The FG Programming Environment: Reducing Source Code Size for Parallel Programs Running on Clusters

Elena Riccio Davidson *
Thomas H. Cormen†

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

Abstract

FG is a programming environment designed to reduce the source code size and complexity of out-of-core programs running on clusters. Our goals for FG are threefold: (1) make these programs smaller, (2) make them faster, and (3) reduce time-to-solution. In this paper, we focus on the first metric: the efficacy of FG for reducing source code size and complexity. We designed FG to fit programs, including high-end computing (HEC) applications, for which hiding latency is paramount to designing an efficient implementation. Specifically, we target out-of-core programs that fit into a pipeline framework. We use as benchmarks three out-of-core implementations: bit-matrix-multiply/complement (BMMC) permutations, fast Fourier transform (FFT), and columnsort. FG reduces source code size by approximately 14–26% for these programs. Moreover, we believe that the code FG eliminates is the most difficult to write and debug.

1. Introduction

In this paper, we demonstrate that our programming environment, called ABCDEFG (FG for short) [9], reduces source code size for out-of-core implementations of bit-matrix-multiply/complement (BMMC) permutations, fast Fourier transform (FFT), and columnsort. Replacing each of these C and C* programs by a comparable program written with FG saves 468, 1322, and 2004 lines of source code, respectively. These reductions amount to percentage decreases of 14.6%, 17.4%, and 25.6% of the source-code lines, respectively.

The high-end computing (HEC) applications on which we focus are *out-of-core* programs running on clusters. In an out-of-core program, the amount of data exceeds the capacity of main memory, and therefore data must reside on disk. Performing disk I/O is a high-latency operation, and so in order to achieve a high-performance implementation, it is essential to hide latency in these programs. We take two separate but related approaches to hide latency. First, we must overlap work. Since we often use disk I/O and interprocessor communication, we can overlap these two types of operations with computation on the CPU. Second, we must use buffers to access data. A buffer is simply a block of memory; in our programs, we read into and write from buffers in order to amortize the cost of transferring data among levels of the memory hierarchy. The pairing of writing asynchronous code to overlap work and using buffers to access data effectively hides latency in HEC parallel programs. We call the code for creating asynchrony and managing buffers *glue*.

Each of the three programs that we focus on in this paper fits into a pipeline framework. For example, Figure 1 illustrates the pipeline structure we use for our implementation of out-of-core columnsort. The pipelines in each of the three programs contain a stage that reads from disk, a stage that writes to disk, a stage that performs interprocessor communication, and one or more stages that perform computation. To introduce asynchrony into our programs, we overlap work by running the stages of each pipeline concurrently. Buffers travel from stage to stage; every stage may be working on a distinct buffer simultaneously.

Every time a buffer travels the length of the pipeline, we say that one *round* of execution has completed. Since we are in an out-of-core setting, we expect that the number of rounds demanded by a program far exceeds the number of buffers that can fit in memory. Therefore, we must reuse buffers after they travel the length of the pipeline. We use a global pool for buffers; we store free buffers in the pool after we initially allocate them, and we return each buffer to the pool whenever it completes a round.

Hiding latency by way of creating asynchrony and man-

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

†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.

Figure 1: An implementation of out-of-core columnsort, represented as a pipeline. The first stage reads data from disk into a buffer. The second stage performs a local sort. The third stage performs interprocessor communication. The fourth stage performs a local permutation. The final stage writes data from the buffer to disk. The stages run concurrently so that, at any moment, each stage may be working on a distinct buffer.

aging buffers is a difficult task. Without FG, it means the programmer must produce a considerable amount of glue in addition to the code required to implement the algorithm. We define as *base code* any code that is not the glue; essentially, the base code is what the program would be without any attempt to overlap. The base code does not change significantly between FG and non-FG programs. With FG, however, the programming environment makes it much easier to incorporate the glue, thus reducing the source code size and complexity of out-of-core programs.

The remainder of this paper is organized as follows. Section 2 discusses two former methods for writing pipeline-structured programs: the ViC* system and C programming with threads. Section 3 is a brief description of the FG programming environment. Section 4 presents the three out-of-core applications we use as benchmarks for comparing FG and non-FG code. Section 5 analyzes the reductions in code size and complexity that we have achieved with FG. Finally, Section 6 offers some concluding remarks.

2. Previous approaches

In this section, we present two prior approaches that researchers at Dartmouth have taken to writing out-of-core programs that fit into a pipeline framework. ViC* [3] was a software system, developed during the period 1992–2001, that adapted C* programs for massive datasets. ViC* used static scheduling to introduce asynchrony, overlapping I/O with communication and computation. It was too difficult, however, to overlap communication with computation in the ViC* framework. Starting in 2001, we moved to programming pipeline-structured out-of-core applications in C with threads [1]. Threads use dynamic scheduling, and so we were able to overlap all three of communication, computation, and I/O. The programmer was responsible for coordinating all the actions associated with threads, however. With both approaches, the programmer was responsible for writing the code that managed buffers.

2.1. ViC*

ViC* was a compiler and run-time system, and it was the focus of out-of-core programming at Dartmouth starting in 1992. We implemented two significant out-of-core programs in ViC*, namely bit-matrix-multiply/complement (BMMC) permutations [5, 7] and fast Fourier transform (FFT) [8].

ViC* overlapped only I/O with other operations, and it used static scheduling to do so. That is, in order to enjoy even the partial overlapping of asynchronous I/O, the programmer had to produce the code that scheduled the I/O operations. Writing asynchronous I/O is far more complex than writing synchronous I/O. Figure 2 illustrates a simplified example of using asynchronous and synchronous I/O within an out-of-core permutation. The asynchronous version is more efficient than the synchronous version, because it uses a blocking wait to perform the in-core permutation while waiting for the disk I/O to complete, whereas the synchronous version first reads, then performs the in-core permutation, then writes. It is clear, however, that the synchronous version is much simpler to code.

Furthermore, in ViC*, the programmer was responsible for all aspects of buffer management, a task made more complicated by the presence of asynchronous I/O. The programmer had to allocate and deallocate buffers, store them in a global pool, keep track of which buffers were free, and recycle buffers that had traversed the entire pipeline. In addition to these primary buffers, the programmer was responsible for allocating and maintaining secondary buffers, used for in-core permutations. Although any permutation can be done in-place, it is often simpler to use distinct source and target buffers. With ViC*, the programmer had to allocate designated secondary buffers to use as target buffers, ensure that a particular one was free to use in a permutation, and, after the permutation, release the secondary buffer so that it could be used again.

2.2. Threaded programming

After the ViC* project, the focus of out-of-core computing at Dartmouth turned to writing C code using threads. We used standard POSIX threads [10] to overlap I/O as well as communication and computation, and so we were able to take advantage of the dynamic scheduling inherent in the pthreads package. With dynamic scheduling, any thread that is ready can run when the CPU becomes available. Moving from ViC* to threads meant that overlapping work for asynchrony no longer necessitated writing large amounts of code for statically scheduled asynchronous I/O.

```
b = 0
start read into buffer [b, 0]
while some read has not been started do
    wait for read into buffer [b, 0]
    if not first or second time through
        then wait for write to complete from target buffer [b, 1]
    if not working on the final buffer
        then start read into buffer [1 − b, 0]
    permute (in-core) from read buffer [b, 0] into target buffer [b, 1]
    start write of target buffer [b, 1]
    b = 1 − b
wait for writes to complete from target buffers [0, 1] and [1, 1]
```

(a)

```
while some read has not been started do
    read into buffer 0
    permute (in-core) from read buffer 0 into target buffer 1
    write target buffer 1
```

(b)

Figure 2: An out-of-core permutation using asynchronous and synchronous I/O operations. **(a)** Using asynchronous I/O. While waiting for a read or write to complete, we can begin the in-core permutation, but we must schedule it statically. **(b)** Using synchronous I/O. It is much simpler, but much less efficient, than its asynchronous counterpart.

It introduced a different kind of glue, however—all the code associated with spawning and coordinating the actions of the threads. Also, the programmer's burden in terms of buffer management was no different than with ViC*.

We implemented an out-of-core version of Leighton's columnsort algorithm [11] using this approach [1, 2]. In the threaded C code, we represented each stage of a pipeline as a thread. Since threads run concurrently, it was up to the programmer to ensure that they operated on buffers in order. The programmer spawned threads and coordinated among them using semaphores. Each stage had to wait for a signal from its predecessor before operating on a particular buffer; each stage also had to signal its successor after it finished working on the buffer. The structure of the pipeline, therefore, was tied to the operations within threads. The programmer had to write code to ensure that the threads signaled each other appropriately.

As with ViC*, the programmer was entirely responsible for buffer management in threaded code. The programmer had to allocate, deallocate, and store buffers, and recycle them from the global pool when necessary. Moreover, the programmer had to allocate and store additional buffers for stages whose work could not be done in place.

3. The FG environment

In this section, we present a simplified description of FG. The central job of FG is to provide the glue for HEC applications that fit a pipeline structure. To create asynchrony, FG represents each step of work as a pipeline stage and maps it to a thread. FG also manages the buffers to amortize the cost of transferring data among levels of the memory hierarchy. By shouldering both of these tasks, FG hides latency in such programs.

FG uses pthreads to overlap work in the pipeline. The programmer does not write any code associated with pthreads but instead has the simpler task of creating FG-defined objects. FG spawns all the threads, coordinates the semaphores for communication among threads, and kills the threads after the pipeline has completed. The programmer maps one or more functions to each thread; these functions, however, are completely synchronous. The programmer need not take overlap into consideration when writing code with FG. In fact, a programmer with only a rudimentary knowledge of threads can easily produce threaded code in FG.

FG also assumes all aspects of buffer management. It allocates buffers at the start of execution and deallocates them at the end. The programmer need only specify the number and size of buffers. FG also recycles buffers appropriately, so that the programmer need not write code to establish or maintain a global pool of buffers. Moreover, FG introduces a new kind of buffer that we call an *auxiliary buffer*. An auxiliary buffer does not traverse the pipeline, but is simply a block of memory that can be requested by any stage. We have seen that it is sometimes necessary to use a second buffer in a stage, such as one that performs a permutation, and FG supplies auxiliary buffers for this purpose. Finally, FG ensures that buffers traverse the pipeline in sequential order. A stage does not have knowledge of its successor and predecessor stages. Instead, each stage simply calls FG-supplied functions to accept buffers from its predecessor and convey buffers to its successor.

After writing simple, synchronous C or C++ stages, there is little a programmer must do to put together an FG program. FG provides a class to describe a thread, so that the programmer need not interact directly with the pthreads interface, as well as a class to describe a stage. Both of these classes are easy to use. The programmer creates the FG-defined stages and threads necessary for the pipeline. Then the programmer simply assigns the appropriate functions to the stages and maps each stage to a thread. All that is left is to create the pipeline, another FG-provided class. We will show in Section 5 that setting up and running the FG pipeline is quite a bit simpler for the programmer than setting up and running a pipeline with ViC* or with threads.

The preceding description does not tell the whole story of FG. Its capabilities extend well beyond the linear pipeline structures that our three benchmark applications fit. Additional features of FG include multistage threads, multistage repeat, buffer swapping, macros, hard barriers, soft barriers, and implicit threads. Additionally, programmers can incorporate fork-join constructs and directed acyclic graphs into an FG pipeline. FG also uses time-balance strategies to reduce the execution time of a pipeline on the fly. Although we touch on some of these features in Section 5, the details are beyond the scope of this paper.

4. Benchmark applications

In this section, we present our benchmark applications: out-of-core implementations of BMMC permutations, FFT, and columnsort. We implemented the first two programs in ViC* and the third in C code with threads; we implemented all three in FG for comparison. In Section 5, we will show the reductions in source code size and complexity that FG affords for the three programs.

4.1. BMMC permutations

A BMMC permutation is specified by an $n \times n$ characteristic matrix A whose entries are drawn from $\{0, 1\}$ and that is nonsingular over $GF(2)$. That is, multiplication is replaced by logical-and, and addition is replaced by exclusive-or. The following are examples of BMMC permutations: matrix transpose when all dimensions are powers of 2, shuffle and unshuffle permutations, Gray-code permutations, and bit-reversal permutations.

For our BMMC-permutation pipeline, the stages are as follows: a read stage, a first permute stage, a communicate stage, a second permute stage, and a write stage. The read and write stages perform disk I/O. The communicate stage performs interprocessor communication across the cluster. The two permute stages work only within each node's CPU and local memory. We can overlap the five stages, therefore, because the CPU is idle during the read, write, and communicate stages, and it is busy during the two permute stages.

Let us explore the path of a buffer through this pipeline. First, the read stage reads a portion of the data into the buffer from disk. Each item i to be permuted initially belongs to some processor $P(i)$ and has a destination processor $P'(i)$. The first permute stage rearranges the data on each processor so that the items mapped to each target processor are contiguous in local memory. The communicate stage performs interprocessor communication so that each item i moves from $P(i)$ to $P'(i)$. The second permute stage rearranges the data locally on each processor. Finally, the write stage writes the data from buffer to disk.

4.2. FFT

The FFT is a computationally efficient algorithm for computing the discrete Fourier transform of an N-element vector. First, the input undergoes a bit-reversal permutation. Then a butterfly graph of $\lg N$ stages is computed. (We use $\lg N$ to mean $\log_2 N$.) In the sth stage of the butterfly graph, elements whose indices are 2^s apart participate in a butterfly operation [6, Chapter 30].

Figure 3 illustrates our out-of-core FFT implementation. We start with a bit-reversal permutation, for which we use our out-of-core BMMC permutation pipeline as a subroutine. Then there are $\lg N / \lg F$ superlevels, where F is the buffer size. Each superlevel consists of N/F separate "mini-butterflies" (on F elements and with depth $\lg F$) followed by a particular type of BMMC permutation on the entire vector. Each pass of the FFT implementation, therefore, consists of a pipeline with a read stage, a mini-butterfly stage, and a write stage, followed by a subroutine that performs a BMMC permutation. See [8] for details.

4.3. Columnsort

We implemented an out-of-core version of Leighton's columnsort algorithm in C code with threads. Columnsort sorts N items, which are treated as an $r \times s$ mesh. When columnsort completes, the mesh is sorted in column-major order. Columnsort proceeds in eight steps. Steps 1, 3, 5, and 7 are identical: they sort the columns of the mesh. Each of the even-numbered steps performs some fixed permutation on the mesh, but the fixed permutation differs from stage to stage.

Our columnsort implementation makes four separate passes over the data. Each pass performs two of the eight steps of the columnsort algorithm. Each pass also includes a read stage, a write stage, and a communicate stage. Figure 1 illustrates a pass of columnsort. It represents the gen-

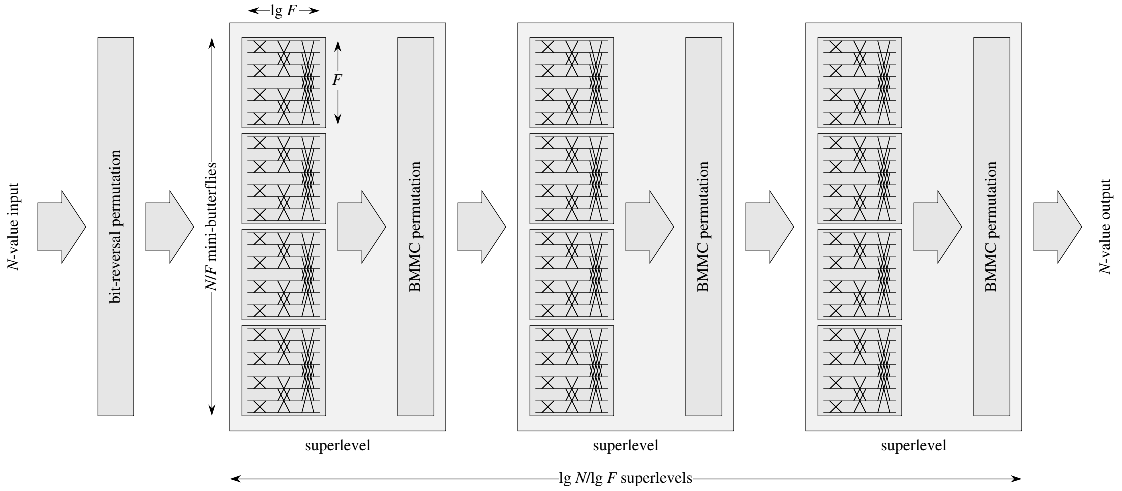

Figure 3: The structure of the out-of-core FFT algorithm. After a bit-reversal permutation, we perform $\lg N / \lg F$ superlevels. Each superlevel consists of N/F mini-butterflies on F values, followed by a BMMC permutation on the entire vector.

	Seconds: 8 GB/proc				Code size			
Program	non-FG	FG	difference	improvement	non-FG	FG	difference	improvement
BMMC	1844	570	1274	69.1%	3204	2736	468	14.6%
FFT	4245	1638	2607	61.4%	7612	6290	1322	17.4%
Columnsort	1893	1862	31	1.6%	7824	5820	2004	25.6%

Table 1: Running times and code size reductions for our three benchmark programs with and without FG. We show the running times, in seconds, and the lines of source code. We also show quantitative differences and percentage improvements. Each time shown is the average of three runs.

eral structure of a pass, although the details of the stages vary with each pass. In our columnsort implementation, the buffer size is equal to the size of one column of the input mesh; every time we read from or write to disk, we transfer exactly one column. Let us explore the path of a buffer through the pipeline. First, the read stage reads one column of the input mesh from disk into the buffer. The sort stage sorts each column. As with BMMC permutations, each item i initially belongs to some processor $P(i)$ and has a destination processor $P'(i)$. Therefore, the communicate stage transmits items among processors so that each item i moves from $P(i)$ to $P'(i)$. The permutation stage permutes the data locally on each processor. Finally, the write stage writes the data from the buffer to disk.

5. Reducing code size and complexity

In this section, we present the reductions in source code size and complexity that FG affords. The authors have shown previously [4] that using FG speeds the execution time of the BMMC permutations, FFT, and columnsort implementations. Table 1 summarizes the running times for the three programs using 8 GB of data per processor on a 16-node cluster. Due to disk-space limitations, 8 GB per processor was the largest problem size that we could test. The authors presented more detailed running-time results in [4], but the focus of the present paper is on code size and not experimental results, and so it suffices to show a representative problem size here.

Table 1 also summarizes the differences in source code size between FG and non-FG programs. For BMMC permutations, using FG reduces source code size by 468 lines, or approximately 14%. For FFT, FG reduces source code size by 1322 lines, or approximately 17%. Finally, for columnsort, FG reduces source code size by 2004 lines, or approximately 25%.

Where do these reductions come from? FG lessens the size and complexity of source code in ViC* programs by eliminating the need for writing asynchronous I/O code. In threaded programing, FG eliminates the need for writing any code associated with the threads. Furthermore, FG takes on all buffer management, a common component of both ViC* and threaded programming.

For our benchmark programs, we separate the code into two parts: glue and base code. Figure 4 illustrates the breakdown between glue and base code in FG and non-FG programs. In FG, the glue does not disappear completely. Instead, the glue is code that we use for setting up, running, and dismantling pipelines, as well as for accepting and conveying buffers. As the figure shows, however, the glue in

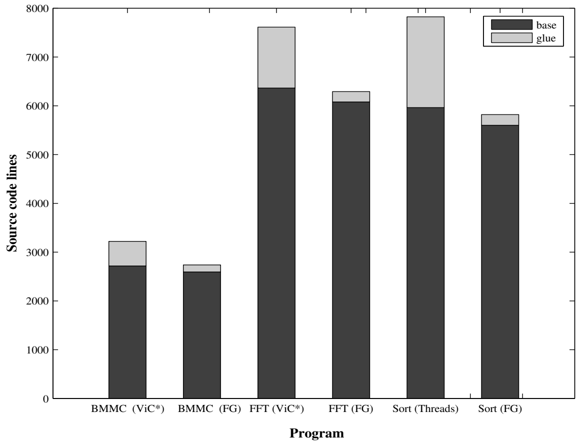

Figure 4: Lines of source code dedicated to glue and to base code for BMMC permutations, FFT, and columnsort, with and without FG. Without FG, the BMMC permutation program uses 502 lines of code for glue, FFT uses 1249 lines, and columnsort uses 1861 lines. With FG, these programs require only 143, 212, and 220 lines of code for glue, respectively.

the FG programs accounts for substantially fewer lines of code than in the non-FG programs.

With the two ViC* programs, the glue is devoted mostly to asynchronous I/O and buffer management. As we show in Figure 4, ViC* requires 502 lines of glue for BMMC permutations and 1249 lines for FFT. These lines amount to 15.59% of the total code for BMMC permutations and 16.41% for FFT. The corresponding FG programs, on the other hand, need only 143 lines of glue for BMMC and 212 lines for FFT, respectively, 5.23% and 3.37% of the total code.

With the threaded program, most of the code reduction comes from setting up and coordinating the threads as well as from managing buffers. In the threaded implementation of columnsort, there is a considerable amount of code devoted to spawning threads and coordinating the concurrent actions among them. Figure 4 shows that, in columnsort implemented with threading, the glue accounts for 1861 lines of code, which is 23.77% of the total. With FG, on the other hand, the glue is reduced to setting up, running, and shutting down the pipeline, which requires only 220 lines of code, or 3.78% of the total.

Reducing source code size is not the only benefit of FG; it lessens the complexity of the code as well. We cannot quantify this claim, but in our experience we have found that the glue FG provides is particularly difficult to write and debug. Without FG, the programmer must not only implement the algorithm itself, but also write the code to make the implementation run efficiently in an HEC environment. With FG, the programmer writes little glue, and the functions are straightforward and synchronous. In our experience, the great majority of base code is far simpler to write than the code to overlap operations. Writing the code for buffer management is onerous as well. Furthermore, we have found that it is especially difficult to debug the glue. Particularly in the threaded programs, for which standard debugging tools are not reliable, finding errors in the glue often proves to be a substantial burden.

FG also allows for easy structural experimentation. When writing HEC programs, a programmer often searches for small changes to improve performance. Reducing running time by even a small percentage can be important, and altering the structure of a pipeline can reduce running time. For example, a programmer might map more than one stage to a single thread—mapping a read stage and a write stage to a single I/O thread since the two operations serialize at the disk anyway. Without FG, replacing a one-to-one mapping of stages to threads by a many-to-many mapping entails considerable time and effort. Figure 5 shows that, with FG, it requires only a few lines of code to make the change. Moreover, it is just as easy to revert to the former mapping if the change does not prove effective. FG can simplify the use of threads even further—it is possible to write a program in FG without explicitly creating threads at all. A programmer can simply create the stages of a pipeline, and FG creates a one-to-one mapping from the stages to threads.

FG also has functionality to find performance improvements. Since the best performance generally comes from a time-balanced pipeline, FG monitors the progression of buffers from stage to stage to determine whether any one stage processes buffers more slowly or more quickly than others. FG searches for any stage that becomes a *bottleneck stage*—it has more buffers in its queue than other stages—and replicates it in another thread. It also searches for any stage that becomes a *spewing stage*—it processes buffers more quickly than other stages—and lowers the priority of

```
FG_pipeline_thread_helper io_thread = new FG_pipeline_thread_helper_info();
FG_pipeline_stage_helper
  read_stage = new FG_pipeline_stage_helper_info(io_thread),
  write_stage = new FG_pipeline_stage_helper_info(io_thread);
```

Figure 5: Mapping a read stage and write stage to one single I/O thread in FG. To split the stages between two threads, we would simply need to create one new thread and make a one-to-one mapping—a change that would require about one additional line of code.

```
stage1->replicate();
stage2->lower_priority();
```

Figure 6: Using stage replication and thread priority adjustments in FG. These two lines of code can yield performance improvements of up to approximately 4%.

its thread. We have found that the speed gains from these run-time techniques are modest—up to 4% at best. For such a small gain, it may not be worth the time and effort to implement them by hand. As Figure 6 shows, however, with FG, it requires little effort on the part of the programmer.

6. Conclusion

We conclude with a discussion of related work and our future plans for FG.

6.1. Related work

StreamIt [13] is a high-level language for stream programs that provides an abstraction for manipulating streams of word-size entities. One of the goals of the project is to simplify the programming of these streaming applications. To enable simpler code, it represents an algorithm as a hierarchical network of filters. It has a graphical editor to represent the hierarchy of components in a user-friendly way. With the StreamIt graphical editor, the programmer initially sees the top level components, and upon clicking on a component, it expands into its subcomponents. Another click causes the hierarchy to contract. FG and StreamIt share some common structures, and both projects attempt to simplify source code, but StreamIt is strictly for streaming applications.

StreamBit [12] is an optimizing compiler for StreamIt that targets bit streaming applications such as cryptography. It enables the programmer to produce a piece of code simply by sketching it. A sketch is a partial specification of the full implementation, and StreamBit derives the missing details. StreamBit uses this sketching capability to improve productivity for transforming a functional specification (written by a domain expert) to an optimization specification (written by a system expert). The domain expert writes an algorithm in a high-level domain-specific language, and the system expert optimizes it for the specific system. Although FG also simplifies programming, it does not use a sketch of code as StreamBit does. Rather, the programmer writes C or C++ code, and FG hides the complexity inherent in making the code run efficiently.

6.2. Future work

We have shown in the past that FG speeds execution time for the three benchmark programs presented here. In our future work, we plan to conduct usability studies to investigate FG's third goal: reducing time-to-solution. We plan to hold a programming case study with Dartmouth undergraduates who are familiar with threads. Each of the subjects will receive the same threaded programming assignment. Half of them will code with FG, and the other half will code with threads explicitly. We will use this setup to measure the time-to-solution for FG and non-FG programs.

References

[1] 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.

[2] 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.

[3] Alex Colvin and Thomas H. Cormen. ViC*: A compiler for virtual-memory C*. In *Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '98)*, pages 23–33, March 1998.

[4] 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.

[5] 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.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[7] 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.

[8] 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. Also Dartmouth College Computer Science Technical Report PCS-TR97-303.

[9] 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/.

[10] IEEE. Standard 1003.1-2001, Portable operating system interface, 2001.

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

[12] Armando Solar-Lezama and Rastislav Bodik. Templating transformations for bitstream programs. In *First Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, pages 27–37, February 2004.

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