the pointers to B and B′ in the thumbnails. Buffer B′ is now pointed to by thumbnail T and is sent to the successor stage. Auxiliary thumbnail T′ points to buffer B, and they are released.

Buffer swapping

FG allows for buffers to be swapped within a stage. Consider a stage whose work cannot be done in-place, such as the permutation stage in Figure 4, which requires a source buffer and a target buffer. In Figure 4(a), the source buffer B arrives through the pipeline from the stage's predecessor in thumbnail T. This stage must request an additional buffer to assume the role of the target buffer. The stage, therefore, can ask FG for an auxiliary buffer. An *auxiliary buffer* is similar to an ordinary buffer, but its thumbnail does not traverse the pipeline. Figure 4(a) shows the auxiliary buffer B′, which is pointed to by auxiliary thumbnail T′. The permutation stage copies the contents of buffer B into buffer B′, permuting the data along the way. Now, as Figure 4(b) shows, it is buffer B′, not B, that the permutation stage needs to convey to its successor. The pipeline thumbnail T, however, has information in it that we do not want to lose, including its round number and the caboose flag. In order to keep this information, we simply swap the addresses of the two buffers in their thumbnails. Now the pipeline thumbnail T points to buffer B′, which contains the result of the permutation, and the auxiliary thumbnail T′ points to buffer B, whose contents are no longer needed. The stage, then, can release thumbnail T′ and buffer B so that another stage can use them. The pipeline thumbnail T and the buffer B′ can make their way to the next stage.

Pipeline macros

FG includes the ability to plug one pipeline into another, a structure that we call a *pipeline macro*. This mechanism is useful for a stage that expands into a pipeline. For example, one stage of an FFT pipeline would typically be a bit-reversal permutation, which on its own might require several stages. When designing the pipeline for the FFT, the programmer can designate one stage as the bit-reversal stage, and FG substitutes the various stages that accomplish the bit-reversal permutation. Macro expansion can occur in any stage of any pipeline, giving rise to a tree structure. The frontier of the tree corresponds to the pipeline stages actually executed.

4 Experimental results

In this section, we show that FG makes parallel, asynchronous programs easier to write, smaller, and faster. For benchmarks we use programs operating on massive data on a distributed-memory cluster with a disk at each node. Such programs are rare, however. The other current systems of which we are aware, such as TPIE [1], operate on just one processor with one disk. Therefore, we focus our FG comparison against previous Dartmouth out-of-core programs written for clusters. We also compare FG to UNIX pipes.

ViC* [7] was a software system designed to adapt programs written in C* for massive datasets, and it was the focus of out-of-core programming at Dartmouth starting in 1994. ViC* used static scheduling to mitigate la-

Program	Code size			Seconds: 4 GB/proc			Seconds: 8 GB/proc		
	non-FG	FG	improvement	non-FG	FG	improvement	non-FG	FG	improvement
BMMC	3204	2736	14.6%	1049	327	68.8%	1844	570	69.1%
FFT	7612	6290	17.4%	1996	722	63.8%	4245	1638	61.4%
Columnsort	7824	5820	25.6%	1029	979	4.9%	1893	1862	1.6%

Table 1: Code size and running times in seconds for three programs with and without FG. Each time shown is the average of three runs. The non-FG programs for BMMC permutations and FFT use static scheduling. The non-FG columnsort program uses dynamic, thread-based scheduling.

tency, and it overlapped only I/O, not communication or computation. We compare FG with two ViC* programs, Fast Fourier Transform (FFT) [9, 11] and bit-matrix-multiply/complement (BMMC) permutations [8]. Following ViC*, researchers at Dartmouth made the pipeline observation from Section 1 and moved to dynamic scheduling with threads [4, 5, 6]. We compare FG with one threaded program, columnsort [12].

We present experimental results on a Beowulf cluster in terms of two metrics: source code size and running time. As Table 1 shows, FG both reduces code size and improves performance. We compare against programs written for ViC*'s static scheduling and against a program written for dynamic thread-based scheduling. Compared with the static scheduling of ViC*, FG-based programs required approximately 15–17% fewer source lines, and they ran approximately 61–71% faster. Compared with the more robust, dynamic, thread-based scheduling, the FG-based program required approximately 26% fewer source lines, and it ran approximately 2–5% faster. We were unable to measure "time to solution," but we plan to do so in our future work.

The programs that use static scheduling in their non-FG versions are for performing out-of-core BMMC permutations and computing out-of-core FFTs. The program that uses dynamic, thread-based scheduling in its non-FG version is one of our out-of-core implementations of columnsort.

Source code size

As Table 1 shows, using FG reduces the number of lines of source code. Compared with source code that uses static scheduling, source code that makes FG calls has 14.6% fewer lines in the BMMC-permutation program and 17.4% in the FFT program. The reduction in source code size is even better—25.6%—for the out-of-core columnsort program, which uses dynamic, thread-based scheduling.

The code that uses static scheduling has to do all of the following explicitly: read and write buffers asynchronously, perform the appropriate blocking waits, and start up and shut down the pipeline. When FG performs these functions, that source code is omitted.

The program that uses dynamic, thread-based scheduling has to do even more than its static counterparts. Specifically, it has to do all of the following and do it in each stage: spawn the thread associated with the stage; allocate the necessary buffers; create the semaphores; accept a buffer from the previous stage; pass the buffer to the next stage; and, when the pipeline completes, join and kill the thread, destroy the semaphores, and deallocate the buffers. With FG, the programmer has none of these responsibilities. Instead the programmer just needs to associate each stage with a thread; FG shoulders the remaining tasks.

Running time

Reducing code size is pointless if programs written with FG do not perform at least as well as those written without it. Indeed, we find that FG improves performance.

We ran experiments on a Beowulf cluster of 16 dual 2.8-GHz Intel Xeon 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. Communication occurs via MPI calls. We use the ChaMPIon/Pro implementation of MPI because it works with the Myrinet network and it allows multiple threads to make MPI calls. (We know of no free versions of MPI that match what ChaMPIon/Pro can do.)

We report here on experiments using all 16 nodes and either 4 GB or 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.

As Table 1 shows, FG yields huge improvements over static scheduling. The improvement for FFT exceeds the improvement for BMMC permutations because the FFT program performs relatively less communication than the BMMC-permutation program. Hence, the improvement due to FG's ability to overlap communication with other functions is less. We believe that the advantage of using FG for the FFT program diminishing with more data is due to FFT's $O(n \lg n)$-time computation component, which FG

		FG times			UNIX pipe times			ratio: FG / UNIX pipe		
Program	Data source	User	System	Total	User	System	Total	User	System	Total
Columnsort	file	23.83	0.83	12.53	22.97	2.28	12.93	1.04	0.37	0.97
Columnsort	memory	18.08	0.09	9.27	19.16	1.58	10.66	0.94	0.06	0.87
FFT	file	45.47	11.67	29.43	41.55	88.89	66.20	1.09	0.13	0.45
FFT	memory	28.98	0.72	14.85	19.50	52.42	36.03	1.49	0.01	0.41

Table 2: Running times, in seconds, for columnsort and FFT programs using FG and UNIX pipes. Each time is the average of three runs on a single node of the Beowulf cluster. Times are given according to the tcsh `time` command's breakdown of user, system, and total time. Total times are approximately half the sum of user and system times because the node has two processors.

may be less able to hide as n increases. (For the BMMC-permutation program, the improvements due to using FG in both the 4-GB and 8-GB per processor cases are similar because the computation time, as well as the communication and I/O times, is linear in the problem size.)

We also see from Table 1 that FG yields a modest improvement over dynamic, thread-based scheduling. How can this be when FG itself uses dynamic, thread-based scheduling? We believe that FG's advantage comes from its ability to manage queues of several buffers between stages, compared to the non-FG code, which allows only one buffer at a time between stages. Like FFT, the columnsort program has an $O(n \lg n)$-time computation component, and so the benefit of using FG diminishes for extremely large problem sizes.

That FG yields only a small improvement over non-FG code in the dynamic thread-scheduling case is of no great concern. What is important is that FG is competitive with, and even beats, hand-tuned code that is difficult to write.

FG vs. UNIX pipes

Like FG, UNIX pipes [2, Section 5.12] act on data streams. UNIX pipes transfer buffers between processes, however, and not between threads. Hence, they are a more heavyweight mechanism than the queues that FG uses. The implementations are similar, except that UNIX pipes require memory to be shared between processes. Moreover, because the programmer accesses UNIX pipes as if they were files, the implementation of pipes also needs to interact with the file-system interface. Finally, although it is possible to have multistage processes with UNIX pipes, it is much more convenient to use FG's multistage thread mechanism.

We implemented a small suite of pipeline-structured programs using FG and using UNIX pipes to compare their running times. The programs performed columnsort and FFT. All computations were on buffers that fit in memory, but some programs generated the data in memory whereas others read data from disk and wrote the results back to disk. The programs performed no interprocessor communication and ran on one node of the Beowulf cluster. We obtained user, system, and total times via the built-in tcsh `time` command.

Table 2 summarizes the results. The FG implementation ran faster than the UNIX pipe version in all cases. Indeed, FG took under half the time of UNIX pipes for the FFT programs. The pipeline for the FFT programs has 16 stages, 13 of which compute levels of the FFT butterfly graph. The UNIX pipe FFT programs spend much system time synchronizing pipes and copying buffers between processes. The columnsort programs have only 7 stages, and so the relative inefficiency of UNIX pipes is not quite so pronounced.

5 Conclusion

We conclude by summarizing FG's benefits, discussing some related work, and looking toward FG's future.

FG provides a programming environment for asynchronous programs that run on clusters and fit into a pipeline framework. It mitigates latency by overlapping communication, computation, and I/O. FG makes such programs easier to write, smaller, and faster. Programs written with FG run approximately 61–69% faster than equivalent programs with static scheduling that do not overlap communication with other operations. Moreover, programs written with FG run approximately 2–5% faster than equivalent programs that explicitly use threading to achieve asynchrony.

Perhaps the closest match to FG is the TPIE project at Duke [1]. TPIE provides the abstraction of streams of out-of-core data, where each stream is a C++ object. Object methods perform I/O and handle asynchrony in the I/O operations. TPIE provides a lower-level interface that allows the programmer to read and write blocks of data explicitly. It also provides methods to merge and partition streams. To the best of our knowledge, TPIE runs only on a single processor and with a single disk.

StreamIt [13] is a Java-like language that allows the programmer to manipulate in-core streams of data. The lan-

guage is architecture-independent, and the StreamIt compiler and run-time system run the stream-based program efficiently on the target architecture. StreamIt allows fork/join and loop constructs within streams. We plan to investigate how to incorporate these constructs into FG.

We plan for future versions of FG to include the capabilities to monitor performance and make adjustments on the fly. Although the current version of FG makes performance tuning easy, we wish to eliminate the programmer's need to experiment with various combinations of parameters to see how they affect performance. This enhancement would relieve the programmer of some of the burden related to performance tuning. It would be up to FG, and not to the programmer, to make such changes as altering the number of threads or the mapping of stages to threads. Many factors may have an impact on performance, and we plan for FG to tune these factors dynamically without the programmer's having to rewrite any code at all.

Finally, we would like to determine how much FG reduces time-to-solution. We plan to conduct controlled experiments to quantify FG's productivity benefits. One obstacle is finding programmer subjects who have sufficient experience writing threaded programs.

Acknowledgments

Elizabeth Hamon produced an early implementation of FG and helped us find several flaws in our original design. Geeta Chaudhry wrote the columnsort program and advised us when we converted it to use FG. 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

[1] Lars Arge, Rakesh Barve, David Hutchinson, Octavian Procopiuc, Laura Toma, Darren Erik Vengroff, and Rajiv Wickremesinghe. *TPIE User Manual and Reference*. Department of Computer Science, Duke University. Draft of August 29, 2002.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[3] 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.

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

[5] 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.

[6] Geeta Chaudhry, Thomas H. Cormen, and Elizabeth A. Hamon. Parallel out-of-core sorting: The third way. *Cluster Computing*. To appear.

[7] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dartmouth College Department of Computer Science, November 1994.

[8] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the Parallel Disk Model with the ViC* implementation. *Parallel Computing*, 23(4–5):571–600, June 1997.

[9] Thomas H. Cormen and David M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing*, 24(1):5–20, January 1998.

[10] 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.

[11] 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.

[12] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[13] StreamIt Language Specification, Version 2.0. http://www.cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf, February 2003.

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

[15] 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.