Asynchronous Buffered Computation Design and Engineering Framework Generator (ABCDEFG)

Thomas H. Cormen* Elena Riccio Davidson† Siddhartha Chatterjee‡

1 Introduction

A key challenge to achieving high-performance computing in a High End Computing (HEC) system is to hide the latency inherent in accessing data that reside far from the processor. In current and probable future HEC systems, storage components have significantly higher capacities but also much higher access latencies as they get further from processors. Likely HEC applications will be data-intensive, working with datasets large enough that only the outer reaches of the memory hierarchy will have the capacity to hold them. Technology is progressing in such a way that accessing even data in main memory will have unacceptably high latency relative to processor and cache speeds. Programming environments that "scale the memory wall" will be necessary to achieve high performance in the face of continually increasing memory latencies and block sizes. Our thesis in this white paper is that *we can adapt the extensive corpus of out-of-core techniques to build such a programming environment, and demonstrate reduction in "time to solution" for data-intensive HEC applications while delivering high performance*.

Another fact of life when using memory hierarchies is that accesses are "blocked," in two meanings of the word. First, to amortize the high latency of a single access, data blocks consisting of many bytes are transferred between levels of the memory hierarchy in a single access. For example, cache misses cause entire cache lines to move, and page faults cause entire pages to move. Second, a single thread of execution may be prevented from making progress—i.e., it may be blocked—until the access completes.

Over the years, several researchers have observed that several programs that work on large amounts of data, as an HEC application would do, operate in a stylized way. Specifically, a program manipulates the data in one or more passes over it, where each pass reads each piece of information from somewhere in the memory hierarchy, operates on it, and writes it back to the same level of the memory hierarchy from which it came. Out-of-core applications (e.g., [**?**, **?**] and see [**?**] for a survey) provide a good example of this computing style. The individual passes in a given program may operate on the data differently, but they often have a common structure: a pipeline. Figure **??** shows a pipeline for one pass of out-of-core columnsort from [**?**]; the stages in this pipeline read from disk, locally sort, communicate, locally permute, and write to disk.

*Associate Professor, Dartmouth College Department of Computer Science, 6211 Sudikoff Laboratory, Hanover, NH 03755, thc@cs.dartmouth.edu, 603-646-2417

†Graduate Student, Dartmouth College Department of Computer Science, 6211 Sudikoff Laboratory, Hanover, NH 03755, laney@cs.dartmouth.edu, 603-646-0406

‡Manager, High Performance Software Environments, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, sc@us.ibm.com, 914-945-1682

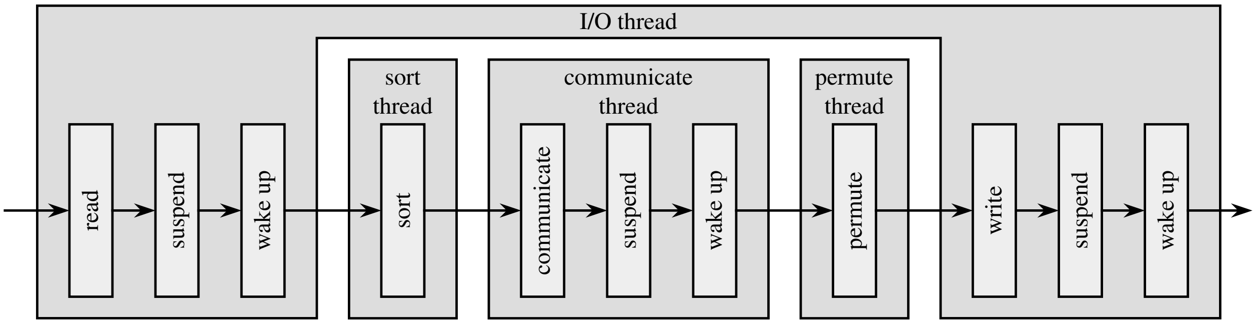

Figure 1: The pipeline for a pass of out-of-core columnsort. An I/O thread reads from a disk into a buffer; the thread suspends during the read, and when it wakes up, it passes the buffer to the sort thread. The sort thread locally sorts the buffer and passes it to the communicate thread. The communicate thread suspends during interprocessor communication, and when it wakes up, it sends the buffer to the permute thread. The permute thread locally permutes the buffer and passes it to the I/O thread. The I/O thread writes the buffer to disk, suspending during the write, and when it wakes up, the buffer has finished it way through the pipeline. Each thread works asynchronously with respect to the other threads.

It can be beneficial for the pipeline to be run asynchronously so that if a stage blocks due to a high-latency access (as would occur in read, write, and communicate stages), other stages can make progress. In order for the pipeline to run asynchronously, the program should be implemented so that each stage is clearly demarcated and the program should be run so that stages can run asynchronously. For example, these conditions would be satisfied if each stage were a thread. Moreover, each stage must be designed so that it transmits data to its successor stage.

Based on the above observations, we propose a programming environment that facilitates asynchronous stages within a pipeline, where the stages work on and transmit buffers of data. Each buffer corresponds to a block in the outer reaches of the memory hierarchy. Viewing a pipeline as a computational framework, the programming environment would generate the pipeline structure, based on the design of its stages. Ideally, the programming environment would also help the programmer engineer the program for performance through a combination of feedback and adaptation. Hence the name "Asynchronous Buffered Computation Design and Engineering Framework Generator," with the acronym "ABCDEFG." We call this programming environment "FG" for short.

The following sections discuss specific HECRTF charges.

2 Charge (2d): Discussion of resources, tools, and techniques needed to minimize "time to solution" by users of HEC systems

FG will greatly reduce the time to solution for users of HEC systems. In this section, we present the major components of FG, discuss how FG will indeed reduce the time to solution, examine some additional FG features, and finally outline some future enhancements.

FG components. The fundamental construct in FG is a pipeline of asynchronous stages. In a nutshell, the primary purpose of FG is to allow the programmer to specify and run a pipeline that operates on buffers. Each stage of the pipeline is written as a C or C++ function by the programmer within a thread that is written in part by the programmer and in part provided by FG.

Each stage takes a buffer of a given size as its input, works on that buffer, and then transmits the buffer to the next stage. The buffer size is fixed at the time that the pipeline is established; it 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 queue sits between each pair of successive stages. A stage conveys a buffer by placing it into the queue, and its successor stage accepts the buffer by removing it from the queue.

An FG-supplied source stage sends each buffer on its way through the pipeline, and an FG-supplied sink stage absorbs buffers that have progressed all the way through. In reality, the sink recycles buffers back to the source. Along with emitting buffers into the pipeline, the source stage numbers each buffer as it enters the pipeline, and this value travels along with the buffer as a tag that can be used by any stage. The sink stage has a special responsibility as well: it shuts down the entire pipeline upon the last buffer passing through it.

FG achieves asynchrony and hides latency by means of mapping stages to threads. For example, suppose that a stage reads data from a disk, which is a very high-latency operation. The stage would be designed to perform a simple, synchronous, disk-read operation. While the stage is waiting for the read to complete, its thread can yield to any thread whose stage is ready to run on the CPU. Stages that write to disk (or other high-latency locations in the memory hierarchy) or perform network-based communication would also be designed to perform synchronous operations within threads, which can yield.

Time to solution. Over the years, we have developed several out-of-core programs for problem domains such as permuting, sorting, and Fast Fourier Transforms. In our experience, a significant amount of code is "glue" holding together the portions of the program that perform asynchronously. This glue code has two major functions: (1) it statically overlaps I/O, computation, and communication, and (2) it manages buffers. Our programs have overlapped these operations both by using asynchronous I/O and communication calls in a single-threaded program and by making these calls synchronous but within a multithreaded program. The percentage of code that is this glue varies, but we estimate it to be in the range of 10%–35%.

Code size is one measure, but time to produce correct code is another matter altogether. The glue is often the most difficult code to write, and it is the most onerous and frustrating to debug. We found that, even making calls to the standard pthreads package, the glue accounted for approximately half of the total coding and debugging time, and sometimes even more.

FG eliminates almost all of the programmer's effort in supplying the glue. The programmer writes stages as synchronous operations, maps the stages to threads, specifies the number and size of buffers, and orders the stages in the pipeline. FG does the rest. How much coding and debugging time for the glue will FG save? Because FG is still under development, we do not yet know. But we expect that almost all of the coding and debugging of the glue will vanish, cutting development time by about half.

Additional FG features. FG is more complex than the above description might lead one to believe. Here, we explore some additional aspects of FG.

FG allows multiple stages to reside in a given thread. There are a few reasons why the programmer might wish to design multi-stage threads. First, if there were fewer buffers than threads, there would always be some thread making no progress because it is waiting for a buffer to arrive. By allowing more than one stage to co-reside in a thread, there are fewer threads, and so fewer buffers are needed to keep all threads busy. Second, some stages may require exclusive use of a common resource, and so they will serialize regardless of the thread structure. For example, in the pipeline of Figure **??**, one stage reads from a disk and another stage writes to the same disk. Because these two stages will serialize at the disk, they share a common I/O thread. Additional FG features, not described here, provide for good performance when these stages are mapped to a common thread. Third, with fewer threads, there is less overhead incurred from thread swapping. Fourth, placing multiple stages in the same thread allows them to share state.

The ability to easily place multiple stages in the same thread points out another significant benefit of FG in terms of time to solution. With FG, the programmer can experiment freely with different pipeline structures. Changing the mapping of stages to threads becomes easy, as does changing the number of buffers in the pipeline. Without FG, coding and debugging such changes is difficult. With FG, the main task left to the programmer is measuring the impact of such changes on performance.

FG allows buffers to be swapped within a stage. The reason is that a stage may take one buffer as its input but send a different buffer as its output to the next stage. For example, consider a stage that permutes its input buffer, where the permutation is not performed in-place. This stage would acquire an auxiliary buffer, permute its input buffer into this auxiliary buffer, and send the auxiliary buffer on to the next stage. Alternatively, we could have copied the auxiliary buffer back into the input buffer and sent the input buffer to the next stage, but we would rather not incur the expense of the extra copy.

FG manages all buffers that go through the pipeline. Therefore, if a stage needs to acquire an auxiliary buffer, FG supplies it. When the programmer specifies the number of buffers to create, this specification also includes the number of auxiliary buffers. The number of regular buffers never changes, nor does the number of auxiliary buffers. When a stage wants to treat an auxiliary buffer like a regular buffer, so that it can be conveyed to the next stage, FG requires that the auxiliary buffer and the regular buffer swap roles (via an FG-supplied function). This mechanism keeps the counts of the two types of buffers unchanged, and the memory that had held the regular buffer becomes an auxiliary buffer available for use by any stage that needs to acquire one.

Future enhancements. The above describes FG as it currently exists. Down the road, we envision even more capability.

Pipelines should be able to plug into other pipelines, as if they were (nonrecursive) subroutines. An equivalent view is that a pipeline stage is a node in a rooted tree, and any node can be the root of a subtree whose frontier is a linear sequence of stages. If there were only one stage per thread, implementing this notion would be straightforward. Multi-stage threads make such an implementation more of a challenge, however. We have designed a scheme for pipeline subroutines in the presence of multi-stage threads, but the details are beyond the scope of this white paper.

Performance tuning is currently the responsibility of the programmer. Future versions of FG will include the capability to instrument and give performance information about the individual stages, so that the programmer can alter the pipeline structure and number of buffers to improve performance. Even further in the future, FG will attempt to make such changes on its own, in a dynamic fashion, based on performance information that it gathers at run time. By relieving the programmer of some of the burden of performance tuning, the time to solution will be further reduced.

Yet another enhancement is to allow computational structures beyond simple, linear pipelines, e.g., directed acyclic graphs. We also envision integrating concepts from the StreamIt project at MIT [**?**] into FG. StreamIt allows pipelines with fork/join and looping constructs.

3 Charge (2c): Discussion of the types of system design specifications needed to effectively meet various application domain requirements

From the above section, we see that in order for FG to effectively measure performance of the pipeline components, there must be a way to determine how much time each stage spends running, ready, and waiting. Thus, there needs to be an accurate measurement mechanism for thread performance, as well as a way to report that information back to the programmer. Moreover, since we envision FG eventually making dynamic changes to the pipeline and buffering, there will need to be a way for FG to query this information.

To the best of our knowledge, current commodity hardware and operating-system software do not give accurate performance information about thread running, ready, and waiting times. Therefore, in order for FG to achieve its full potential, systems will have to provide such information.

4 Conclusion

The FG system facilitates the development of programs that make passes over data and that need to "scale the memory wall" for high performance. It allows the programmer to easily specify the structure of the pipeline for each pass and in so doing, it can reduce the time to solution by reducing coding and debugging time significantly.