Dartmouth College

Dartmouth Digital Commons

6-2003

Enhancing Asynchronous Parallel Computing

Elizabeth Anne Hamon
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation

Enhancing Asynchronous Parallel Computing

Elizabeth Anne Hamon

Department of Computer Science, Dartmouth College
Seniors Honors Thesis
Advisor: Thomas H. Cormen

Abstract

In applications using large amounts of data, hiding the latency inherent in accessing data far from the processor is often necessary in order to achieve high performance. Several researchers have observed that one way to address the challenge of latency is by using a common structure: in a series of passes, the program reads in the data, performs various operations on it, and writes out the data. Passes often consist of a pipeline structure composed of different stages. In order to achieve high performance, the stages are frequently overlapped, for example, by using asynchronous threads. Out-of-core parallel programs provide one such example of this pattern. The development and debugging time resulting from coordinating overlapping stages, however, can be substantial. Moreover, modifying the structure of the overlap in an attempt to achieve higher performance can require significant additional time on the part of the programmer. This thesis presents FG, a Framework Generator designed to coordinate the stages of a pipeline and allow the programmer to easily experiment with the pipeline's structure, thus significantly reducing time to solution. We also discuss preliminary results of using FG in an out-of-core sorting program.

Contents

Chapter 1

Introduction to FG

1.1 Background

Our goal is to create a framework generator for writing high-performance buffered asynchronous programs. The motivation behind this framework generator is latency. Latency poses a challenge to researchers developing numerous kinds of high-performance applications. While the computer industry has seen great improvements in the speed of both networks and processors in recent years, a corresponding advancement in the speed of accessing data further out in the memory hierarchy has yet to come. The time required for reads and writes often sets the lower limit on the total amount of time a program takes. In addition, accessing a large contiguous block of data is nearly as efficient as accessing a single word. Therefore, to access a given amount of data, it is more efficient to access it fewer times in larger chunks. As a result, an important goal on the part of many researchers is to minimize the total number of accesses to high-latency memory locations. One way of addressing this issue is to construct the program to perform a small number of passes over the data, with each pass consisting of a number of stages. For example, a pass might consist of the following stages: a read stage, a sort stage, a communicate stage, and a write stage.

Merely using a small number of passes is insufficient, as accessing the data from the memory hierarchy is still great. Thus, developers try to keep the total time close to the physical limit imposed by the data access, by doing other work during the time the program waits for the data access to complete. One way of accomplishing other work during this time is to run the stages asynchronously, doing, for example, CPU and/or communication work during the data access. In order to maximize efficiency, the stages pass to each other large buffers of data, containing the large blocks of data read from high-latency memory locations. These programs have a clear pipeline structure formed by the stages that pass the buffers; they often recycle the buffers, each using one buffer for a new set of data when it finishes processing a previous set of data.

Out-of-core programs provide one example where latency is an issue. These programs often follow the model of a small number of passes over the data with each pass consisting of a pipeline (see [CC02], [CCW01] and [BC99] for sample programs). Out-of-core programs often use a small number of passes over the data. Because of memory limitations, the first stage thus typically consists of a read of the data from disk and the last consists of a write of the data back to disk. Since the total size of all allocated buffers in an out-of-core program is smaller than the size of all the data to be sent through, out-of-core programs provides an example of recycling of buffers.

Asynchronous threads have greatly reduced the impact of latency on overall performance. The reader may thus wonder why the framework generator is needed; threads have been available for years, so why would there be a need for change? There is nothing wrong with threads, in the same way that there is nothing wrong with assembly language. Assembly language can do anything a higher-level language can do. Nonetheless, higher-level languages exist on top of assembly language, because they make the programmer's task easier by allowing the programmer to focus on the more essential and problem-specific parts of his program.

Similarly, threads and the synchronization mechanisms that accompany them do meet the needs of programmers. In programs addressing latency, however, these mechanisms are used again, resulting in code that is similar across many of these programs. Furthermore, the code is tricky to reason about and program, and thus it is prone to mistakes. Therefore, the development time and debugging time for the asynchronous parts of the code can be substantial. Modifying the thread structure can also require substantial revisions to the code, and it can introduce new errors. Similar to the purpose of higher level languages, the framework generator will be useful in eliminating the details of the asynchronous parts of the code, allowing the programmer to focus on what is essential to his code and give him ways of finding new thread structures to improve overall performance. The additional execution time resulting from the framework generator should be insignificant. In the following chapters, we discuss our framework generator, called the Asynchronous Buffered Computation Design and Engineering Framework Generator, or ABCDEFG. We refer to ABCDEFG as FG, pronounced "effigy."

A programmer writing asynchronous buffered programs must deal with many details. In terms of pipeline structure, these details include setting up and shutting down the pipeline, spawning and waiting for threads, and waiting for and posting to semaphores to control correct buffer access. Calculating which buffer each stage should access, recycling buffers, and determining when a stage should shut down are additional details. Lastly, the programmer must ensure that a stage knows its predecessor and successor in the pipeline and that a thread with multiple stages knows when to execute each stage.

FG takes care of all of these details. It eliminates the programmer's need to worry about which buffer to access and the need to know which stage follows which stage in order to post a signal, because FG conveys the correct buffer directly to each stage. FG takes care of setting up the pipeline, running the pipeline and recycling the buffers. It frees the programmer from the hassles of dealing with synchronization mechanisms. It ensures that all threads know which buffer is the last and exit at the correct time. It also gives the programmer a way to end the pipeline if one stage deems that early termination is necessary.

We would like to evaluate FG on whether it reduces the programmer's time to solution. Such a study would however require two nearly identical programmers, one developing a program with FG and one developing the same program without. As a result, this evaluation is beyond the scope of this thesis. Instead, we will measure achievement based on the amount of additional execution time and on whether we can find a higher performance thread structure without much additional work.

In the following section we describe an overview of the workings of FG, discussing in more detail the less obvious parts of FG. In Chapter 2, we give a detailed design specification of FG. Chapter 3 discusses the issue of deadlock in FG. Finally, Chapter 4 describes experimental results, comparing a program written both with and without FG.

1.2 Overview of FG components

1.2.1 Stages, threads, and buffers in the pipeline

At its most fundamental level, FG creates and runs a pipeline of stages that operate on a set of buffers. Each stage maps to a thread, and thus the pipeline stages are asynchronous. The programmer specifies the stages and the threads as C functions, along with the mapping from stages to threads. All buffers in FG are the same size, but the programmer specifies both the number and size of the buffers. Each stage receives a buffer from the previous stage, processes it, and conveys it to the next stage. A queue exists for each pair of successive stages. A stage thus accepts a buffer by removing the first buffer in the queue and receiving it; it conveys the buffer by placing it in the queue that will be read by the next stage. Each time a stage processes a buffer, the stage is on a new *round*.

FG also includes a source and a sink stage, each belonging to their own thread. The source stage marks each buffer with the number of its round and passes it into the pipeline. FG flags a buffer as the last buffer and updates itself so that the threads will now exit. We refer to this flagged buffer as the ***caboose*** and to the flag as the ***caboose flag***. Unless the programmer chooses to set the caboose flag, the source also is responsible for setting the caboose flag. FG's sink takes the buffers from the end of the pipeline and passes them back to the source; once the sink receives the caboose, it stops conveying the buffers and shuts down the pipeline.

1.2.2 Multi-stage threads

In FG, the programmer divides up the program into stages of a pipeline, with each stage belonging to a thread. Each thread, however, may consist of one or more stages. We consider a thread consisting of more than one stage to be a ***multi-stage thread***. Multi-stage threads exist in order to give the programmer additional flexibility and efficiency. While thread overhead is generally small, it may be non-negligible in programs with multiple threads sharing a common resource, such as a disk. In these programs, the shared resource can cause the threads to serialize, making the gain in performance from overlap small. It may somtimes be smaller than the loss of performance caused by thread overhead. A similar loss of performance could occur in programs with a large number of CPU-bound or communication-bound stages. Multistage-threads provide the user the opportunity to experiment without paying thread overhead.

A second reason to allow the option of multi-stage threads is to give the programmer additional flexibility. Before any experimentation, the programmer does not likely know the best thread structure; even if the optimal structure is easy to implement, the programmer may still need to put substantial time and effort into discovering this structure. With the option of multi-stage threads, the programmer can experiment with the stages more easily to find the best thread structure, and he may find surprising combinations to yield the most efficient program. The programmer might even find that the cost/benefit ratio for using more threads depends on the specifics of the parameters to the algorithm, rather than the algorithm itself. For instance, sorting smaller amounts of data may have a different optimal structure than sorting larger amounts, or the benefits of having separate read and write threads may depend on the speed of the disk or the amount of data accessed from the disk each time. The programmer could develop the program to determine the specific thread

structure at runtime, based on parameters passed to it. Without multi-stage threads, determining thread structures at runtime would be difficult, if not impossible.

1.2.3   Running the stages

In order for the pipeline to run, some part of the program must call the stages at the appropriate times. Instead of forcing the programmer to write each stage to call some function to start the next stage, FG takes care of calling all of the stages itself. FG does so by means of a function that not only calls the stages but decides which stage a thread calls at a given moment. We refer to this function as the ***decider function***. An FG pipeline is static, as the stages that the decider function calls are predetermined when the pipeline is established; no modification of which stages run can occur afterwards. The decider function has two additional purposes: to provide additional flexibility to multi-stage threads through multi-stage repeat, and to control the shutting down of the pipeline. We discuss multi-stage repeat here and shutting down the pipeline in Section 1.2.5.

Multi-stage repeat

Multi-stage repeat is the number of times the decider function calls a stage before it executes the next stage of a multi-stage thread. By further limiting the idle time of the threads, this option allows FG to be a pipeline of stages rather than of just threads. In essence, it allows the programmer to gain some of the benefits of using one thread for each stage without always paying the cost of thread overhead; with multi-stage repeat, stages sharing a thread can overlap in a more efficient way. In Figures 1.1 and 1.2, we consider an example in which we have two pairs of stages: CPU_1 and CPU_2, which use the CPU intensively and belong to a *CPU* thread, and $Comm_1$ and $Comm_2$, which communicate with other processors and belong to a *Comm* thread. The order of the stages is CPU_1, $Comm_1$, CPU_2, and $Comm_2$. Figure 1.1 shows the result if no multi-stage repeat is allowed. While $Comm_1$ is processing, $Comm_2$ cannot process, as it has no buffer to process. CPU_1, which may have buffers ready from its predecessor stage, cannot process them because the *CPU* thread is waiting in CPU_2. Thus, CPU_1 does not execute until $Comm_2$ also executes. At event 0, CPU_1 starts to process buffer 1, and at event 3, $Comm_2$ starts to process buffer 1; at time 4, we have finished processing buffer 1. As the figure shows, it takes $(3n + 1)$ events to process n buffers.

In Figure 1.2, we consider an identical setup, except that we have a multi-stage repeat of 2. Thus, CPU_1 must execute twice before CPU_2 does. We see from the figure that a multi-stage repeat of 2 allows CPU_1 to process a buffer at the same time as $Comm_1$ is processing a buffer. Thus, we find that the total number of events to process $2n$ buffers is $(4n + 1)$; this improvement is approximately 50%. Moreover, this example is not unrealistic. When we implemented an out-of-core sorting algorithm called M-columnsort, it consisted of a set of internal stages composed of 4 pairs of sort and communicate stages, with the sort stages belonging to one thread and the communicate stages belonging to another. Multi-stage repeat can provide a benefit over using all different threads, because there is no cost of swapping between the threads. We considered other variations on a single multi-stage repeat, but we chose this one because it ensured that deadlock would not arise (see Section 3.2 for other variations and their associated risk of deadlock).

The other task of the decider function is to ensure that the pipeline shuts down correctly. Recall that the caboose indicates the last round in the pipeline. FG provides the capability for a stage in

Event	Stage CPU_1	Stage $Comm_1$	Stage CPU_2	Stage $Comm_2$
0	Process buffer 1	Wait	Wait	Wait
1	Wait	Process buffer 1	Wait	Wait
2	Wait	Wait	Process buffer 1	Wait
3	Process buffer 2	Wait	Wait	Process buffer 1
4	Wait	Process buffer 2	Wait	Wait
5	Wait	Wait	Process buffer 2	Wait
6	Process buffer 3	Wait	Wait	Process buffer 2
7	Wait	Process buffer 3	Wait	Wait
8	Wait	Wait	Process buffer 3	Wait
9	Process buffer 4	Wait	Wait	Process buffer 3
10	Wait	Process buffer 4	Wait	Wait
11	Wait	Wait	Process buffer 4	Wait
12	Process buffer 5	Wait	Wait	Process buffer 4

Figure 1.1: The progression of buffers through an FG pipeline without multi-stage repeat. The buffers travel through the pipeline one stage at a time. One thread runs both stages CPU_1 and CPU_2, while another thread runs both stages $Comm_1$ and $Comm_2$. Despite having two threads, FG can process two buffers at most once every three events.

Event	Stage CPU_1	Stage $Comm_1$	Stage CPU_2	Stage $Comm_2$
0	Process buffer 1	Wait	Wait	Wait
1	Process buffer 2	Process buffer 1	Wait	Wait
2	Wait	Process buffer 2	Process buffer 1	Wait
3	Wait	Wait	Process buffer 2	Process buffer 1
4	Process buffer 3	Wait	Wait	Process buffer 2
5	Process buffer 4	Process buffer 3	Wait	Wait
6	Wait	Process buffer 4	Process buffer 3	Wait
7	Wait	Wait	Process buffer 4	Process buffer 3
8	Process buffer 5	Wait	Wait	Process buffer 4

Figure 1.2: The progression of buffers through an FG pipeline with a multi-stage repeat of 2. One thread runs both stages CPU_1 and CPU_2, while another thread runs both stages $Comm_1$ and $Comm_2$. Because of multi-stage repeat, FG can process two buffers every event. Multi-stage repeat allows for the overlap in events $4n + 1$ and $4n + 2$ that otherwise would not be possible.

the middle of the pipeline to set a buffer to be the caboose. No stage should run with a buffer that follows the caboose, but all stages starting with the one that set the caboose flag need to see the caboose. Thus the decider function also serves to check before running stages whether it would run the stage with a buffer before or after the caboose.

1.2.4   Pipeline buffers and auxiliary buffers

FG transfers data between stages, and stages access the data through buffers. FG creates these buffers at the time the pipeline is established according to parameters that the programmer passes it. FG consists of two different types of buffers, which we refer to as ***pipeline buffers*** and ***auxiliary buffers***. The difference does not lie in the buffers themselves but in the way the programmer uses them, and they are interchangeable. In this section we discuss the two types of buffers and how they are used.

Pipeline buffers

Pipeline buffers circulate through the pipeline, carrying data from stage to stage. Both the number and size of the pipeline buffers are fixed during the duration of the pipeline run. The size is constant across all buffers, allowing for an easy exchange of any two buffers. In our experience with parallel asynchronous programs, much of the work of each stage involves conveying and accepting buffers. Consequently, the maximum number of stages that may be running at one time is usually the number of pipeline buffers. To allow the programmer the opportunity to store and pass buffer-specific information from stage to stage, FG wraps each buffer in a ***thumbnail*** that contains not only a pointer to the buffer but also such information as the round number, the size of the buffer, and a pointer to anything the programmer wishes to associate with it. Although FG does not enforce it, every stage should accept and convey exactly one pipeline buffer each time it executes. As a stage conveys a buffer, FG assigns the buffer to the next stage.

Auxiliary buffers

Auxiliary buffers provide a very different service to the programmer. They serve to store information temporarily. For instance, many sorting routines are not in-place, and communication calls require different source and destination buffers. A request for an auxiliary buffer returns any one free auxiliary buffer, as the auxiliary buffers are not specific to a stage or thread. Thumbnails also enclose auxiliary buffers, but aside from the size of the buffer, no other field of the thumbnail is valid for an auxiliary buffer.

Recognizing that these buffers may often be large, FG allows the programmer to swap any two buffers, including a pipeline buffer and an auxiliary buffer. Note that such action only swaps the buffer addresses inside the thumbnails, and a pipeline thumbnail is always a pipeline thumbnail. Since this implementation of FG requires all pipeline buffers to be of the same size, the ability to swap two buffers requires that the auxiliary buffers must also be of the same size. If the programmer wishes to use other sizes of buffers, the stages must create their own buffers, and the programmer cannot exchange these buffers with pipeline buffers.

In order to obtain an auxiliary buffer, the programmer makes a simple call to FG_get_aux. Because the auxiliary buffers are in a pool and not specific to a stage or thread, requesting one does require the use of synchronization mechanisms. Thus, the programmer has two options for using auxiliary buffers. One option is to acquire an auxiliary buffer whenever needed, within a stage. This option is useful if the auxiliary buffers are needed for only short periods of time and the number of auxiliary buffers is small. Alternatively, the programmer can acquire an auxiliary buffer for the stage or thread in the thread's initial setup function or during the first time the stage runs. The programmer will prefer this option if he knows the stage will need the buffer every time, as he will pay the cost of the synchronization mechanisms only once.

1.2.5   Shutting down the pipeline

Shutting down the pipeline is a critical step in running any pipeline, whether the programmer uses FG or not; previous experience has also shown that it is sometimes tricky. All stages must know when to stop processing, lest they wait infinitely for a buffer that will never arrive or continue to manipulate the data after the necessary steps have completed. Assuming that the stages finish

correctly, threads must be sure to terminate only after all stages have finished. If the programmer allocated any resources that must be released, he must release them, but only after all stages sharing them have finished for this pass. Thus, FG provides the programmer with a caboose flag with every buffer to indicate whether this buffer is the last one through this stage. In this section, we discuss the function of the caboose and how the programmer uses it, as well as the way the pipeline shuts down after the caboose.

The caboose

FG uses the caboose to control the termination of the pipeline. A stage can also use the caboose to know after what round the pipeline will shut down. In our experience, a stage may need to do a different task the first and/or last time it runs. For instance, when we implemented an out-of-core columnsort, the last round involved reading in the leftover data, not the normal amount. While the same size of buffer is used, the amount of valid data in it is different. The stage can check if it is the first time, because all buffers include a tag indicating the round, and the buffer with tag 0 is the first buffer. The stage, however, does not know if it is the last buffer; in order to do so, it would need to know the exact number of rounds that the pipeline will have.

Although the stage may know the number of rounds in some cases, at other times it will not be able to find out this number. For instance, if a stage other than the FG-supplied source stage sets the caboose, the other stages have no way of knowing in what round this stage will set the caboose. Of course, the programmer could store the last round number in a global variable, but it is easier for the programmer if FG keeps track of it. Thus, we introduce the caboose flag into the thumbnail that goes with every buffer, allowing the stage to perform a simple check to see if this round is the last one and if it needs to do anything special. It is the programmer's responsibility to check the caboose and act accordingly if he needs to do anything special in the last round of the stage. If the steps he needs to do at the end of the pipeline are specific to the thread, such as freeing memory shared by all stages in the thread, then the programmer can ask FG to call the thread's cleanup function at the appropriate time by passing it to FG at the time the programmer establishes the pipeline.

FG itself uses the caboose to determine when the pipeline should terminate. Once a stage receives the caboose, FG never calls the stage again. The termination of the pipeline as a whole begins when the caboose reaches the sink. As a result, if the programmer chooses to set the caboose partway through the pipeline, all stages prior to the stage that set the caboose will never receive the caboose.

In the event that a stage other than the source sets the caboose and a stage prior to it is part of a multi-stage thread, the prior stage may have already begun to execute and is waiting for a buffer that will never come. In this case, once the caboose reaches the sink, the sink lets all stages know that the end is here, and all stages currently waiting will exit without finishing the stage. The reasoning behind exiting stages before they finish is that no stage should need to manipulate data once the pipeline has completed. Manipulation of data for rounds after the caboose round can only occur if a stage other than the source sets the caboose.

Once all stages in a thread finish, FG has the thread call its cleanup function, provided it is non-null, and then FG has the thread exit. After FG has ensured that all threads have exited, it frees up all resources, resets its parameters for the next call and exits. The memory freed consists

of all memory that FG allocated, including both pipeline buffers and auxiliary buffers. Thus if a programmer fails to release an auxiliary buffer at the end of the pipeline, it is nota memory leak. As a result, any data currently in the buffers is no longer valid after the pipeline run is completed.

Chapter 2

FG Design Specification

In this chapter, we present a functional design specification of FG, written in C. We discuss the variables, functions, arrays, and structs that make up the workings of FG. We omit only a few functions that serve as helper functions to provide clarity in the code. All variables and arrays in FG are static, preventing the programmer from accessing any of them outside of FG-provided functions. Many of the functions are also static, used only internally by FG. Of the remaining functions, the programmer is required to use some of them, such as FG_convey_buff, in order for FG to work properly. In addition to variables and functions, FG also consists of four kinds of structs: a pipeline stage helper, a pipeline thread helper, a stage's struct FG_params, and of course, the struct to which the FG_thumbnails refer. Of these four, we expect the programmer to set the fields of the first and second directly. In order for proper functioning of FG, however, the programmer should never modify the latter two except by FG-provided methods; modifications of these fields may cause FG to function incorrectly in a number of ways. Because the current implementation is in C, we cannot prevent the programmer from modifying such fields, and thus we rely on the programmer to self-enforce. In the following sections, we describe FG's variables, arrays, structs, and functions grouped according to their component area, as discussed in Section 1.2: setup, threads, stages, thumbnails, buffers, the caboose, and shutdown.

2.1 Pipeline setup

A correct setup of the pipeline is essential to FG. The setup consists of two parts: the programmer's part and FG's part. The programmer's job consists of specifying the desired stage and thread structures in arrays of pipeline thread helpers and pipeline stage helpers, discussed in Sections 2.2 and 2.3, respectively, and then calling the function to establish the pipeline with the appropriate parameters. Aside from writing the stages that the pipeline actually runs, the programmer's work finishes here. Upon return from the call to establish the pipeline, the pipeline has finished running. FG verifies, to the extent that C permits, that the parameters make sense. It then sets up its variables and arrays. Lastly it starts the pipeline by creating the pipeline's threads.

int FG_establish_pipeline(FG_pipeline_stage_helper * psh,
         int pipeline_size,
         FG_pipeline_thread_helper * pth,
         int num_threads,
         int num_buffers,
         int num_aux_buffers,
         int buffer_size,
         int rounds,
         int multi_stage_repeat,
         int param_size)

FG_establish_pipeline is the overarching function of FG. It sets up the pipeline and calls FG_run_pipeline and FG_cleanup_pipeline. When FG_establish_pipeline returns, FG has run the entire pipeline. We discuss how FG uses FG_establish_pipeline's parameters in the following sections: psh, 2.3; pipeline_size (which is the number of stages in the pipeline), 2.3; pth, 2.2; num_threads, 2.2; num_buffers, 2.5; num_aux_buffers, 2.5; buffer_size, 2.5; rounds, 2.6; multi_stage_repeat, 2.3, and param_size, 2.4. The return value indicates whether an error occurred. A return value of FG_SUCCESS indicates no error.

FG_in_establish_pipeline

The variable FG_in_establish_pipeline indicates whether FG is currently in FG_establish_pipeline. The value is set to 1 if FG is in FG_establish_pipeline and 0 otherwise. A call to FG_set_caboose_-setters will succeed only if FG_establish_pipeline is 0.

int FG_set_caboose_setters(FG_stage_rep * srs, int size, int source_sets_caboose)

FG_set_caboose_setters sets the stages in srs to be stages that can set the caboose without warning to the programmer. Source_sets_caboose indicates whether the source should set the caboose. If the source does not set the caboose, FG ignores the number of rounds given to FG_establish_pipeline. If size, indicating the size of the array of FG_stage_reps, is negative, ERR_NEG_VALUE is returned; else if srs is null, ERR_NULL_PARAM is returned. The programmer can only call this function before the pipeline is established; thus, if the pipeline has already been established, this function returns ERR_PIPELINE_STARTED. If the program is out of memory, it returns ERR_OUT_OF_MEM. Otherwise, it returns FG_SUCCESS.

2.2 Threads

FG assumes that the programmer wishes to run a pipeline, thus running the stages asynchronously. FG uses the standard pthreads package to implement the asynchronous stages. Each thread is responsible for a few tasks. If the programmer specifies an init function, the thread runs the function. Then it runs its stages. Lastly, if the programmer specifies a cleanup function, the thread runs this function. The init and cleanup functions are particularly useful if the stages within a thread wish to share resources.

Private variables about threads

FG_num_threads

FG_num_threads is the number of threads in the pipeline. FG_establish_pipeline sets FG_num_threads to be its formal parameter, num_threads, increased by 2, one for the source and one for the sink. The value of FG_num_threads remains constant throughout a pipeline run. Every thread array is of size FG_num_threads unless otherwise noted.

FG_thread_reps

In order to establish the pipeline, and in particular to map stages to threads, FG must have a way of uniquely identifying each thread. FG chooses to use a string, which we typedef as an FG_thread_rep. FG_thread_reps is an array of FG_thread_rep, with each entry storing the name of a unique thread. FG_establish_pipeline forms the array from the FG_pipeline_thread_helper array that FG_establish_pipeline receives.

FG_init_funcs

FG_init_funcs is the array of pointers to initial functions for threads. FG forms this array from the init_thread_function field in each struct in the FG_pipeline_thread_helper array that FG_establish_pipeline receives. If the thread's entry in FG_init_funcs is not null, FG_decider_func will run this function before it runs any stage.

FG_cleanup_funcs

FG_cleanup_funcs is the array of pointers to the thread's cleanup functions. FG forms this array from the cleanup_thread_function field in each struct in the FG_pipeline_thread_helper array that FG_establish_pipeline receives. If the thread's entry in FG_cleanup_funcs is not null, FG_decider_func will run this function after it has finished running all stages and before it exits.

FG_pipeline_thread_helper

An array of FG_pipeline_thread_helper structs determines the thread structure of the pipeline. An FG_pipeline_thread_helper is a struct that exists for each thread containing all of the relevant parameters for that thread. The struct consists of an FG_thread_rep for the thread name, a pointer to an initial thread function, a pointer to the parameters of an initial thread function, a pointer to a cleanup thread function, and an integer representing the thread priority. The programmer passes an FG_pipeline_thread_helper array to FG_establish_pipeline to set up the threads. The order of the threads in the array is the order in which FG will spawn the threads, but this order has nothing to do with the order of the stages in the pipeline.

FG_thread_rep the_thread Each FG_pipeline_thread_helper includes an FG_thread_rep containing the name of the thread. The programmer should always refer to the thread by using this name. Consequently, all FG_thread_reps in the array passed to FG_establish_pipeline must be different from each other as well as from FG's source and sink FG_thread_reps, which

are respectively "FG_source_thread" and "FG_sink_thread." If any duplicate name exists, FG_establish_pipeline will return with ERR_INVALID_PARAM during setup and the pipeline will not run.

void * init_params Init_params is a pointer to parameters that the thread's init_thread_function will use. For instance, if the init_thread_function opens files, the programmer may need to send in the names of the files that this function will open. This field is only used if the thread has an init_thread_function (i.e., the init_thread_function of the thread is non-null). If the programmer does not wish to use parameters in the thread's init_thread_function, he should set this field to null.

FG_ptr_function init_thread_function FG_pipeline_thread_helper is a pointer to an initial function for the thread. FG runs this function after it spawns the thread but before it runs for the first time any of the stages belonging to the thread. For instance, the programmer may wish to open files at the beginning of the run. In order to not use an initial function, the programmer must set the field to null. If the programmer chooses to use the function, he has the option to pass initial parameters to it, using the params field (a void *) of the FG_pipeline_thread_helper.

FG_ptr_function cleanup_thread_function Similarly, FG_pipeline_thread_helper is a pointer to a cleanup function for the thread. FG runs this function at some point after the caboose buffer reaches the sink. If all stages in the thread see the caboose, FG runs the function immediately after the caboose reaches the sink. If some stages in the thread do not see the caboose (because the caboose is set by a stage later on in the pipeline), then FG runs the function as soon as the caboose flag has been set by some stage and the thread calls FG_accept_buff (or if the thread is currently waiting in this function).

int thread_priority The field thread_priority is an int that gives the programmer's relative thread priority for this thread. In the current implementation of FG, however, this value is not used, because Linux's implementation of thread priorities does not currently work. Once this attribute is fixed in Linux, FG will change to make use of this field. The programmer will not need to specify a priority in the range of valid thread priorities, as FG will automatically adjust the thread priorities to fit this range.

Public thread functions

int FG_get_pipeline_stages(FG_thread_rep thrd, FG_stage_rep ** rep, int * num_stages)

FG_get_pipeline_stages is a function that allows a thread to find out which stages belong to it. At the time the programmer establishes the pipeline, he identifies each stage by an FG_stage_rep, which is simply typedef-ed to be a string. The programmer passes a FG_thread_rep corresponding to the thread's name, and FG stores an array of all the FG_stage_reps for that thread in *rep and stores the number of such stages in *num_stages. The programmer has no particular need to use this function, but it exists if the programmer wishes to find out this information in the middle of a stage, for example when he is debugging. The return value is FG_SUCCESS; ERR_NO_SUCH_THREAD, indicating that the programmer requested stages for a thread that does not exist; or ERR_NULL_PARAM, indicating that one of the two pointers in the parameter list is null.

int FG_get_mystage(FG_params * fg_params)

FG_get_mystage takes in an FG_params struct and returns the index of the stage according to fg_params. The programmer will find this function useful if he wants the stage or thread to know where in the pipeline the stage is located. The index of the source is 0.

Private thread functions

int _FG_run_pipeline()

_FG_run_pipeline runs the stages. FG_establish_pipeline calls this function when FG_establish_-pipeline has finished the setup. _FG_run_pipeline then proceeds to spawn each thread with the method FG_decider_func, which actually calls each stage function. It returns either FG_SUCCESS, indicating success, or ERR_PTHREAD_CREATE, indicating that FG failed to create one of the threads.

void FG_thread_killer(FG_params * p)

Each thread calls FG_thread_killer when the thread should exit. FG_thread_killer calls the cleanup function specific for that thread, if one exists. It then exits the thread.

2.3 Stages

Stages form the building blocks of the pipeline, specifying what work the FG pipeline actually does. Each stage belongs to at least one thread, specified in its FG_pipeline_stage_helper at the time the programmer establishes the pipeline. The stage runs repeatedly until it has either seen all buffers through the buffer marked as the caboose or until FG discovers that the programmer has set the caboose on a buffer further in the pipeline. Each stage accepts a thumbnail from the previous stage, does its work, and then conveys the thumbnail to the next stage. If a stage requires any parameters, it can access them through its parameter, FG_params * fg_params.

FG_params * fg_params

When a stage starts up it receives a pointer to an FG_params struct. This struct consists of two fields: a void * prg_params and int my_pipelinestage. The former points to any parameters the stage may need, set at the time the programmer establishes the pipeline. The second indicates where in the pipeline this stage is located, and it should never be modified. The stage will pass fg_params to the calls to FG_accept_buff and FG_convey_buff to ensure that each stage receives the correct thumbnails.

Private stage variables

FG_pipeline_size

FG_pipeline_size is the number of stages in the pipeline. FG_establish_pipeline sets FG_pipeline_-size to be its formal parameter, pipeline_size, increased by 2 (one for the source and one for the sink). This variable remains constant as the pipeline runs. Every stage variable consisting of an array is of size FG_pipeline_size, unless otherwise noted.

FG_stage_reps

FG_stage_reps is an array of FG_stage_rep, each storing the name of a stage. FG defines an FG_stage_rep as a char *. Each FG_stage_rep is unique. FG_establish_pipeline forms the array from the FG_pipeline_stage_helper array that FG_establish_pipeline receives.

FG_stage_n_func

FG_stage_n_func is an array containing the pointers to the functions that constitute each stage. Each function has a return type of void and has a parameter of type FG_params *. The functions taken in FG_establish_pipeline forms this array using the stage_function field of the FG_pipeline_stage_-helpers it receives. FG_stage_n_func[i] is the function that FG_decider_func will run for the stage whose name matches FG_stage_reps[i].

FG_stage_thread_map

FG_stage_thread_map is an array of integers that maps a stage index to a thread index. Thus FG_stage_rep[i] belongs to FG_thread_rep[FG_stage_thread_map[i]]. FG_establish_pipeline creates this array from the FG_pipeline_thread_helper and FG_pipeline_stage_helper arrays it receives.

FG_all_params

FG_all_params is an array of FG_params that constitutes the parameters for each stage. It consists of the stage index and a pointer to a struct of the programmer's choice. The programmer should use this array to maintain state in a stage between successive rounds. FG_establish_pipeline creates this array from its parameter, the FG_pipeline_stage_helper array. Each stage receives a pointer to its entry in this array as its sole parameter when the thread calls it.

FG_multi_stage_repeat

FG_multi_stage_repeat is an integer indicating how many times a multi-stage thread should run a stage before switching to the next stage. If all stages map to unique threads, its value is irrelevant. In order to prevent deadlock, all multi-stage threads must have the same repeat, as discussed in Section 3.2. The minimum value for multi-stage repeat is 1, and the maximum is the number of buffers. FG_establish_pipeline sets this value to that of its parameter multi_stage_repeat. However, if the programmer calls for a multi-stage repeat greater than the number of pipeline buffers, FG will print an error and reset it to the number of pipeline buffers.

FG pipeline stage helper

FG_pipeline_stage_helper is the stage version of FG_pipeline_thread_helper, detailing the structure of each stage. The array of FG_pipeline_stage_helpers determines the structure of the stages. FG_establish_pipeline takes the array of FG_pipeline_stage_helpers and uses it to set up the pipeline. The order of the stages in the pipeline must match the order of the stages in the array of FG_pipeline_stage_helpers passed to FG_establish_pipeline. Each FG_pipeline_stage_helper consists of an FG_-thread_rep, an FG_stage_rep, a pointer to the stage function, and a pointer to any initial parameters the stage may need.

FG_stage_rep the_stage Each FG_pipeline_stage_helper contains an FG_stage_rep the_stage, the name of the stage. Every FG_stage_rep in the array of FG_pipeline_stage_helpers must be different from all others as well as different from FG's source and sink FG_stage_reps, respectively "FG_source_stage" and "FG_sink_stage." If there are any duplicate stage names, FG_establish_pipeline will fail in the middle of setup and return.

FG_thread_rep the_thread Each FG_pipeline_stage_helper contains an FG_thread_rep the_thread, the name of the thread to which the stage belongs. Every stage's thread must be one entry in the array of FG_pipeline_thread_helper structs, and at least one stage must belong to each thread listed in the array of FG_pipeline_thread_helpers; if the arrays do not meet both of these conditions, then FG_establish_pipeline will fail in the middle of the setup and return.

FG_ptr_function stage_function This field indicates what function the stage actually executes every round. Every function should take a pointer to an FG_params struct as its sole parameter. It should return void, as FG will ignore this value and nothing else sees it.

void * params The variable params serves to give the stage any parameters, probably encapsulated in a struct, that the stage may need to execute the first time the stage runs. The stage's FG_params struct's field prg_params stores this value before the stage runs the first time, and unless the programmer changes that field, the stage will receive this same pointer in all subsequent rounds.

Private stage functions

void FG_decider_func(int * thread_index)

FG_decider_func is the function that runs the stages within a thread. When FG spawns each thread, each thread starts in this function. This function knows which thread called it based on the thread index given as the parameter. This function calls the FG_init_thread_function (discussed in Section 2.2) and then cycles through the stages, calling the stages in order and repeating them according to FG_multi_stage_repeat. Once a stage sets the caboose, however, FG_decider_func will skip over any stages prior to those through which the caboose has passed.

2.4 FG_thumbnail

An FG_thumbnail serves primarily to describe a buffer of data that the stages will manipulate. For two reasons, a stage receives an FG_thumbnail instead of a buffer. The first is that the stage

may wish to change which buffer it passes to the next stage, for example when the data ready for the next stage is in an auxiliary buffer. Since the stages pass FG_thumbnails, instead of buffers, between stages, a simple swap of the pointers in the FG_thumbnails is enough to ensure that the stage conveys the right data without unnecessary copying into the old buffer. The second reason, in our experience, is that a buffer may wish to carry with it certain pieces of information. In columnsort, for example, stages that sort in passes after the first do not need to do a complete sort, but only a merge sort, as the buffer already contains sorted runs in it. Therefore, the thumbnail could contain the information about the size of the sorted runs.

FG_thumbnail and queue private variables

FG_sems

FG_sems is an array of semaphores that control the passing of FG_thumbnails from stage to stage. Each stage has its own semaphore, whose value is the number of FG_thumbnails ready for this stage. The programmer has no access to the semaphores through any function.

FG_thumb_q_array

FG_thumb_q_array is an array of thumbnail queues, each of size FG_pipeline_size. FG_convey_buff uses this array to store the next thumbnail for the next stage, and FG_accept_buff uses it to access the next thumbnail for the current stage.

FG_thumbnail components

An FG_thumbnail serves to contain either a pipeline or an auxiliary buffer, as well as any information specific to the buffer. Each FG_thumbnail consists of a pointer to a struct containing an integer size, an FG_tag (indicating the current round), a pointer to the address of the buffer, a pointer to the parameters for the FG_thumbnail, and a caboose flag. In an auxiliary FG_thumbnail, neither the tag nor the caboose field is relevant.

FG_tag The FG_tag field in an FG_thumbnail indicates the round for the current stage. This field relies on the assumption that no programmer will ask for more than one non-auxiliary FG_thumbnail within one stage; doing so would violate FG's concept of a stage. As each FG_thumbnail passes through the source, the source sets its tag to indicate its round. The programmer can see the round by calling the function FG_get_tag but should never access this field directly.

size The size field in FG_thumbnail gives the size of the FG_thumbnail's buffer in bytes. The programmer can check its value through FG_get_size. FG deliberately does not provide an FG_set_size, and the programmer should not set this field manually. Doing so would not actually change the size of the buffer, as FG allocates all buffers of the same fixed size and does not check or modify the size of the buffers at any point during the running of the pipeline.

params The params field allows the programmer to send information about a buffer between stages. Typically, params will point to a struct of the programmer's design. The programmer

can either allocate and deallocate the memory to which each params points, or he can let FG allocate and deallocate it. If FG is responsible, then each FG_thumbnail's params field points to the same amount of memory, which is the amount specified by the parameter param_size passed to FG_establish_pipeline. The programmer should then only access the params field using the function FG_get_params. If the programmer wishes to allocate structs of different sizes or wishes to control this memory himself, he must allocate it and deallocate it himself. In this case, he should pass 0 as the param_size to FG_establish_pipeline. The programmer then accesses the field using both FG_get_params and FG_set_params. We highly recommend that the programmer think carefully before using static variables in stage functions and consider whether this field would be more approrpriate. Either two stages using the same function or two runs of a pipeline using the same functions may result in invalid states during various parts of the pipeline, and we have found that our out-of-core applications typically rely on several passes of which the saved states are relevant only to the particular pass.

caboose The caboose field indicates whether the current FG_thumbnail is the caboose, with 0 indicating that it is not the caboose and 1 indicating that it is the caboose. Once a stage sets the caboose field in a buffer, no stage, including the stage that set it, can unset it. In addition, FG permits at most one FG_thumbnail to contain the caboose at any moment. The programmer must access this field only through FG_get_caboose and FG_set_caboose. FG_set_caboose will return FG_SUCCESS if the call successfully sets the caboose.

Public thumbnail functions

int FG_get_size(FG_thumbnail th)

FG_get_size is a simple function that returns the size of the buffer in FG_thumbnail th.

void * FG_get_address(FG_thumbnail th)

FG_get_address is a function that returns the address of the buffer in FG_thumbnail th.

FG_tag FG_get_tag(FG_thumbnail th)

FG_get_tag returns the round of the FG_thumbnail th.

void * FG_get_params(FG_thumbnail th)

FG_get_params simply returns the address of the programmer's parameters for the FG_thumbnail th.

void FG_set_params(FG_thumbnail th, void * params)

FG_set_params sets the params field of FG_thumbnail th to be params.

int FG_get_caboose(FG_thumbnail th)

FG_get_caboose returns 0 if the caboose flag has not been set on any buffer and 1 if the caboose flag has been set for some buffer.

void FG_set_caboose(FG_thumbnail th, FG_stage_rep s)

FG_set_caboose tries to set the caboose for FG_thumbnail th. If the caboose flag was already set on another thumbnail, it returns ERR_CABOOSE. Otherwise, it returns FG_SUCCESS. If s is not a stage with permission to set the caboose, FG sets the caboose anyway, but it also prints a warning.

Private thumbnail functions

int _FG_fill_pool()

_FG_fill_pool creates and sets up the pool of FG_thumbnails, as well as the buffers that go within them. Every FG_thumbnail receives a size, a buffer, and a caboose value of 0. If the program is out of memory, it returns ERR_OUT_OF_MEM, and FG will immediately free all memory allocated so far, and it will return without running the pipeline. Otherwise, it returns FG_SUCCESS.

2.5 Buffers

FG assumes that the programmer wishes to convey buffers of data between successive stages in the pipeline. It recognizes that a programmer may also wish to store data in other locations for operations that are not in-place. Consequently, FG provides the programmer with both a set of buffers that can travel the pipeline and a set of buffers for auxiliary use. A programmer accesses all buffers through the addr field of its FG_thumbnail. Every time a stage runs, one buffer should enter the stage, and one buffer should exit the stage.

Private buffer variables

FG_buffer_size

FG_buffer_size is the size of each and every buffer in the auxiliary and pipeline buffer pools. FG_establish_pipeline sets this size, according to its formal parameter buffer_size. The size remains fixed for the duration of the pipeline run. The FG_get_size method returns this variable, but no method allows modification of the buffer size.

FG_num_buffers

FG_num_buffers is the number of buffers that circulate through the pipeline. The minimum number is 1, and the logical maximum is the number of rounds. FG_establish_pipeline sets this value according to its formal parameter num_buffers, and nothing else can modify it.

FG_num_aux_buffers

FG_num_aux_buffers is the number of auxiliary buffers. Stages can use the auxiliary buffers as needed. FG_num_aux_buffers can be 0 and has no maximum, as a stage may require several auxiliary buffers to do its work. FG_establish_pipeline sets this value according to its formal parameter num_aux_buffers, and nothing else can modify it.

Pipeline buffers

We have designed FG as a framework for use in programs that transfer buffers between stages. Buffers transferred between stages are referred to as ***pipeline buffers***, in contrast to ***auxiliary buffers***, which are used locally within a stage. All buffers, both auxiliary and pipeline buffers, have the same size, and the programmer can retrieve the size of the buffer by calling FG_get_size on the FG_thumbnail to which the buffer belongs. At the beginning of each stage, a stage must ask for its next buffer by calling the function FG_accept_buff. At the end of each stage, a stage must call the function FG_convey_buff to send the buffer to the next stage. If the desired result happens to be in an auxiliary buffer, then the programmer should call FG_swap_buffs. This call makes the auxiliary buffer now a pipeline buffer and the pipeline buffer an auxiliary buffer, allowing the next stage to have access to the correct information without copying it over.

Auxiliary buffers

In developing out-of-core asynchronous applications, we have frequently used temporary buffers; FG provides this functionality in the form of auxiliary buffers, which stages can obtain and release on an as-needed basis. The programmer receives an auxiliary buffer by calling FG_get_aux and returns it to the pool by calling FG_release_aux. FG does not permit the auxiliary buffers to have different sizes. The programmer can manage buffers of different sizes but cannot swap these buffers with the pipeline buffers. If a thread needs an auxiliary buffer and the programmer does not wish to get and release it between successive rounds of each stage, the programmer should give this thread an initial function to obtain it.

   FG provides no mechanism for ensuring that deadlock does not arise in trying to obtain an auxiliary buffer. We determined that the cost of checking for deadlock outweighs the benefits, especially since deadlock is avoidable. To guarantee that no deadlock will arise, the programmer must request at least as many auxiliary buffers as might be in use at one time and release them as appropriate.

Public buffer functions

void FG_convey_buff(FG_params * fg_params, FG_thumbnail th)

FG_convey_buff takes the FG_thumbnail and puts it in the queue that will be read by the next stage, where the current stage is determined by fg_params. It then signals the semaphore for the next stage, so the stage knows that there is now an FG_thumbnail ready for it and where to get it.

void FG_accept_buff(FG_params * fg_params)

FG_accept_buff determines the stage requesting the next FG_thumbnail from fg_params and calls a wait on the semaphore for that stage. When sem_wait returns, it checks whether it returned because the caboose reached the sink. If so, it calls FG_thread_killer to finish up with that thread. Otherwise, it updates accordingly and returns the appropriate FG_thumbnail.

void FG_swap_buffs(FG_thumbnail t1, FG_thumbnail t2)

FG_swap_buffs takes in two thumbnails and swaps the pointers to t1 and t2's buffers, thus exchanging the buffers between the two thumbnails.

Every stage in the pipeline would therefore use both FG_convey_buff and FG_accept_buff. A sample stage that uses no auxiliary buffers might look as follows:

```
void sample_stage(FG_params * fg_params)
{
  FG_Thumbnail thumb = FG_accept_buff(fg_params);

  /* do sample_stage work */

  FG_convey_buff(fg_params, thumb);
}
```

A more complex stage might use an auxiliary buffer. It might get an auxiliary buffer and have the data end up in this auxiliary buffer. In this case, a sample stage might look like:

```
void sample_stage_2(FG_params * fg_params)
{
  FG_Thumbnail thumb = FG_accept_buff(fg_params);
  FG_Thumbnail aux = FG_get_aux();

  /* do sample_stage_2 work, with data ending in aux */

  FG_swap_buffs(thumb, aux);
  FG_release_aux(aux);
  FG_convey_buff(fg_params, thumb);
}
```

FG_thumbnail FG_get_aux()

FG_get_aux returns a free thumbnail, with its buffer, from the pool. Until a buffer is free, this function blocks. A lock governs access to the pool of auxiliary buffers, and if an error occurs in using the lock, this function will return null.

int FG_release_aux(FG_thumbnail t)

FG_release_aux find the FG_thumbnail t in the auxiliary buffer pool and marks it as free. It signals that a thumbnail is now free, in case another thread is waiting for an auxiliary buffer in FG_get_aux. If a lock error occurs, it returns ERR_LOCK_UNLOCK; if a semaphore error occurs, it returns ERR_SEM_POST. Otherwise, it returns FG_SUCCESS.

2.6 Caboose

The caboose is FG's indication that the stages of the pipeline are seeing their last buffer. Every FG_thumbnail thus contains a caboose flag. A stage need not determine that the caboose arrives if the stage does not need to execute anything special at the end. If the stage needs to know whether the current round is the caboose round, it can find out by calling FG_get_caboose on the FG_thumbnail it is given. If FG_get_caboose returns 0, it is not the caboose buffer; otherwise it is.

Caboose variables

FG_num_rounds

FG_num_rounds gives the expected number of rounds in a pipeline run. FG_establish_pipeline sets it to the value of its formal parameter num_rounds, and nothing modifies it later. FG_num_rounds is a variable private to FG. FG_num_rounds determines on which buffer the source stage sets the caboose. FG uses this value only if the source sets the caboose. If another stage sets it before the source does or if the programmer forbids the source to set the caboose, this variable is irrelevant.

FG_num_caboose_setters

FG_num_caboose_setters is the number of threads that can set the caboose. This number is 1, for the source thread, unless the programmer called FG_set_caboose_setters.

FG_caboose_setter

FG_caboose_setter is the index for the stage that set the caboose. If no stage has yet set the caboose, the value is -1. FG sets this value when either the source or the programmer calls FG_set_caboose for the first time; FG generates errors for all subsequent requests to set the caboose. This variable is private to FG, but the programmer can modify its value by calling FG_set_caboose.

FG_caboose_setters

FG_caboose_setters is an array of size FG_num_caboose_setters of FG_stage_reps. A stage whose FG_stage_rep is not in this array can set the caboose, but FG will display an error if one does so. The array consists only of the source thread unless the programmer called FG_set_caboose_setters.

FG_source_sets_caboose

FG_source_sets_caboose indicates whether the source thread should set the caboose. Its value is 1 if it should and 0 otherwise. The default value is 1; a call to FG_set_caboose_setters can set it false.

FG_seen_caboose

FG_seen_caboose is an array of size FG_num_stages. FG_seen_caboose[i] is 0 if the stage has not seen the caboose and 1 if it has. FG_decider_func uses this array to determine whether it should cease to run a stage.

Public caboose functions

FG_set_caboose

See Section 2.4.

FG_get_caboose

See Section 2.4.

FG_set_caboose_setters

See Section 2.1.

int FG_is_caboose_setter(FG_stage_rep s)

FG_is_caboose_setter returns 1 if stage s has been given permission to set the caboose and 0 if it has not.

2.7 Shutting down the pipeline

FG begins the shutdown of the pipeline once a stage sets the caboose, and it completes it shortly after the caboose reaches the sink. Either the source will set the caboose on the last round or another stage can set the caboose. When giving permission for other stages to set the caboose, FG allows the programmer to prevent the source from setting the caboose. However, doing so introduces the risk that no stage ever sets the caboose. If the source sets the caboose, FG ensures that each stage is never run after it sees the caboose. It is important to recognize that if another stage sets the caboose, a risk exists that stages prior to that stage may see extra buffers and change the state. If a thread recognizes that all of its stages have seen the caboose, or the caboose is beyond all of its stages, the thread will call its cleanup function, provided one exists, and exits. Otherwise, once the caboose reaches the sink, the sink will signal all the remaining threads to cleanup and exit.

int _FG_cleanup_pipeline()

_FG_cleanup_pipeline is the function that cleans up after the pipeline. The main thread spends little time in FG_establish_pipeline and in _FG_run_pipeline, spending most of its time waiting in this function for the spawned threads to run the pipeline and join. It calls _FG_free to get rid of all allocated memory, and then it resets all counts (like FG_num_buffers) for the next time. FG_establish_pipeline calls this function after it calls and returns from _FG_run_pipeline. If any of the threads join with an error, it prints the error, and returns ERR_PTHREAD_JOIN. Otherwise, if _FG_free returns with an error, it prints the error, and returns the error. If no error occurs, it returns FG_SUCCESS.

int _FG_free()

_FG_free deallocates all memory that FG_establish_pipeline had allocated and resets all its pointers to null.

2.8 Source and sink

void _FG_source(FG_params * source_params)

_FG_source circulates the thumbnails around, sending each FG_thumbnail with a new tag, indicating the round that this FG_thumbnail represents. If the source has permission to set the caboose and no other stage has set the caboose by the time the source is ready to send out the last FG_thumbnail, then the source calls FG_set_caboose on this FG_thumbnail before it sends it out.

void _FG_source_init(FG_params * source_init_params)

_FG_source_init tags each buffer with its round for its first time through the pipeline and places it in the source's queue.

void _FG_sink(FG_params * sink_params)

_FG_sink circulates each FG_thumbnail it receives back to the source. Before it conveys each FG_thumbnail to the source, it checks if this FG_thumbnail is the caboose. If the sink receives the caboose, it signals to all other threads that they need to shut down, and then it exits itself.

Chapter 3

Deadlock

The presence of either multiple processors or threads in a program raises a risk of deadlock; the combination makes the challenge of avoiding deadlock even greater. Because FG includes threads and supports parallel processors, we must take into account the risk of deadlock. We have tried to build FG to minimize this risk whenever possible. In FG, deadlock could conceivably occur from buffers (both pipeline and auxiliary), from the way we run stages, and from not shutting down the pipeline correctly. In this section, we discuss ways in which the design of FG tries to avoid or eliminates the risk of deadlock in these areas; we also consider how deadlock can still occur.

3.1 Buffers

Auxiliary buffers

The existence of buffers shared between the threads creates the possibility of deadlock, a possibility which can be only partially avoided. While we do not guarantee that a program written with FG avoids buffer deadlock, the ways in which deadlock can still occur are avoidable if the programmer is careful. If a stage requests an auxiliary buffer and fails to get one while the other stages continue without releasing one, eventually all pipeline buffers will end up in this stage's queue, and the program will deadlock. This scenario may occur because the number of requests for auxiliary buffers at one point is greater than the total available; the simple solution is obviously creating additional auxiliary buffers. Another scenario that would achieve deadlock is if a stage accidentally fails to release an auxiliary buffer; this leak in the auxiliary buffer pool may then yield deadlock when the stage requests another auxiliary buffer later on.

Pipeline buffers

In addition, pipeline buffers also have the potential to cause deadlock. Once again, this deadlock is avoidable. Every time a stage executes, it should accept its next buffer; at the end of the round, it should convey the buffer, allowing the next stage to use it. Failure to do either of these actions may eventually result in deadlock. In the latter scenario, the stage's refusal to pass the pipeline buffers to the next stage means that the next stage will hang, waiting for buffers it never gets, and no stage will receive any of these buffers again. If a stage sometimes fails to convey the buffer, then the leak will be slower but still will eventually cause deadlock. Deadlock will also occur if stages fail to accept buffers, as these buffers will remain forever in this stage's queue and never reach subsequent stages. Because FG has its own ending stage, the programmer's last stage must also be sure to convey the buffer. Thus, accepting buffers is necessary for each stage even if the stage does not use the buffers.

We could have written FG to avoid the possibility of deadlock from pipeline buffers by calling each function and passing to it the correct buffer; in essence we could let FG be the one in charge of accepting and conveying a stage's buffer. Since we would need the thumbnail to be ready for the function before we called the stage, no stage could do steps independent of the buffer ahead of time. If the stage would do a substantial amount of work before it needed

the buffer, the stage would likely take additional time; to avoid this loss of efficiency, we require that the programmer manage the pipeline buffers, and the burden of avoiding deadlock lies on the programmer's shoulders.

3.2 Decider function and multi-stage repeat

In order to prevent multi-stage repeat from resulting in deadlock, we place limits on it. Deadlock will occur with multi-stage repeat if all buffers end up in a set of stages that will not execute until at least one stage in a different set finishes executing; since the latter set of stages does not have any of the buffers, they never finish executing. Although provable, we do not demonstrate in this thesis that having a multi-stage repeat that is the same for all multi-stage threads does not result in deadlock. We discuss, instead, some of the ways we prevent deadlock from occuring.

A maximum on multi-stage repeat

One limit we impose is that the maximum multi-stage repeat is the number of buffers in the pipeline; if the programmer chooses a number greater, FG sets it to this number. Without this limit, if one multi-stage thread exists, deadlock is certain. The deadlock that occurs does so almost immediately. Since the pipeline is linear, a buffer must pass through every stage for a given round before FG can reuse it for the next round.

- We consider a pipeline with n buffers and a multi-stage repeat of $n + 1$.

- The first stage of the multi-stage repeat will go through the first n buffers without trouble and wait for buffer $n + 1$.

- Eventually, all of these n buffers make it into the second stage's queue.

- The second stage waits for the thread to finish the first stage so the thread can execute the second stage.

- The first stage refuses to complete as it has not received buffer $n + 1$.

Since the second stage has all the buffers, the first stage waits for a buffer it will never get. Thus the thread remains in the first stage and never switches to try to execute the second stage. To avoid this deadlock, we ensure that the multi-stage repeat is at most the number of pipeline buffers.

No stage-specific repeat

Choosing to have a multi-stage thread execute one stage more than once before executing the next is necessary for FG in terms of performance; we can permit this flexibility in a number of different ways, however. One thing we could do is to allow a stage-specific repeat, in which each stage in a multi-stage thread specifies how many times it should repeat before the next stage in the thread gets a turn. A stage-specific repeat may provide a lot of flexibility, but we chose not to allow it, as deadlock can easily result. When it can result, it does so every time and very quickly. We illustrate one example in which deadlock occurs. Consider a thread consisting of two stages X and Y, with repeats x and y respectively, where $x < y$, as shown in Figure 3.1, and X appears in the pipeline before Y.

Let the number of buffers in the pipeline be n. In order to keep the example simple, we ignore the presence of any other stages, including FG's source and sink stage. Doing so will not change the outcome, as deadlock depends not on the relative speed at which the different stages pass the buffers, but instead on the relative number of buffers that each stage passes. Initially, X has all of the buffers.

- Since X appears in the pipeline first, the thread executes X first and will not execute Y until X has passed x buffers. X first passes buffer 1 to Y, and it remains in Y's queue, as it is still X's turn to execute. X has $x - 1$ buffers left to pass before it will be Y's turn.

- X likewise passes buffers 2 through $x - 1$ to Y.

- X has only one more buffer left to pass, buffer x, and it does so.

	Stage X		Stage Y	
Event	Queue size	Turns left	Queue size	Turns left
Initial	n	x	0	N/A
X sends buffer 1 to Y	$n-1$	$x-1$	1	N/A
X sends buffer 2 to Y	$n-2$	$x-2$	2	N/A
X sends buffer 3 to Y	$n-3$	$x-3$	3	N/A
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
X sends buffer $x-1$ to Y	$n-x+1$	1	$x-1$	N/A
X sends buffer x to Y	$n-x$	N/A	x	y
Y sends buffer 1 to X	$n-x+1$	N/A	$x-1$	$y-1$
Y sends buffer 2 to X	$n-x+2$	N/A	$x-2$	$y-2$
Y sends buffer 3 to X	$n-x+3$	N/A	$x-3$	$y-3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Y sends buffer $x-1$ to X	$n-1$	N/A	1	$y-x+1$
Y sends buffer x to X	n	N/A	0	$y-x$

DEADLOCK!

Figure 3.1: The progression of buffers through two stages sharing one thread, illustrating how stage-specific repeat can result in deadlock in an FG pipeline. As each stage finishes processing a buffer, the buffer goes in the queue of the next stage (where the next stage wraps around), and the number of turns before the thread switches to the other stage is decremented. N/A in the column "Turns left" indicates that the thread is executing a stage other than this one. In this example, stage X has a repeat of x, and stage Y has a repeat of y, with $x < y$. After some time, all of the buffers end up in stage X's queue, and the thread waits forever to execute a buffer in stage Y, thus causing deadlock.

- It is Y's turn and X, with a queue of $n-x$ buffers, currently does not execute. Y first processes buffer 1 and passes it on, where it eventually makes it back to X.

- Y now has only $y-1$ more turns before X gets to execute again. It executes with buffers 2 through x.

At this point, Y has $y-x$ turns left before X goes again. However, Y has processed all of its buffers in its queue and there are no more buffers for it to process. Eventually, all of these x buffers make it back to X, and X has a queue of $n-x+x$, or all n buffers. So Y will not get any more buffers until X goes, but X will not go until Y processes and conveys an additional $y-x$ buffers. Since $y > x$, Y hangs in the middle of its stage, waiting for a buffer that will not come. Thus, deadlock occurs.

No thread-specific multi-stage repeat

Another possibility would be to allow the multi-stage repeat to be thread-specific, but we rejected this possibility as well because it also can result in deadlock. Again when deadlock can arise, it does so every time and soon into the execution of the program. It can arise from a number of different combinations of multi-stage repeats. We illustrate one such example in Figure 3.2, in a pipeline consisting of 4 stages: X_1, Y_1, Y_2, and X_2, which appear in that order in the pipeline. Let thread X own stages X_1 and X_2 and have a multi-stage repeat of x. Let thread Y own stages Y_1 and Y_2 and have a multi-stage repeat of y, where $y > x$. Let the number of buffers in the pipeline be n, where $n \geq y$. We assume that each stage requires the same amount of time to execute. In actuality, their relative speeds do not matter, as they change only how quickly in time the deadlock occurs, not that it will occur.

- Initially X_1 starts out with all of the buffers. Both X_1 and Y_1 are the active stages in their respective threads.

- X_1 passes buffer 1 to Y_1.

- X_1 passes buffer 2 to Y_1 and Y_1 passes buffer 1 to Y_2.

- For the next $x - 2$ rounds, X_1 passes a buffer to Y_1 and Y_1 passes a buffer to Y_2.

At this point, there are $n - x$ buffers in X_1's queue, 1 buffer in Y_1's queue, $x - 1$ buffers in Y_2's queue, and 0 buffers in X_2's queue. Since X_1 has processed x buffers, thread X goes to execute X_2. Since X_2 has no buffers in its queue, thread X hangs in this stage. Y_1 has another buffer in its queue, so it processes it and sends it to Y_2, leaving 0 buffers in Y_1's queue and x buffers in Y_2's queue. Likewise, Y_1 still needs $y - x$ buffers before its turn is over and Y_2 can execute, and there are no buffers in its queue. So thread Y also hangs in Y_1. Now both threads X and Y are hanging in X_2 and Y_1, respectively. Since all of the buffers are in X_1 and Y_2, however, and the threads are hanging, no stage will ever pass another buffer again, and we encounter deadlock.

Event	Stage X_1		Stage Y_1		Stage Y_2		Stage X_2	
	Queue size	Turns left	Queue size	Turns left	Queue size	Turns left	Queue size	Turns left
Initial	n	x	0	y	0	N/A	0	N/A
X_1 sends buffer 1 to Y_1	$n - 1$	$x - 1$	1	y	0	N/A	0	N/A
X_1 sends buffer 2 to Y_1 and Y_1 sends buffer 1 to Y_2	$n - 2$	$x - 2$	1	$y - 1$	1	N/A	0	N/A
X_1 sends buffer 3 to Y_1 and Y_1 sends buffer 2 to Y_2	$n - 3$	$x - 3$	1	$y - 2$	2	N/A	0	N/A
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
X_1 sends buffer $x - 1$ to Y_1 and Y_1 sends buffer $x - 2$ to Y_2	$n - x + 1$	1	1	$y - x + 2$	$x - 2$	N/A	0	N/A
X_1 sends buffer x to Y_1 and Y_1 sends buffer $x - 1$ to Y_2	$n - x$	N/A	1	$y - x + 1$	$x - 1$	N/A	0	x
Y_1 sends buffer x to Y_2	$n - x$	N/A	0	$y - x$	x	N/A	0	x

DEADLOCK!

Figure 3.2: The progression of buffers through four stages, illustrating how thread-specific repeat can result in deadlock in an FG pipeline. One thread runs stages X_1 and X_2, and another thread runs stages Y_1 and Y_2. As each stage finishes processing a buffer, the buffer goes in the queue of the next stage (where the next stage wraps around), and the number of turns before the thread switches to the other stage is decremented. N/A in the column "Turns left" indicates that the thread that executes this stage is a multi-stage thread executing a different stage at the moment. In this example, stages X_1 and X_2 have a repeat of x, and stages Y_1 and Y_2 have a repeat of y, with $x < y$. After some time, all of the buffers end up in the queues of stages X_1 and Y_2, but the threads are waiting forever in stages Y_1 and X_2 for buffers. Thus deadlock results.

Static execution of stages in a multi-stage thread

In the initial design of FG, the programmer had substantially more flexibility in running the stages in a multi-stage thread; however, such flexibility left a wide open hole for deadlock, and we discarded this feature. In particular, the programmer could specify with what probability a multi-stage thread would run a particular stage. In the end, the thread would run all stages the same number of times, but it might run some stages more often in the beginning. For instance, when given a choice between a read and a write, where both stages have buffers ready, the programmer might choose the read stage 80% of the time. In order to make this scheme efficient, FG would use the programmer's rules only to choose among those stages that had buffers ready. Depending on various environmental factors, this rule meant that one day a thread might run the first stage three times and then the second stage three times, and another day it might run the first stage once and then the second stage three times. At first glance, this feature seems to be a

great idea, as it yields a new level of effi ciency and flexibility to FG. In truth, it is a great idea if FG runs only on a single-processor system.

The introduction of multiple processors enables this feature to get FG into deadlock. An essential part of programs written for parallel processors is the communication steps that exist between the multiple processors. If exactly one communication stage exists, then the program will still be free of deadlock. If multiple communication stages exist, however, any two processors will not necessarily have the same threads in the CPU at the same time. Thus, at a given point, a certain stage may have completed on one processor and not on another. As a result, when FG examines which stages have buffers ready, some processors will show some buffers as being ready that others do not. Since only one communication can be done at once, it is likely that a programmer will choose to group multiple communicate stages in one multi-stage thread. Doing so however, means that one processor may call one communicate stage while the other processor calls another, thus causing the two machines to deadlock, waiting for the corresponding communication that will not come until the other's communication has succeeded.

Chapter 4

FG Experimental Results

We obtained preliminary results of the performance of a program created by FG by porting a 4-pass threaded out-of-core columnsort application [CC02] to use FG. We did not obtain results comparing the time to solution on the part of the programmer. To get an accurate estimate, we would need two nearly identical programmers, one writing a program with FG and one writing it without. Such a study is beyond the scope of this thesis.

We ran both the original application and the application rewritten with FG on a Beowulf cluster of 32 dual 2.8-GHz Intel Xeon nodes. Each node has 4 GB RAM and an Ultra-320 36 GB hard drive. The nodes are joined by a high-speed Myrinet network. Both versions use the C stdio interface for disk I/O, the pthreads package of Linux, and standard synchronous MPI calls within threads. We used the package MPI/Pro, since it correctly supports simultaneous MPI calls from different threads.

Due to the flexibility that FG allows, we created two versions of our FG port of columnsort and ran experiments on both. The first is a close port of the original 4-pass threaded columnsort, in which the read and write stages are shared by one thread, just as the original columnsort does. The second is a slight modification of the first, in which the read and write I/O stages belong to different threads. For the purpose of this discussion, we refer to the original 4-pass threaded columnsort as Original, the first version with FG, containing a single I/O thread for both read and write stages, as FG-combined, and the second version, with different read and write I/O threads, as FG-separate. The preliminary results here compare Original to FG-combined and FG-separate.

For each version, we ran various combinations with 4, 8, and 16 processors. We ran both 4 GB per processor and 8 GB per processor runs. We did not run less than 4 GB per processor, as the original columnsort is designed for out-of-core applications, and file-caching effects will mask the out-of-core aspect of the problem for data sizes of less than 4 GB. Because we save the original input to verify each sort and we require a temporary file in addition to the output file, we need three times as much disk space as the size of the file. This requirement prevents us from running tests with more than 8 GB per processor. The results for 8 GB per processor show the same trends as those for 4 GB per processor, so we present only the 4 GB per processor results here. All three versions have runs with 3 and 4 buffers. For FG-combined and FG-separate, we use multi-stage repeats of 2, 3, and 4 (the original columnsort has no notion of a multi-stage repeat).

Figure 4.1 summarizes results comparing the three versions of columnsort for 4 GB per processor with 3 buffers and 4 buffers, respectively. For each point, we ran each program multiple times and varied the number of processors. The points represent the ratios of mean execution times per (GB per processor), i.e., each processor's execution time per GB, normalized to that of Original. The ratios are computed using the execution times of the optimal multi-stage repeat for the specified number of buffers for FG-combined. Varying the multi-stage repeat for FG-separate yielded negligible differences in execution time, as expected; for the sake of consistency, we show here the FG-separate execution times with the same multi-stage repeats as those of FG-combined. The multi-stage repeats for the figure are 3 for 3 buffers and 4 for 4 buffers.

As the figure shows, Original is always better than FG-combined, even with the best value for multi-stage repeat. The increase, however, is 5-10% of the original for 4 buffers and 10-15% for 3 buffers. FG-combined shows a greater increase in execution time compared to Original for 3 buffers than for 4 buffers. It also shows a greater increase in
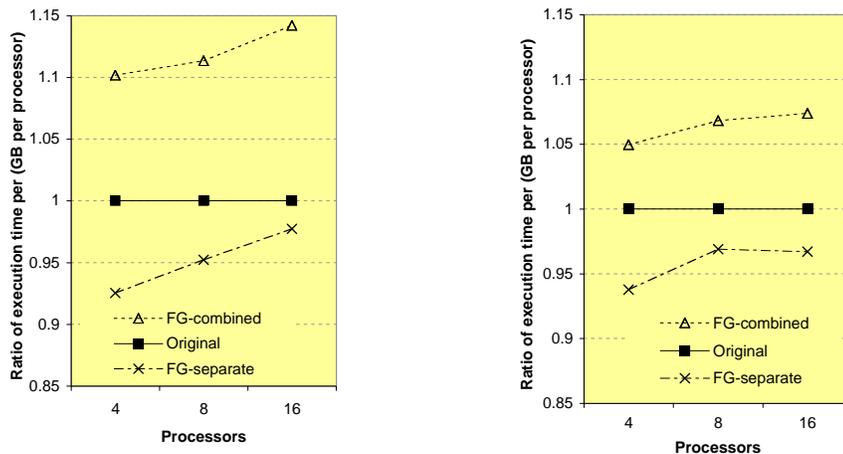
Figure 4.1: The ratio of execution times per (GB per processor), normalized to that of Original. The graph on the left is for 3 buffers, while the graph on the right is for 4 buffers. All runs are with records of size 64 bytes and buffers of 2^{21} records.

execution time for 16 processors than for 4 processors. FG-combined has a different way of alternating between reads and writes, as we discuss later, than Original. This difference may cause less filling of the pipeline for FG-combined than for Original in the case of 3 buffers and 16 processors and thus explain the differing ratios.

The difference in performance between Original and FG-combined is likely due to three factors: FG overhead, pass overlap, and the pattern of read versus write by the I/O thread. FG overhead adds some time to the execution time, as there is additional setup and cleanup for each pass. In addition, FG does extra calculations, such as to which semaphore a stage should post, that do not exist in a program like Original, where variables like semaphores are hard-coded for each thread. A second increase in time comes from pass overlap. FG-combined consists of 4 passes like Original, but only FG-combined requires the completion of a pass before the next pass begins. Thus, Original has a bit of a head start for all passes after the initial one.

The third potential increase comes from the difference in the way that Original and FG-combined alternate between reading and writing. Original, for a buffer size of n with r rounds, does n reads, $r - n$ pairs of a write followed by a read, and then n writes. FG-combined does not have this flexibility, but it instead has only the option of either doing r pairs of a read followed by a write, or with multi-stage repeat of m, r/m groups of m reads followed by m writes. In addition, Original has no synchronization between a write and a read as, in all pairs, we use the same buffer in both write and read. Thus, the I/O thread never hangs waiting for a buffer to be ready for the read it is about to perform; only writes require a wait. FG-combined, even with a multi-stage repeat, does not guarantee that a buffer will be ready for either a read or a write when the stage is called, and it does require synchronization between read and write; in particular, there are three sets of synchronization: between the write and the sink, between the sink and the source, and between the source and the read. Although both the sink and the source stages take up almost no execution time, these threads may be swapped out while another stage, such as the CPU-intensive sort stage in the algorithm, processes. If the source or the sink stage fails to process some of the buffers before the last write in the multi-stage repeat finishes, a delay will arise before the read stage can start processing. This delay can also account for some of the extra execution time that FG-combined takes.

In contrast to FG-combined, FG-separate is always faster than Original. This change is most significant for 4 processors and least significant for 16 processors, both for 3 and 4 buffers. In both FG-separate and Original, more processors or fewer buffers yield higher execution times, but the difference is more gradual for Original. The execution time of both Original and FG-separate depends on the speed of the various stages of the pipeline. As the number of processors increases, the amount of time taken by a communicate stage increases.

The increase in the communicate stage's time may explain why FG-separate's win over Original decreases for a larger number of processors. In both programs, since the write stage occurs after the communicate stage, more
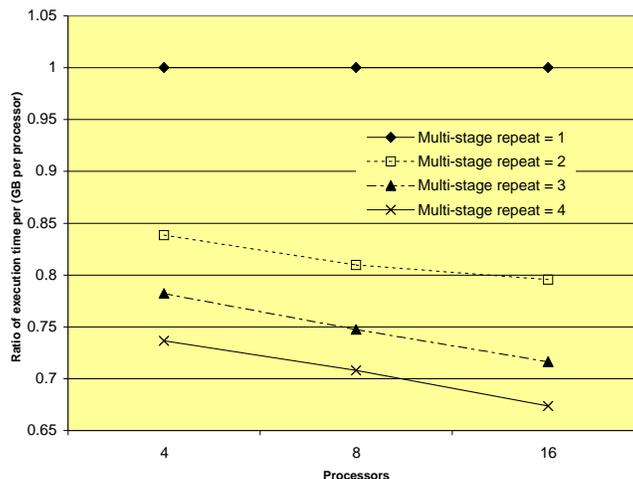
Figure 4.2: The ratio of execution times per (GB per processor), normalized to that of Original for FG-combined. The times are with 4 GB per processor and 4 buffers, with varying multi-stage repeats. All runs are with records of size 64 bytes and buffers of 2^{21} records.

processors means more time for a buffer to reach the write stage. Thus, the order in which buffers are ready for reads and writes may differ with the number of processors. In Original, the different ordering will not matter, as Original will always follow the same pattern in choosing whether to do a read or write next. In contrast, FG-separate uses the disk I/O scheduler to determine whether to read or write next. Because the order in which the reads and writes are ready varies with the number of processors, the pattern of disk reads and writes by the disk I/O scheduler may vary with the number of processors. If one pattern of reads and writes is more efficient than the other, the difference in the number of processors will be greater on FG-separate than Original.

Since the read and write calls are synchronous, we did not expect that having separate threads for them would provide a win in total execution time. Thus, no version of 4-pass threaded columnsort with separate I/O threads currently exists, aside from FG-separate. Based on the results of FG, however, a new version with separate read and write threads is in progress. We expect that this new version will give a clearer picture of how much performance loss results from FG overhead, as it will allow us to have two programs, one with FG and one without, with a more identical pattern of reads and writes. This lesson from FG is one of the primary reasons for FG. Even if a programmer chooses not to use FG in production code, he can use it without much effort to experiment with different thread structures and use the results to decide whether a change in his code that requires substantial time is likely to yield an overall gain.

Figure 4.2 summarizes the results of FG-combined for 4 buffers, 4 GB per processor, and varying multi-stage repeats. We ran the program several times with each multi-stage repeat and calculated the mean execution time of each. The figure shows the mean execution time per (GB per processor) for each multi-stage repeat, normalized to that of a multi-stage repeat of 1 (no multi-stage repeat). FG-combined is the only version of columnsort in which the value of multi-stage repeat is used to determine the order in which stages are executed.

The figure shows that the multi-stage repeat matters significantly in the execution time. As demonstrated previously in figure 4.1, the number of buffers does influence the overall execution time. Figure 4.2 illustrates that the value of multi-stage repeat matters more than the number of buffers in overall efficiency. An increase in the multi-stage repeat shows a decrease in execution time, all the way up to the maximum multi-stage repeat of 4 (since the number of buffers here is 4). In the case of 16 processors, where the communicate stage will take longer, we see that the multi-stage repeat of 4 is particularly helpful in keeping the pipeline full, yielding an improvement of more than 30%. Thus we see the importance of multi-stage repeat; without it, the buffers wait too much in queues instead of keeping the pipeline full.

The code differences between Original and FG-combined indicate that FG programs will likely be smaller, if not also less complicated. We could not easily compute an exact count of the difference in source code size because

Original is part of a larger program with parts that the FG versions lack. We estimate, however, that the source code size is 10–20% lower with FG, and experience suggests that this decrease in code size would likely apply for other programs. In addition, the difference in FG-combined and FG-separate is 5 lines out of several hundred, illustrating that modifications to the thread structure in FG prorams can be quite simple from the programmer's point of view. Making an equivalent change in Original will result in a more significant change to the source code.

Chapter 5

Conclusion

While using asynchronous threads have served to substantially improve the performance of these programs, it also adds significantly to the development and debugging time. Moreover, experimenting with the structure in an attempt to improve efficiency further can prove difficult and require several changes to the program. We designed FG to meet this very need. FG creates a framework that would allow the programmer to not only use asynchronous threads but also allow the programmer to create the program without much of the effort normally required when using asynchronous threads. Furthermore, it enables the programmer to easily modify the thread structure of the program without much effort or much change to the body of code. FG follows this common pipeline structure found often within the individual passes over the data. Like these programs, FG also passes its data from stage to stage using buffers.

FG is designed to keep efficiency high while still preventing deadlock. It permits a global repeat of stages within multi-stage threads in order to keep the pipeline full. It controls the order in which the stages are executed and prevents stages from being repeated varying numbers of times, to ensure that deadlock does not occur. FG can yield deadlock, but only if the programmer fails to use the buffers correctly. Moreover, whenever possible and especially at the time the pipeline is established, FG checks the programmer's parameters for validity and reports errors.

We have obtained preliminary results comparing an out-of-core sorting program with and without FG. Although FG does add some time to the total execution time, it is not substantial. Moreover, using FG greatly simplifies the program and should reduce debugging time. Furthermore, we used FG to modify the program's thread structure and found an improvement in performance. The changes to the program were small and much less than they would be in the original version of the program. Thus, we have already seen an example in which FG allows for easy experimentationin the program that can yield higher performance.

In preliminary tests, FG has already demonstrated itself to be useful and provide the programmer with several benefits. We have plans now to improve FG by adding various new features to it. Already plans to convert FG to C++ to enable, among other things, better error checking and control of FG's variables have begun. We also plan to expand FG to allow for other pipeline structures, such as nested pipelines, thus increasing the scope of FG substantially. FG will continue to grow and provide programmers with new opportunities.

Chapter 6

Acknowledgements

Without the help of many people, I would certainly never have finished this thesis. In particular I would like to thank my parents for their love, support, and countless sacrifices throughout the years. Without their help, I would never have come to Dartmouth and had the opportunity to write a thesis here. Sara Szkola and John Paul Reid were incredibly supportive through it all, and John Paul also willingly read my thesis repeatedly at all hours of the night. I would also like to thank the rest of my family and friends, for standing by me during the difficult weeks.

I am grateful for having had the opportunity to work with Geeta Chaudry and Elena Davidson during the course of my thesis. It was a pleasure to work with them, and their help and support was immense. I would like to also acknowledge my thesis committee who gave up their time to help me, both during and at the end of the thesis process. I wish I had had the time to implement more of their suggestions.

My deepest gratitude goes to my advisor, Professor Cormen, who has been there from the first day. Without his encouragement, I would probably have never been a computer science major and would definitely not have undertaken a senior honors project. There is no way I can express my gratitude for the countless hours he gave up to help me with this thesis, particularly during the last month.

Bibliography

[BC99]   Lauren M. Baptist and Thomas H. Cormen. Multidimensional, multiprocessor, out-of-core FFTs with
           distributed memory and parallel disks. In *Proceedings ofthe Eleventh Annual 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.