TACKLING LATENCY USING FG

Dartmouth Computer Science Technical Report TR2011-706

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Priya Natarajan

DARTMOUTH COLLEGE

Hanover, New Hampshire

September 2011

Examining Committee:

(chair) Thomas H. Cormen

David F. Kotz

M. Douglas McIlroy

Laura I. Toma

Brian W. Pogue, Ph.D.
Dean of Graduate Studies

Abstract

Applications that operate on datasets which are too big to fit in main memory, known in the literature as *external-memory* or *out-of-core* applications, store their data on one or more disks. Several of these applications make multiple passes over the data, where each pass reads data from disk, operates on it, and writes data back to disk. Compared with an in-memory operation, a disk-I/O operation takes orders of magnitude ($\approx$100,000 times) longer; that is, disk-I/O is a *high-latency* operation. Out-of-core algorithms often run on a distributed-memory cluster to take advantage of a cluster's computing power, memory, disk space, and bandwidth. By doing so, however, they introduce another high-latency operation: interprocessor communication. Efficient implementations of these algorithms access data in blocks to amortize the cost of a single data transfer over the disk or the network, and they introduce asynchrony to overlap high-latency operations and computations.

FG, short for Asynchronous Buffered Computation Design and Engineering Framework Generator, is a programming framework that helps to mitigate latency in out-of-core programs that run on distributed-memory clusters. An FG program is composed of a pipeline of stages operating on buffers. FG runs the stages asynchronously so that stages performing high-latency operations can overlap their work with other stages. FG supplies the code to create a pipeline, synchronize the stages, and manage data buffers; the user provides a straightforward function, containing only synchronous calls, for each stage.

In this thesis, we use FG to tackle latency and exploit the available parallelism in out-of-core and distributed-memory programs. We show how FG helps us design out-of-core programs and think about parallel computing in general using three instances: an out-of-core, distribution-based sorting program; an implementation of external-memory suffix arrays; and a scientific-computing application called the fast Gauss transform. FG's interaction with these real-world programs is sym-

biotic: FG enables efficient implementations of these programs, and the design of the first two of these programs pointed us toward further extensions for FG. Today's era of multicore machines compels us to harness all opportunities for parallelism that are available in a program, and so in the latter two applications, we combine FG's multithreading capabilities with the routines that OpenMP offers for in-core parallelism. In the fast Gauss transform application, we use this strategy to realize an up to 20-fold performance improvement compared with an alternate fast Gauss transform implementation. In addition, we use our experience with designing programs in FG to provide some suggestions for the next version of FG.

Contents

Chapter 1

Introduction

1.1 Latency

The term *latency* is defined as the time between when an operation starts and when it completes. Let us look at some aspects of latency through an example based on cooking. Suppose you want to cook vegetable stir-fried rice for 20 people for a potluck event. Although you volunteer enthusiastically, you later realize that you don't have a stockpot large enough to get the job done. After the initial panic, you calmly analyze the task at hand, and you conclude that 7.5 cups of raw rice will be sufficient. With a two-cup rice cooker at your disposal, you will need just four iterations. You also realize that you have a good medley of frozen vegetables for the dish and a skillet large enough so that you can prepare enough vegetables for one iteration. In the 30 minutes that you know it will take for the rice to cook in each iteration, you grab the right amount of vegetables for this much rice from the freezer and wash them in two minutes, and you take another 10–12 minutes to prepare them. After four such iterations, your job is done. In the above example, cooking rice takes a long time to finish, i.e., it is a high-latency operation, whereas gathering the vegetables and preparing them takes much less time, and so we can characterize them as low-latency operations.

We can make two more observations from the above example. First, although you did not have a stockpot large enough to handle all the cooking at once, you had enough resources to break down the problem and still finish the job in a reasonable amount of time. Second, you realized that cooking rice and preparing vegetables require independent resources, which let you overlap these operations so that the low-latency operations hid behind the high-latency operation. Surely, performing the

operations sequentially would have taken much longer.

1.2 Latency and parallel computing

As in everyday life, high-latency and low-latency operations occur frequently in the computing world. Mitigating the effects of high-latency operations, along with exploiting the available parallelism in a program are the central themes of this thesis. First, let us look at the high-latency operations that we might come across in a computer program. In *out-of-core* applications, the input datasets are too big to fit in main memory; therefore, data reside on one or more disks.[1] Accessing data from disk takes approximately 10 milliseconds to complete, which seems fairly fast, until we compare this time with the 100 nanoseconds (approximately) that it takes to fetch data from main memory. Because a disk access takes about 100,000 times longer than a main-memory access, we characterize disk I/O as a high-latency operation and a main-memory access as a low-latency operation.

Often, out-of-core applications work on datasets that are too big to fit on the hard disk of a single machine. In such situations, these applications take advantage of the computation and disk resources offered by the nodes of a distributed-memory cluster. Distributing the data and computation across the nodes of a distributed-memory cluster usually introduces another high-latency operation: interprocessor communication, which typically takes 1–10 milliseconds to complete. Because each node of a distributed-memory cluster has its own disk and its own memory, the nodes can work concurrently on distinct pieces of data, i.e., we can parallelize the computation across the nodes. In addition to distributed-memory parallelism, today's era of multicore machines and parallel-computing libraries also allows program designers to leverage the in-core parallelism inherent in an application.

1.3 Tackling latency

From our previous discussions, we can appreciate that we must reduce the effect of high-latency operations in a program to achieve an efficient implementation. Implementors of out-of-core programs often employ three techniques to reduce the effect of latency on their implementation.

[1] Out-of-core applications are also known as external-memory applications in the literature.

1. Access data in blocks,[2] where the block size is small relative to the total input size, but large relative to an individual data element. Usually, the block size is such that many blocks can fit in main memory. In our example, we cook two cups of rice at a time and not just one grain. Similarly, during a disk access, we read or write a block that is many kilobytes (or even megabytes) in size instead of a single data element. Using block accesses, we can amortize the cost of transferring data from disk to memory. Likewise, during interprocessor communication, we transfer data in blocks from one node's memory to another node's memory.

2. Design algorithms to minimize the number of disk accesses or network transfers. See the survey by Vitter [56], for example, which describes many out-of-core algorithms that access data in blocks while minimizing the number of such accesses. Indeed, the Parallel Disk Model by Vitter and Shriver [57] is dedicated to the design of such algorithms.

3. Overlap disk I/O and interprocessor communication with computation. Usually, when a process accesses the disk or the network, it blocks[3] and yields the CPU so that the CPU is free to perform other operations. Hence, we can overlap disk I/O and interprocessor communication with in-memory computation.

In order to overlap operations, we introduce asynchrony in our program, either by introducing asynchronous I/O and communication operations in a single-threaded program or by having multiple threads in our program, with each thread running a synchronous function. In the first case, we have to statically schedule the high-latency operations so that they overlap with computation operations, whereas in a multithreaded program, the operating system dynamically schedules the threads so that when a thread blocks, the OS runs another thread that is ready to run. Both of these approaches are cumbersome and error-prone to program, and the peripheral code—often termed *glue*—required to introduce asynchrony and access data in blocks is usually unrelated to the underlying application.

[2]The word "block" can serve both as a noun, where it means a chunk, and as a verb, where it means to stall. Unfortunately, both grammatical forms of the word bear relevance to our work; the intended meaning of an occurrence will, however, be clear from its context. In this particular case, we mean a chunk.

[3]And here, we mean to stall.

1.4 Using FG to tackle latency

In this thesis, we will use FG [14, 18], a programming framework for pipeline-structured programs, to mitigate the effects of latency in out-of-core programs. Chapter 2 describes the features of FG in detail, but in short, FG provides the code to introduce asynchrony and manage data blocks. Programmers use FG to model their out-of-core applications as a pipeline of software stages, where the programmer writes a simple, synchronous function for each stage. FG runs each stage of the pipeline in its own thread, thus introducing asynchrony, and it provides methods to circulate buffers to hold data through the pipeline. In a nutshell, FG provides the glue, leaving the tasks more relevant to the actual application to the programmer. Initial papers on FG [14, 18] suggest that FG makes it faster, simpler, and more efficient to implement parallel programs.

We will look at how FG helps us design out-of-core programs and think about parallel computing in general using three instances: an out-of-core, distribution-based sorting program; an implementation of external-memory suffix arrays; and a scientific-computing application called the fast Gauss transform.

We have implemented an out-of-core, distribution-based sorting program using FG, which we nickname "dsort." Until dsort, all programs written using FG could be implemented efficiently with a single pipeline on each node of a cluster. As Chapter 3 will detail, dsort exhibits disk I/O and interprocessor communication patterns that vary based on the input dataset. In particular, we observed that in dsort, nodes might send and receive data at different rates during interprocessor communication, and that a stage might consume data from different streams at varying rates and it might produce data at some other rate. Dsort's design requirements helped us introduce two new pipeline structures in FG, disjoint pipelines and intersecting pipelines, to handle the situations that we just described. Chapter 3 will show how these new pipeline structures made it simpler to design and implement dsort, while being efficient in practice. We could have programmed dsort using a single, linear pipeline, but doing so would have left a substantial portion of the asynchronous programming burden on the user, which is against FG's principles. Furthermore, the stage functions would have been unwieldy to program, which made us realize that FG's design at the time was insufficient for implementing dsort.

FG works well for out-of-core sorting in a shared-memory setting, too. Our implementa-

tion of out-of-core sorting in shared-memory using FG outperforms an implementation that uses STXXL [20, 22], which is a library for out-of-core programs.

Disjoint and intersecting pipelines worked well for dsort, but we wondered whether the utility of these pipeline structures would end with dsort. Our doubts were quelled when we implemented external-memory suffix arrays [29, 41] using FG. Suffix arrays are useful in pattern matching, text compression, and computational biology. Not only did we use intersecting pipelines extensively in this project, but we were also able to combine pipelines in more innovative ways than we did for dsort. We were able to efficiently tackle latency using inventive FG pipeline structures in this external-memory algorithm. With FG taking care of the heavy-duty multithreaded code under the hood, we had to write code only specific to the application. The algorithm that we used for constructing external-memory suffix arrays is a recursive algorithm, the first such attempt using FG. The suffix-array algorithm that we used was designed to work on a single machine so that we did not require any interprocessor communication, but we performed almost 2 terabytes of I/O for a 4-gigabyte input size. In addition to handling disk I/O efficiently and multithreading the code using FG, we parallelized the in-memory computations to utilize all the available cores in the underlying machine. We used OpenMP and the parallel library from libstdc++ [40] for in-core parallelism. This project also pointed us to an extension for FG, and it revealed how some existing FG features might be redundant. Chapter 4 covers these points in more detail.

The fast Gauss transform (FGT) [27, 28] approximates an n-element sum at m target locations in $O(m+n)$ time instead of the $O(mn)$ time required for exact computation. The fields of computational physics and computational finance, for example, are interested in such computations. Of the two available methods for computing the fast Gauss transform that we came across, we have implemented the algorithm using one technique, both in shared-memory and distributed-memory settings. The main challenge in the shared-memory implementation, which does not use FG, was to identify ways to reuse costly mathematical computations and to locate parallel regions in the algorithm. The distributed-memory implementation requires interprocessor communication, and so we used FG to overlap communication with computation, in addition to applying the strategies that we used in our shared-memory implementation. As in our suffix-array implementation, we were able to use OpenMP parallel regions in our stage functions for in-core parallelism. We also used FORTRAN-based BLAS routines [7] for some vector computations. FG's generic design allowed

us to combine multithreading with OpenMP and FORTRAN-based routines to speed up the running time by factors of up to 20 compared with an alternate FGT implementation. This project showed how we can also use FG stages as a signaling mechanism. Chapter 5 elaborates on our implementation of the fast Gauss transform.

1.5 Contributions of this thesis

This thesis uses the FG programming framework to overlap high-latency operations with other operations in out-of-core and distributed-memory applications. Although FG provides all the code for multithreading an application, the onus of identifying the available parallelism in an application and coming up with a good pipeline design in FG lies with us.

1. While designing an out-of-core, distribution-based sorting program using FG (nicknamed "dsort"), we identified ways to advance FG from supporting just single, linear pipelines to multiple disjoint pipelines and multiple pipelines that intersect at a common stage. Using these new pipeline structures, we were able to implement dsort efficiently, despite its disadvantages of having dynamic I/O and communication patterns.

2. Our implementation of out-of-core sorting in a shared-memory setting using FG is faster by 9.6%–16.3% (approximately) compared with an STXXL-based implementation.

3. Our implementation of external-memory suffix arrays exercised FG's intersecting pipelines in ways that had never been attempted before. The complex FG pipeline structures that we designed for this algorithm also performed well. For the first time, we implemented a recursive algorithm using FG, and we used OpenMP along with FG to fully utilize all the cores of a multicore machine.

4. We have implemented the fast Gauss transform in a distributed-memory setting that uses FG to overlap communication with computation. In this project, we used FORTRAN-based BLAS routines for vector computation, and we used OpenMP to leverage the in-core parallelism offered by the algorithm. We saw speedups in running time by factors of up to 20 compared with an alternate FGT implementation.

5. We have used our experience in designing out-of-core programs using FG to extend FG with additional pipeline structures. We have also identified some FG features that might be redundant, and thus can be removed from the next version of FG.

Chapter 2

FG Introduction

In this chapter, we recap the challenges of an out-of-core program, followed by a description of the FG programming environment. After elaborating on the basic components of an FG program, this chapter continues with some additional structural and programming features that FG provides.

2.1 Birth of FG

The survey by Vitter [56] covers a number of external-memory or out-of-core applications. The input datasets of these applications are too big to fit in the main memory of a single computer and sometimes even that of many computers. Therefore, data resides on one or more disks. Several of these algorithms make multiple passes over the data, where each pass usually involves reading data from disk to memory, performing some computation on it, and writing the results of the computation back from memory to disk. If the data is distributed across the nodes of a distributed-memory cluster, a pass might also require interprocessor communication amid disk-I/O and computation operations. As we saw in the previous chapter, both disk I/O and interprocessor communication are high-latency operations, which can strongly influence the overall running time. In order to mitigate the effects of high-latency operations, implementors of out-of-core applications resort to two coding techniques. First, they access data in blocks to amortize the cost of a single disk I/O or interprocessor communication operation. Second, they introduce asynchrony to overlap high-latency operations with other operations.

The two common ways to introduce asynchrony—using asynchronous I/O and communication

operations in a single-threaded application, and multithreading an application—are both cumbersome and error-prone to program. Furthermore, the glue, which denotes the peripheral code required to introduce asynchrony and access data in blocks, is usually unrelated to the underlying application. We note, however, that for the most part, the glue code is reusable across different applications.

A closer look at some programs for permuting, sorting, and FFTs for out-of-core data [12, 15, 16] that use either of the asynchronous approaches reveals that many external-memory algorithms share yet another property: that of a pipeline. A single pass in these algorithms exhibits a pipeline structure, and operating this pipeline on a different data block each time (to exhaust the input) completes the pass. Although the pipeline for a sorting program differs from that for a permuting program, the code that sets up and runs the pipeline for one program will work for the other program after a few minor changes. These observations helped in crafting FG [14] as a programming environment for applications that fit the pipeline model, thus relieving programmers of the burden of writing the glue.

2.2 FG basics

An FG program is composed of a pipeline of stages, each of which is mapped to a user-defined *stage function* written in C or C++. Because FG maps each stage to its own thread, the stages can run asynchronously to overlap high-latency operations with other operations. Since FG handles the asynchrony, a stage function need only be a straightforward one, containing synchronous calls. The user also specifies the number and size of buffers that should circulate through the pipeline; these buffers are referred to as *pipeline buffers*. Ideally, the buffer size should be the same as the block size in outer levels of memory hierarchy. FG takes care of actually creating the buffers, directing them through the pipeline, and ultimately destroying them.

To aid in the smooth running of a pipeline, FG adds two stages, a source stage at the start and a sink stage at the end, to every pipeline, as shown in Figure 2.1. The source stage emits buffers to the first user-defined stage in the pipeline, starting a new *round* with each emission; FG associates a *round number*, starting from 0, with each buffer. An FG stage *accepts* a buffer from its predecessor when it needs a buffer to work on; similarly, a stage *conveys* a buffer to its successor when it is done

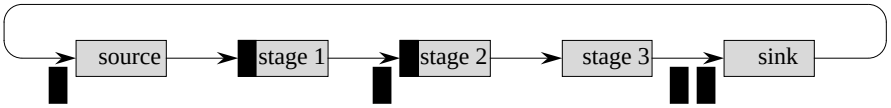


Figure 2.1: A standard FG pipeline comprising a source, a sink, and three other stages. Each black rectangle represents a buffer. Where a buffer appears inside a stage, the stage is currently working on that buffer. Buffers in queues appear below the arrows between stages. The arrow from the sink to the source represents how buffers are recycled.

working on the buffer. When a buffer reaches the sink, it is recycled back to the source stage to be reused in the pipeline with a new round number. The last buffer to go through a pipeline is called the *caboose*, indicated by a flag. FG shuts down the pipeline when the caboose reaches the sink.

Although we have been referring to buffers traversing a pipeline, it is actually *thumbnails* that go through a pipeline. A thumbnail contains a pointer to the actual buffer, among other useful information such as the round number and the caboose flag. FG maintains two queues, an incoming queue and an outgoing queue, of thumbnails per stage. FG completes an `accept_thumbnail` call from a stage by dequeuing the stage's incoming queue; the call blocks if a stage's incoming queue is empty. Similarly, during a `convey_thumbnail` call from a stage, FG enqueues a thumbnail in the stage's outgoing queue. The enqueue operation on a queue of thumbnails never blocks because each queue is big enough to hold the number of buffers that was specified by the user.

If the user knows beforehand the number of rounds for which the pipeline should run, she can specify this number during pipeline setup. For example, a program that reads and processes a 1 GB file using buffers of size 1 MB will need exactly 1024 rounds to finish; the source stage will, therefore, set the caboose on the buffer with round number 1023. Sometimes, however, the user might know when to finish the program only at run time. Going back to our previous example, if we do not know the size of the input file beforehand, we would not be able to set the number of rounds on the pipeline. The stage that reads data into a buffer would be capable of identifying the last buffer when it reads the end of file marker. Therefore, the caboose might need to be set by stages other than the source stage. In order to accommodate such situations, FG provides a method that sets the caboose flag on a thumbnail; that is, the caboose can be set on the fly in FG.

FG maps each stage to its own thread via calls to POSIX pthreads [31]. Although the user does not make explicit `pthread_create` calls, she does provide the functions that the threads call. Without FG, the burden of spawning threads, running them as a pipeline, and finally destroying

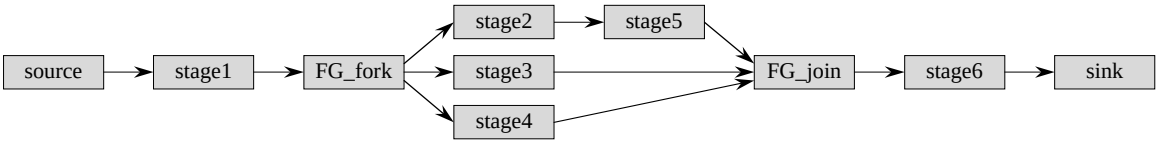


Figure 2.2: An example of FG's fork-join pipeline structure. FG does not spawn the stages named FG_fork and FG_join; the stages are shown only for illustrative purposes. During pipeline setup, the user informs FG that the pipeline should fork at stage1 and join at stage6 using specific FG calls. Buffers and sink-to-source connections are not shown in this figure.

them would be on the programmer; this work would be in addition to that of managing buffers. With FG, a user can concentrate on the specifics of the application without having to worry about writing the cumbersome and error-prone glue. A paper by Davidson and Cormen [14] elaborates on the details of this section and also shows the results of using FG for some out-of-core programs.

2.3 Additional FG features

Using FG, we can deviate from the linear pipeline structure and create pipelines with forks and joins. We can even create pipelines to represent a DAG structure or a macro [14, 18].

The *fork-join* construct in FG, shown in Figure 2.2, allows a pipeline to split at one stage and merge later. FG feeds the multiple successors of a forked stage in a round-robin manner; the user can specify the order, first-come first-served (default) or round-robin, in which a stage should accept from its multiple predecessors.

FG allows pipelines wherein a stage may convey its buffers to any other stage, rather than to only its linear successor, as illustrated in Figure 2.3, provided that the structure of the pipeline represents a DAG. The user need not tell FG in advance of the stage jumps that she intends to make; a stage can decide where to send a buffer on the fly. A stage can, therefore, convey a buffer directly to the sink instead of making the buffer go through the remaining stages in the pipeline. A user might want to convey a buffer to the sink if she decides that it does not contain useful data to work on.

In a subsequent chapter, we shall see that FG also allows disjoint, intersecting, and virtual pipelines. How these structures came about and why they are useful will be clearer in the context of the material presented therein. An outline of FG is incomplete without a mention of these types of pipelines, however. The FG tutorial [13] also discusses macros, hard barriers, and soft barriers.

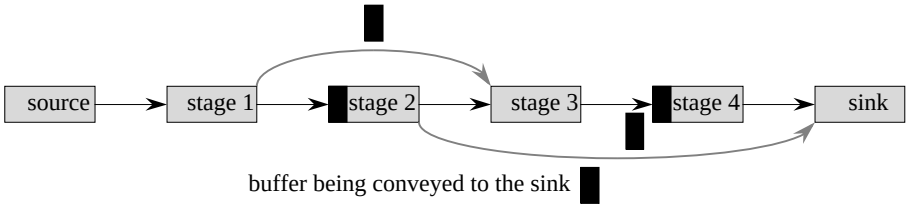


Figure 2.3: A DAG pipeline structure in FG. Stage1 can decide at run time whether it wants to convey a buffer to stage2 or stage3; the figure also illustrates how a stage might decide to convey a buffer to the sink stage. The thicker, gray arrows represent decisions taken at run time; the user need not inform FG of the stage jumps in advance.

An FG *macro* is usually composed of several stages and is similar to a subroutine in a program. A *hard barrier* splits a pipeline into two mini-pipelines by making sure that one part of a pipeline runs completely before the other part. A stage is designated a *soft barrier* if it must accept more than one buffer per round to complete its work.

In addition to these structural features that help with program design, FG provides some features that help during program implementation. Sometimes, a stage might require temporary space to carry out its work. Consider a stage function that permutes the contents of the incoming data buffer; if the permutation is not done in-place, the stage needs another buffer to store the results of the permutation. One approach for the user is to allocate memory at the beginning of the stage function, carry out the stage function using this memory as extra space, copy the results of the permutation from the user-allocated memory to the pipeline buffer, and finally deallocate the user-acquired memory before conveying the thumbnail. That is, this approach involves extra copying. The other approach is for the user to utilize FG's *auxiliary thumbnails* that have been designed expressly for the purpose of providing temporary space to stages that cannot carry out their work in-place. During pipeline setup, the user can specify the number of auxiliary buffers that she anticipates using in her program; FG creates a pool with that many auxiliary thumbnails, which are in addition to the pipeline thumbnails. FG destroys the auxiliary buffers during pipeline shutdown. Auxiliary buffers are always as big as the pipeline buffers, and a stage is free to accept as many auxiliary buffers as it might require. As we saw in our example, the final result of a computation might end up in the auxiliary buffer, whereas it is the pipeline buffers that travel from one stage to the next. FG aids in ensuring that correct data flows to the next stage by providing a method to swap the buffer pointers of a pipeline and auxiliary thumbnail. After swapping the buffers, a stage can convey the

same pipeline thumbnail that it accepted. The swap method, made possible by FG's thumbnail design, saves the user from having to copy the contents from the auxiliary buffer to the pipeline buffer. That is, auxiliary thumbnails are convenient and efficient to use.

The prototype of an FG stage function allows *stage parameters* for information from outside to enter the pipeline. For example, a user can open an input file before setting up the pipeline and pass the file pointer as a stage parameter to the read stage function. FG also provides *thumbnail parameters* to enable control information to flow from one stage to the other; thumbnail parameters exist as part of the thumbnail.

Chapter 3

Dsort: Out-of-Core Distribution-Based Sorting Using FG

In this chapter, we present the design and implementation of our out-of-core, distribution-based, sorting program on a distributed-memory cluster, nicknamed "dsort," using FG. To lay a foundation for dsort, we begin with the general idea behind distribution-based sorting. Dsort begins with a sampling step followed by two passes; the descriptions of each of the two passes will introduce us to new pipeline structures in FG. In pass 1, we will learn about disjoint pipelines in FG, and pass 2 will acquaint us with FG's intersecting and virtual pipeline structures. These structures have a cause-effect relationship with dsort; we conceived of these extensions to FG while designing dsort, and these structures, in turn, enabled a much cleaner design of dsort than would have been possible earlier.

3.1 Distribution-based sorting

Distribution-based algorithms form one of the major paradigms for out-of-core sorting, along with merging-based and oblivious algorithms. A distribution-based sorting algorithm for out-of-core data usually comprises three or four steps. The input, containing N keys, resides on P nodes of a distributed-memory cluster. The first step samples the input data to select $P-1$ keys, called *splitters*. Based on the splitter values, the second step partitions the input into P sets, $S_0, S_1, \ldots, S_{P-1}$, such that all keys in set S_i are less than or equal to all keys in set S_{i+1}; set i resides on the disk of node i.

The third step sorts each partition, and the fourth step, if implemented, load-balances the output among the P nodes of the cluster. Because the I/O and communication patterns generated by a distribution-based algorithm vary depending on the data to be sorted, an implementation should be robust against inputs that can lead to badly skewed patterns of these high-latency operations.

3.2 Dsort

We have implemented dsort in three passes. Pass 0 samples the input and selects splitters, which pass 1 uses to partition the input. At the end of pass 1, each node contains several sorted runs. Pass 2 merges the sorted runs to create a single sorted sequence, and it also performs load balancing and creates striped output. We assume that the input contains N records,[1] distributed on P nodes of a cluster.

By "striped output," we mean that it appears in the order as defined in the Parallel Disk Model [57]. The records reside in fixed-size blocks, which are assigned in round-robin order to the disks in the cluster. If each node has one disk and the block size is B, the first B records reside on node 0's disk, the next B records on node 1's disk, and so on; a block on node $P-1$ is followed by a block on node 0.

3.2.1 Pass 0: Selecting splitters

The preprocessing step selects a set of $P-1$ key values, known as *splitters*, which are used by pass 1 to partition the input. We find the splitters using the technique of *oversampling*, as done by Blelloch et al. [8] and by Seshadri and Naughton [45]. From among its N/P records, each node selects a uniform random sample of size s; the parameter s is known as the *oversampling ratio*, and the records that form the sample are called *candidates*. Each node then sends the set of its s candidates to node 0, which collects the sP candidates and sorts them locally. The $P-1$ final splitters are the records at ranks $s, 2s, \ldots, (P-1)s$ in the sorted list of candidates. Node 0 then broadcasts the $P-1$ splitters to all other nodes.

This method of selecting splitters works fairly well if the input keys are nicely distributed, but it can lead to unbalanced partition sizes if the input keys lie within a small range. In the worst

[1] A record consists of a key along with satellite data.

scenario, all input keys might be equal, causing a single node to sort all N records, which the node might not be able to handle. In order to ensure mostly balanced partition sizes for all types of inputs, we extend each candidate's key with two additional fields: the number of the node (0 to $P-1$) that the candidate comes from, and the candidate's offset (0 to $N/P-1$) within the node. With these extensions, all keys are unique. During pass 1, we extend each record's key in the same manner and decide the record's destination node by comparing its extended key with the splitters. The original keys are extended only to decide a record's partition during passes 0 and 1; the extended parts are discarded immediately after the decision has been made, so that only the original parts of the record are ever stored in a buffer or on disk.

3.2.2 Pass 1: Partitioning and creating sorted runs

Pass 1 partitions the N input records into P partitions using the splitters that have just been selected and broadcast. Because the ith partition, for $i = 0, 1, \ldots, P-1$, belongs to node i, pass 1 distributes the records to their destination nodes using interprocessor communication. Each node stores its partition, as several runs of sorted data, on disk.

Prior to envisioning the idea of disjoint pipelines in FG, we had intended for each node to run a copy[2] of the pipeline structure shown in Figure 3.1, to complete pass 1. Assuming that a buffer can hold β records, FG would run the pipeline for $N/P\beta$ rounds on each node. Below, we outline the work of each stage, per round, in the pipeline.

Read stage: Accept a buffer[3] from the source stage, read β records from the disk into the buffer, and convey the buffer.

Permute stage: Accept a buffer from the read stage and use the splitters to permute the buffer's records into an auxiliary buffer, such that all records belonging to the same partition appear contiguously in the auxiliary buffer. Note that partition sizes may vary. Convey the buffer after swapping it with the auxiliary buffer.

[2]Running "a copy of the pipeline on each node" means that each node runs an instance of an FG pipeline using its own copy of the FG library. On any node, FG is cognizant only of the pipeline running on that node; FG, in and of itself, cannot communicate between pipelines running on different nodes. A user can, however, communicate data between pipelines running on different nodes if she implements a stage to participate in interprocessor communication, using, for example, calls to Message Passing Interface (MPI) functions.

[3]Recall that a stage actually accepts and conveys a thumbnail, which in turn, contains a pointer to the buffer. For the sake of convenience, we say that a stage accepts a buffer.

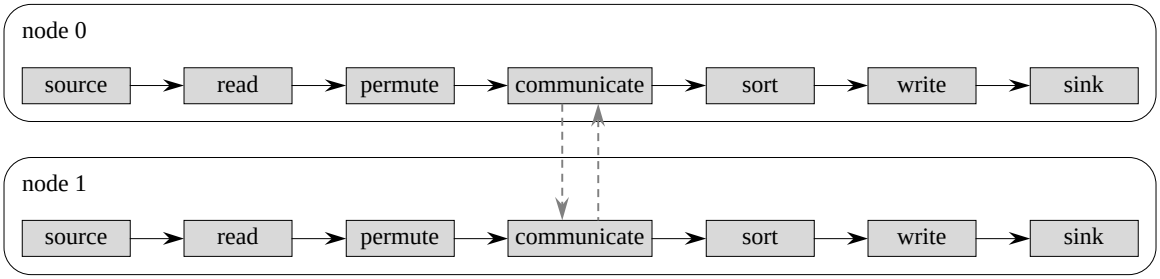


Figure 3.1: The pipeline structure for implementing dsort pass 1 using a single linear pipeline. The dashed arrows between the communicate stages indicate interprocessor communication. The figure shows a copy of the pipeline on two nodes only for illustration purposes; in general, there can be any number of nodes, each running a copy of the above pipeline.

Communicate stage: After accepting a buffer from the permute stage, this stage distributes the records in each partition to their target node, and it also collects the records destined for the node that it is running on.

In order to distribute the partitions to their respective destinations, this stage sends two sets of P messages. In the first set, this stage sends, to each node, the number of records that it should expect in this round from this node. The stage then sends all the records that it has belonging to partition i to node i, for all $i = 0, 1, \ldots, P-1$. This stage also receives data into the buffer until the buffer is full, before conveying the buffer to the next stage.

Sort stage: Accept a buffer from the previous stage, sort the elements in the buffer, and convey the buffer.

Write stage: Accept a buffer containing sorted records and write it to disk; convey the buffer to the sink stage.

Therefore, at the end of pass 1, each node contains sorted runs of all records that belong to its partition.

Let us now see why the linear pipeline model proved to be insufficient for pass 1 to run smoothly, some of our initial attempts at a solution, and how FG's disjoint pipelines contributed to a much cleaner implementation. In the discussion that follows, we will address the issues in the context of node i, and in one round, say r, of the pipeline running on node i.

In the linear pipeline above, the communicate stage acts as both a sender and a receiver. As

a sender, the stage *always* distributes a buffer's worth of records in each round. As a receiver, however, the stage might need to collect more or less than a buffer's worth of data in each round. How much data the stage receives in a round depends solely on the input data that is currently being processed over all the P nodes.

Suppose that, in a round, the communicate stage receives fewer than β records; the stage can then proceed in one of two ways: either convey the partially filled buffer, or wait for the buffer to fill up before conveying it. Because the buffer that the communicate stage conveys will ultimately be written to disk, conveying partially filled buffers is inefficient. It is, therefore, beneficial for the stage to wait for the buffer to fill up. The next set of messages,[4] however, might send more records to node i than the remaining capacity of the buffer. That is, in the same round, we might also have to prepare for the communicate stage to receive more than a buffer's worth of records.

If the communicate stage receives more than β records in a round, it can store the first β records in the buffer that the stage accepted. The stage must, however, find extra buffers to stow away the remaining[5] records, and it must also ensure that these buffers get conveyed to the sort stage. One immediate thought was to solve the problem using FG's auxiliary buffers, because auxiliary buffers offer extra storage, and a stage can accept as many auxiliary buffers as it needs. Just being able to store the received data is not enough, however; we must also be able to convey all the auxiliary buffers that we accept. Using auxiliary buffers was not a viable solution for us for two reasons: first, we can swap at most one auxiliary buffer with the incoming buffer, and second, we would not want to swap the incoming buffer because it, too, contains received data.

Because FG already provided auxiliary buffers, we came up with the idea of introducing Pipeline/Auxiliary Transform buffers, or PAT buffers, in FG. PAT buffers, as their name suggests, would be capable of acting both as pipeline and auxiliary buffers.[6] That is, after using up the pipeline buffer that came in, a stage could accept multiple PAT buffers in a round as auxiliary buffers, but convey each of them as a pipeline buffer to the next stage.

PAT buffers offered a solution to the conveying problem that auxiliary buffers posed, but not without introducing a few problems of their own. Because these buffers get injected in the middle

[4] Presumably, some of the other nodes were able to proceed to the next round; node i, however, is still stuck in round r.

[5] In the worst case, the communicate stage on a node might receive $P\beta$ records.

[6] Elena Davidson also observed that the name was the same as the androgyne character "Pat" in old Saturday Night Live sketches.

of the pipeline, they do not carry round numbers,[7] which is a hindrance to stages that use the round-number information in their implementation. For example, the read and write stages use the round number to determine the file offset for reading and writing. One might argue that a stage that accepts PAT buffers—the communicate stage in dsort, for example—could coordinate with the source stage to assign successive round numbers to buffers. By virtue of the coordination, however, the round numbers emitted by the source stage need not necessarily be consecutive, leading to gaps in file offsets calculated by the read stage. In order to ensure that both the source stage and the communicate stage emit consecutive round numbers,[8] we could probably make the two stages use separate counters for round numbers. Now, although the input file on a node will be read completely, we run the risk of FG shutting down the pipeline prematurely. By "prematurely," we mean that FG might shut down the pipeline before a node has finished writing its output. Let us explore why. Given that our linear pipeline was set to run for $N/(P\beta)$ rounds, the buffer with round number $N/(P\beta) - 1$ is the caboose. Even in the last round, the communicate stage might accept PAT buffers and convey them after the caboose buffer, which will reach the sink stage before the others. Thus, when the sink stage sees the caboose, it is possible that some PAT buffers are still being processed by the sort or write stages. FG would, however, shut down the pipeline, oblivious to the buffers that follow the caboose buffer. Therefore, PAT buffers provided a flawed solution.

From the above discussion, we notice that what is essentially required is that in each round, a stage should be able to send data at a different rate than at which it receives data. Restricted by FG's single linear pipeline model, we tried to handle the sending and receiving in a single stage, which was clearly not ideal. If we could "split" the pipeline into two pipelines at the communicate stage, as shown in Figure 3.2, and run these pipelines concurrently, each node would be able to send and receive data at different rates. Based on these observations, FG was extended to provide *disjoint pipelines* that can run independently, but concurrently, on each node. A user can now create multiple linear pipelines and pass these pipelines to FG's *pipeline manager*, which takes care of running the pipelines and shutting them down later. Each pipeline can be set to run for a different number of rounds than the other pipelines, and it can have its own number and sizes of buffers. Although the pipelines run concurrently on each node, each pipeline can run at its own speed, independent of the

[7] Only the source stage assigns round numbers to buffers.

[8] Note that, because the round numbers generated by the communicate stage affect the offset in the output file, we must ensure that these round numbers, too, are consecutive.

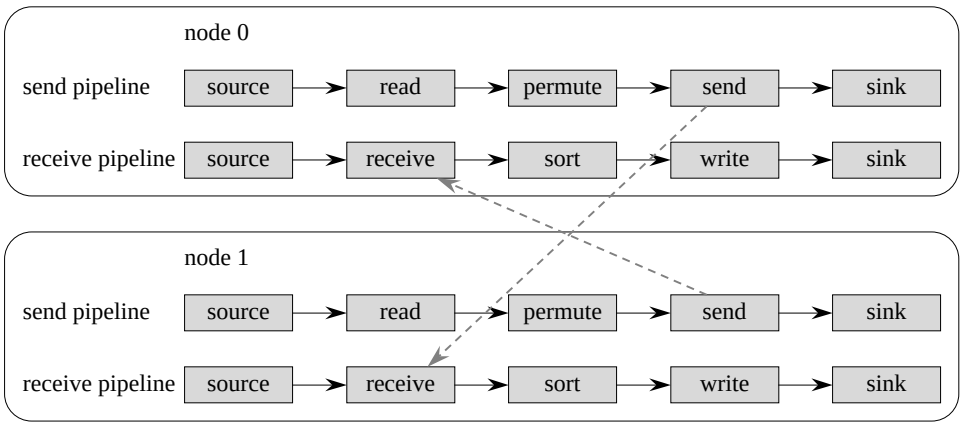


Figure 3.2: Disjoint pipelines that were run in each node for implementing dsort pass 1 using FG. As before, a copy of the pipelines runs in as many nodes as required, though the figure shows only two nodes. Although the two pipelines run concurrently on a node, each pipeline can be set to have a different number and sizes of buffers and it can progress independently of the other pipeline. The dashed arrows represent interprocessor communication; on each node, the speed of the send and receive pipelines is determined in part by the amount of data sent and received.

other pipelines.

In dsort pass 1, we used the pipeline structures shown in Figure 3.2; we set both pipelines to have the same buffer size. On each node, the "send pipeline" ran long enough to exhaust the input on the node, and the "receive pipeline" ran until all records reached their destination node and were written to disk. The stage functions for the read, permute, sort, and write stages are the same as in the single linear pipeline case, except that the permute stage extends each record's key with the node number and record number in order to decide the record's target node. The send stage, as its name suggests, participates in the sending part of the communication; we use MPI for interprocessor communication. In each round, the send stage sends two sets of P messages as before; it tags the messages in the first set as COUNT_MSG, and those in the second set as DATA_MSG. In the last round, the send stage sends an extra set of P messages (one to each node, including itself), each tagged as a CABOOSE_MSG; the send pipeline on a node uses these messages to inform all the receive pipelines to not expect any more data from that node.

The receive stage takes care of receiving the records that fall in the node's partition. After receiving the data that arrives by interprocessor communication into a pre-allocated temporary buffer, we copy the data from the temporary buffer into a pipeline buffer. When the pipeline buffer fills up (which could possibly be after several receive calls), we convey it to the next stage. Upon seeing

the Pth message tagged as a CABOOSE_MSG, the receive stage sets the caboose on the pipeline buffer that it is currently using, and conveys the buffer to the sort stage.

3.2.3 Pass 2: Merging sorted runs and creating striped output

On each node, pass 2 merges the sorted runs that were created in pass 1, and it load-balances the output. As we did for pass 1, we first describe our initial attempt at implementing pass 2 using FG, and then elaborate upon how the intersecting and virtual pipeline structures in FG led to a much cleaner design and implementation.

Although our method of selecting splitters, combined with our key-extension trick, works quite well, some nodes are still bound to end up with more records in their partition than others after pass 1. That is, we were faced with the same problem of uneven send and receive rates among nodes during the final load-balancing in pass 2. We now know that FG's disjoint pipelines can handle this situation well. The only missing piece, of course, is merging the sorted runs, for which we decided at first to use a single stage, resulting in the pipeline structures as shown in Figure 3.3. In these pipelines, the send, receive, and write stages work in the same manner as they did in pass 1. The merge stage, however, was to both read and merge sorted runs. Why did we not have a separate stage for reading the sorted runs? Note that the merge stage requires a block from *each* sorted run, at all times, in order to be able to create merged output. That is, when the merge stage exhausts a block from sorted run j, it cannot proceed until the next block from sorted run j has been read. The order in which blocks from various sorted runs will be consumed on a node are not predetermined. Therefore, the read stage (if there were one) would not know in advance the file offset from where to read data. Although the read stage could read an initial block from each sorted run and pass these blocks to the merge stage, it would need constant feedback from the merge stage to know which succeeding blocks to read. Because only the merge stage knows which data it needs next, we decided to delegate the responsibilities of reading blocks of sorted runs from disk, and merging them, to the same stage.

Again, limited by FG's existing pipeline structures, we had to combine a high-latency operation (reading sorted runs), and a computation operation (merging sorted runs) in a single stage, which affects performance. There seemed to be one promising solution for decoupling the two operations: in addition to the pipelines shown in the earlier figure, have as many pipelines as the number of

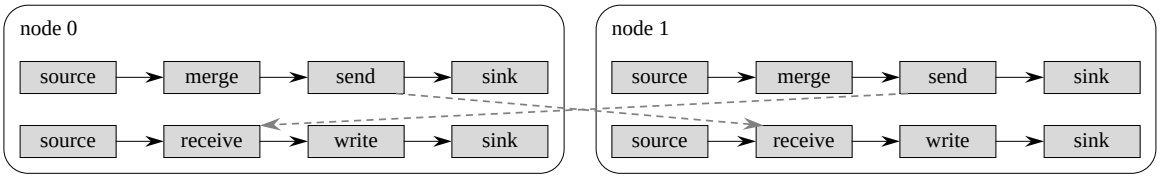


Figure 3.3: Pipeline structures illustrating our first attempt at implementing dsort pass 2 using FG. Disjoint pipelines proved useful for the load-balancing phase of pass 2. The merge stage, however, was overburdened with both reading and merging of sorted runs. As before, the dashed arrows represent interprocessor communication.

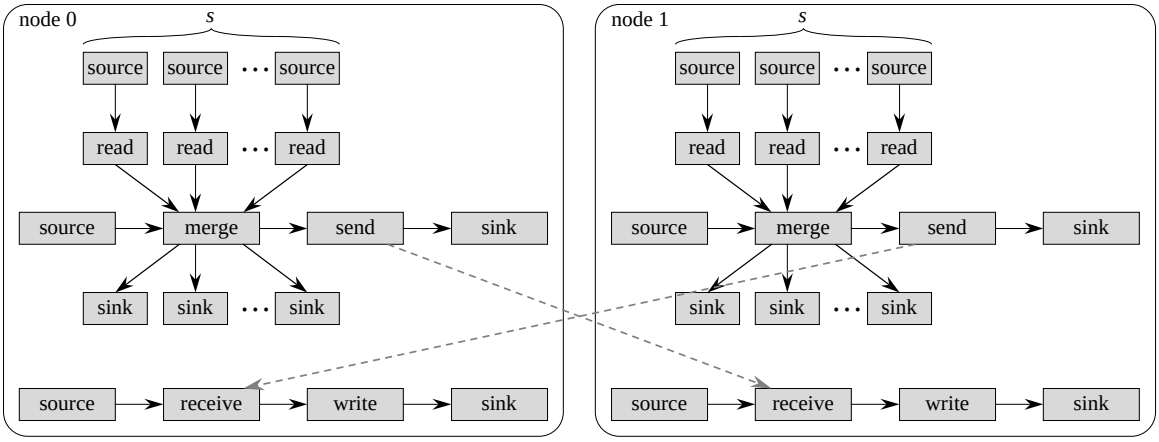


Figure 3.4: Our vision for pipeline structures for dsort pass 2 to disband the reading and merging of sorted runs. In this figure, there are as many vertical pipelines as the number s of sorted runs after pass 1. The merge stage is, therefore, common to $s+1$ pipelines. As we shall soon see, FG's intersecting pipelines made it possible for a stage to be part of more than one pipeline.

sorted runs, with each pipeline consisting of a read stage feeding into the merge stage. That is, assuming that there are s sorted runs, the merge stage would be common to $s+1$ pipelines, as shown in Figure 3.4. If FG could support such a design, the read stage in each of the "vertical pipelines" could take care of reading a block from the respective sorted run and then feeding the queue that sits between it and the merge stage. Let us name the vertical pipelines as p_1, p_2,..., p_s, starting from the left, and let us denote the pipeline consisting of the merge and send stages as p_{s+1}. The merge stage would then operate as follows: in the first round of p_{s+1}, the merge stage would accept a buffer containing a small portion of a sorted run from each of the vertical pipelines, and it would also accept a buffer from the source stage of p_{s+1} to store merged data. After the merge stage is finished with the last record in the buffer that it accepted from p_j (corresponding to sorted run j), the merge stage would proceed only after accepting another buffer from p_j. When

the buffer belonging to p_{s+1} is full, it should be conveyed to the send stage. Therefore, in any round of p_{s+1}, although the merge stage would always accept a buffer from p_{s+1}, it would accept buffers from vertical pipelines only when necessary. That is, with the help of these additional pipelines that intersect at the merge stage, we would be able to read ahead in each of the vertical pipelines using a synchronous function. Each of the $s + 1$ pipelines in Figure 3.4 could run at its own speed, similar to disjoint pipelines.

In order to be able to put the above strategy in practice, FG was extended to provide *intersecting pipelines* that can converge at a common stage, and diverge later if need be. A stage that is common to two or more pipelines is called an *intersecting stage*. As with disjoint pipelines, a user can create intersecting pipelines and pass them to FG's pipeline manager. We overloaded the `accept_thumbnail` method to take a pipeline number[9] as a parameter so that an intersecting stage can specify the pipeline from which to accept a buffer.

In dsort pass 2, each node runs the pipelines shown in Figure 3.4. For efficiency, the send pipeline, also referred to as p_{s+1} earlier, uses much larger buffers than the vertical pipelines. All vertical pipelines use the same buffer size, and as in pass 1, the receive pipeline has the same buffer size as the send pipeline. Because we knew that each sorted run is as big as the buffer size that was used in pass 1, we were able to set the number of rounds for each vertical pipeline. Similarly, each node can set the number of rounds of the send pipeline based on its partition size after pass 1; the receive stage sets the caboose after receiving P messages tagged as CABOOSE_MSG.

When we tried to test intersecting pipelines with a few hundred sorted runs, the program came to an abrupt halt. It turns out that the Linux kernel imposes a limit on the number of pthreads that can run concurrently. Although the documented limit is quite high ($\approx$1024), we found that, in practice, an FG program with more than about 100 pthreads does not run. Because FG maps each stage, including the source and sink stages, to its own thread, a pipeline structure similar to that of dsort pass 2 with hundreds or thousands of sorted runs will reach the system-imposed limit on pthreads fairly quickly. For example, if dsort pass 1 results in 100 sorted runs for a particular dataset, FG would spawn more than 200 pthreads in pass 2. Although we expect most FG applications to have only a few stages and, therefore, not to be hindered by the pthread limit, we had at least one application, dsort, which compelled us to think of a workaround for the number of threads that we

[9] Pipelines are numbered, starting from 0, in the order in which they are passed to the pipeline manager.

spawn.

We noticed that in Figure 3.4, all vertical pipelines have the same structure; also, the read stages all have the same stage function: accept a buffer, read a block from the associated sorted run, and convey the buffer. Therefore, we pondered about whether it would be possible to "squeeze" all read stages into a single stage and, therefore, spawn only one thread for all of them. There were at least two immediate questions that needed to be answered with the single-thread approach. First, how would the read stage know which sorted run to read a block from? Second, would there be multiple buffer queues feeding into and going out from the read stage? Because we introduced pipeline numbers in FG to overload the `accept_thumbnail` method, we decided that we could add the same field to the definition of a thumbnail. That is, a thumbnail would now also carry information about the number of the pipeline to which it belongs. Hence, the read stage could work with a single incoming queue, because all the information that it requires would be carried by the pipeline thumbnail. The merge stage, however, would need a separate incoming queue for each sorted run, as before, so that it can accept the next block from the sorted run that it has just exhausted. That is, the read stage should have multiple queues going out from it.

When stages in multiple pipelines share the same stage function, they can be designated as *virtual* in FG. Each virtual stage runs in its own single thread. A pipeline that has a virtual stage is called a *virtual pipeline*. For example, the read stage in each of the vertical pipelines above could be designated virtual, thereby making each vertical pipeline a virtual pipeline. Therefore, FG will spawn only a single thread for all the read stages, as shown in Figure 3.5. In order to create multiple virtual pipelines in FG, the user first creates all the pipelines as usual; she then collapses the stages that share the same stage function into a single stage using the `make_virtual` call. That is, in order to create the vertical pipelines above, the user would first create the s vertical pipelines in the usual manner. Then, she would use the `make_virtual` call to notify FG to collapse all the read stages into a single stage. FG will, therefore, spawn only one thread for all the read stages. To further reduce the number of threads that are spawned, FG spawns only a single thread each for the sources and sinks of all virtual pipelines; that is, FG makes the sources and sinks of such pipelines virtual. Furthermore, only one buffer queue sits between the single sink and the single source of virtual pipelines.

FG's intersecting pipelines and virtual pipelines enabled us to implement dsort pass 2 cleanly

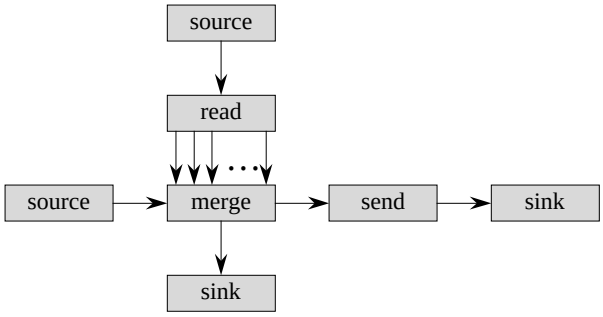


Figure 3.5: Virtual pipelines used to implement dsort pass 2; the receive pipeline for dsort pass 2 is not shown. Each box represents a thread and each arrow is associated with a buffer queue. All the read stages in the vertical pipelines share a common thread, as do all the source and sink stages. Although we have not shown the arrows from the sink to the source stages, each source stage has only one incoming queue.

and efficiently. A pipeline number field was added to the thumbnail's definition in FG; this field provided useful information to the read stage, and it also enabled FG to not have to overload the `convey_thumbnail` call. That is, the read stage makes the usual `convey_thumbnail` call, but under the hood, FG gleans the pipeline number information from the thumbnail and enqueues it in the correct outgoing queue.

3.3 Experimental results

We compared dsort with the FG-based implementation of an out-of-core, oblivious, sorting program based on columnsort ("csort") [38]. We found that dsort runs significantly faster than csort even though the I/O and communication patterns for a given input are predetermined in csort, whereas the patterns of these high-latency operations are determined at run time in dsort. Our results are in stark contrast to those obtained in a similar experiment by Chaudhry and Cormen [11] where the authors compared implementations of dsort and csort that did not use FG.

We also implemented shared-memory out-of-core sorting using FG and compared the results of our implementation with that of an STXXL-based implementation. The STXXL library [20, 22] provides classes for data structures such as vectors, stacks, and queues, and for algorithms such as sorting, including when data resides on parallel disks on a single node. The library also supports pipelining of operations to reduce I/O [6, 22].

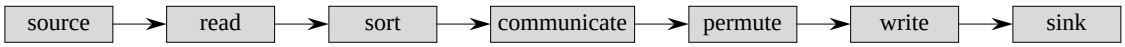


Figure 3.6: The stages involved in a each pass of a four-pass implementation of columnsort. The sort stage accomplishes an odd-numbered step in each pass and the communicate and permute stages together complete the even-numbered step corresponding to the pass. As in dsort, each participating node runs a copy of the above pipeline.

3.3.1 Columnsort

We now briefly describe the columnsort algorithm, and our FG-based, three-pass implementation, csort. Columnsort sorts $N = rs$ records, where the input is interpreted as an $r \times s$ matrix. The number r of rows is restricted to be even, the number s of columns must divide r, and r must be at least $2s^2$. The matrix is sorted in column-major order when columnsort completes. Columnsort proceeds in eight steps, where steps 1, 3, 5, and 7 all sort each column individually. Each even-numbered step performs a fixed permutation on the matrix. Step 2 transposes the matrix and reshapes it back to have r rows and s columns; step 4 performs the inverse permutation of step 2. Step 6 moves the bottom half of each column to the top half of the next column, and the top half of each column into the bottom half of the same column. The the top half of the leftmost column is filled with $-\infty$ values and the bottom half of the new rightmost column is filled with ∞ values. That is, step 6 shifts down each column by $r/2$ positions; inversely, step 8 shifts up each column by $r/2$ positions.

A four-pass implementation of out-of-core columnsort combines each odd-numbered step and its consecutive even-numbered step into a single pass. That is pass 1 performs steps 1 and 2, pass 2 implements steps 3 and 4, and so on. In each pass, each node of the cluster runs a copy of the pipeline shown in Figure 3.6. In each pass, the read stage reads a column of the matrix into a buffer and the sort stage accomplishes the appropriate odd-numbered step. The communicate and permute stages accomplish the appropriate even-numbered step, and the write stage outputs a column to the disk.

Following the earlier implementation [11], in csort we combined steps 5–8 of columnsort into a single pass, to achieve a three-pass implementation. The key observation is that in the four-pass implementation, the communicate, permute, and write stages of pass 3, along with the read stage of pass 4, just shift each column down by $r/2$ positions. In csort, we eliminate one pass by replacing

these four stages by a single communicate stage. Csort has two important properties. First, csort is oblivious to the data values (except for the local sort steps within each node); therefore, its disk-I/O and communication patterns are predetermined. Second, because the communication steps in csort correspond to highly regular permutations such as transposing a matrix or sending half of each node's data to the next node,[10] each node receives exactly as much data as it sends in each communication step.

3.3.2 Observations

Having seen the details of distribution-based sorting and those of sorting based on columnsort, we can say that the distribution-based algorithm has one advantage and two disadvantages compared with the columnsort-based algorithm [43]:

- Both algorithms make multiple passes over the data, where each pass reads each record to be sorted once from one of the disks in the cluster and writes each record once to one of the cluster's disks. The distribution-based algorithm makes only two passes[11] to the columnsort-based algorithm's three, and so the columnsort-based algorithm performs approximately 50% more disk I/O.

- In each pass of the distribution-based algorithm, some nodes might read or write differing volumes of data, and therefore some nodes might read or write more than the average volume of data. In the columnsort-based algorithm, all nodes read and write exactly the same volume of data. Thus, the I/O time consumed by the most heavily used disk in a pass might be greater in the distribution-based algorithm than in the columnsort-based algorithm.

- The distribution-based algorithm's I/O and communication operations are determined only dynamically, as the algorithms execution unfolds, thereby making it difficult to prefetch data. The columnsort-based algorithm's I/O and communication operations are known in advance. Thus, the columnsort-based algorithm is more amenable to overlapping I/O, communication, and computation than is the distribution-based algorithm.

[10] Columns are distributed among nodes in round-robin order, so that columns numbered i, $i + P$, $i + 2P$, etc., reside on node i.

[11] Note that pass 0 is a slight misnomer because we do not read each record in the input when selecting splitters.

The question is whether the disadvantages of the distribution-based algorithm are enough to outweigh its lesser I/O volume.

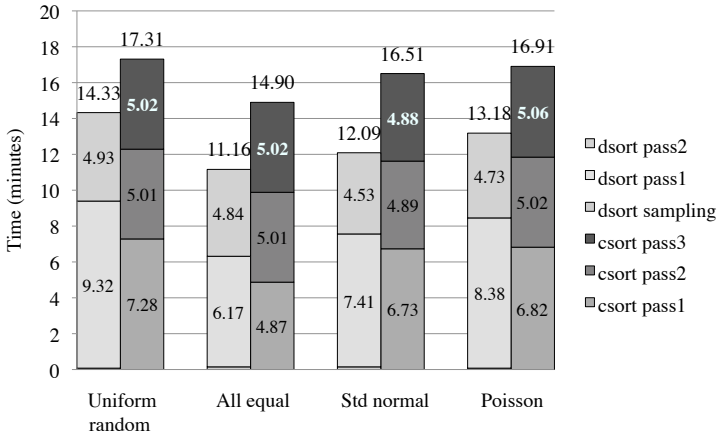
3.3.3 Experimental setup

We ran our experiments on a Beowulf cluster with 32 nodes, of which we used 16 nodes. Each node on the cluster has two 2.8-GHz Intel Xeon processors, 4 GB of RAM, and a 36-GB Ultra-320 SCSI hard drive. The nodes run RedHat Linux 9.0 and are connected by a 2-Gbps Myrinet network. In both programs, we use the C stdio interface for disk-I/O and a thread-safe implementation of MPI, ChaMPIon/Pro, for interprocessor communication. We did not use MPICH2 [42] because MPICH2 was not thread-safe at the time.

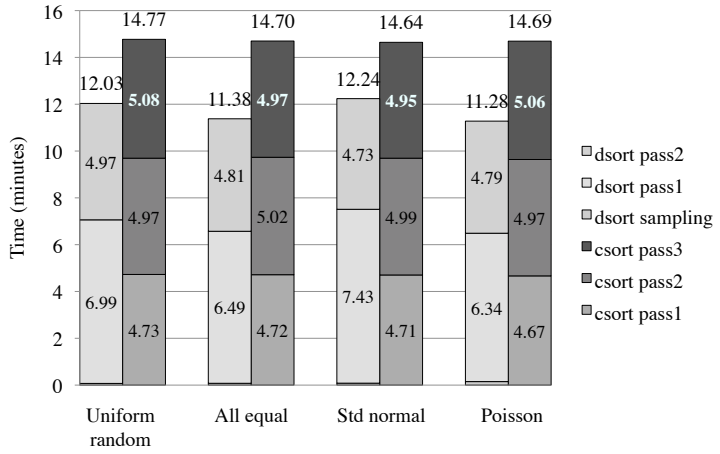
We compared dsort and csort with various key distributions: uniform random, all keys equal, standard normal, and Poisson with $\lambda = 1$. For the input in which all keys had the same value, we extended the keys as described in Section 3.2.1. In addition to these inputs, we ran the two programs on a distribution designed to bring out poor performance in dsort. In this distribution, we designed the keys so that communication is unbalanced in dsort pass 1. Each time communication occurs in pass 1, almost all the records go only to q out of the 16 nodes, with the remaining $16 - q$ receiving very few records. The nodes that comprise the set of q varies over time so that each node ultimately receives its full share of records. We designed input distributions for $q = 1, 2, 4, 8$, and 16. In each experiment, we sorted a total of 64 gigabytes of data, distributed evenly among the 16 nodes. In order to vary the volume of records that are sorted for the same amount of data, we used two different record sizes in our experiments: 16-byte records for a total of 4 gigarecords, and 64-byte records for a total of 1 gigarecord.

3.3.4 Results

Figure 3.7 shows the results for uniform random (averaged over three distinct input datasets), all keys equal, standard normal, and Poisson distributions. In each case, dsort beat csort, taking from 74.26% to 85.06% of csort's time. The figure also illustrates that dsort's advantage of having one fewer pass outweighs its disadvantages of having unbalanced I/O and communication patterns. As seen from the results in Figure 3.8, dsort performed fairly well even when it was run on in-



(a) 16-byte records



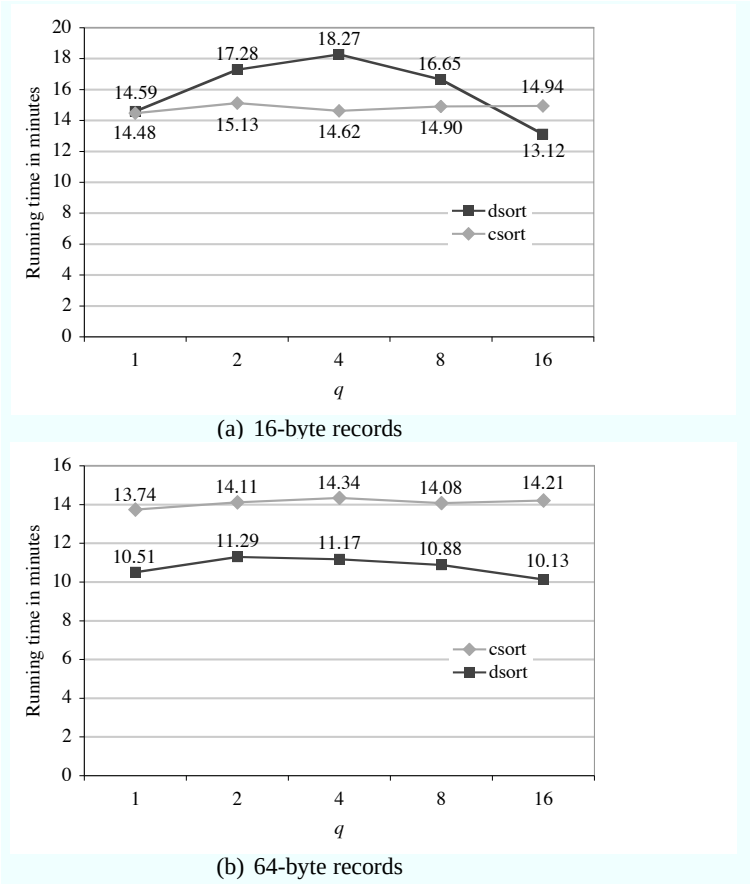
(b) 64-byte records

Figure 3.7: Running time of dsort and csort on various input distributions of 64 GB of data on 16 nodes.

puts devised to slow it down. For 64-byte records, dsort comfortably beat csort for all values of q. For 16-byte records, however, csort outperformed dsort for $q = 1, 2, 4$, and 8 and was slower only for $q = 16$. As we mentioned earlier, our results differ greatly from the non-FG implementations of dsort and csort that were compared in [11]. In this previous work, for 64-byte records, dsort was slower than csort for $q = 1, 2$, and 4 and was faster for $q = 8$ and 16.

The results of our experiments, although welcome, were surprising to us because we expected csort's obliviousness to key values to work in its favor. On the other hand, dsort's performance depends heavily on the disk-I/O and communication patterns generated by the input keys. Having one fewer pass than csort, however, seemed to prevail for dsort.

We believe that the extensions to FG made a significant contribution to dsort's performance.



(a) 16-byte records

(b) 64-byte records

Figure 3.8: Running times of dsort and csort for 64-GB datasets designed as bad inputs to dsort. In each communication of pass 1, almost all the records go to only q out of the 16 nodes.

FG's disjoint pipelines allowed nodes to send and receive data at different rates in pass 1, and multiple intersecting pipelines support stages that consume data from one or more pipelines and emit the results into a different pipeline at varying rates. Thus, using FG was a win-win situation for us: we got better performance and each stage in dsort was fairly simple to program.

Unfortunately, we were restricted to input file sizes of only 4 GB per node due to available disk space in our cluster. Although a dataset of size 64 GB might seem small, given many modern cluster configurations, we believe that our results would also extend to larger inputs. In other words, the dataset size is a limitation of neither dsort nor FG.

3.3.5 Shared-memory out-of-core sorting

We now compare our implementation of out-of-core sorting on a single machine using FG against the external-sorting routine provided by STXXL [6, 22].

Our shared-memory implementation of out-of-core sorting using FG involves two passes; on a single machine, we do not need to select splitters. In pass 1, we implement a simple pipeline with three user-defined stages—read, sort, and write—to create multiple sorted runs on disk. Pass 2 merges these sorted runs to produce the final sorted output using intersecting pipelines, similar to that of Figure 3.4, on a single node, where we replace the send stage with a write stage. On a single node, we do not require the receive pipeline shown in the figure. We use the `sort` and `multiway_merge` routines provided by the parallel library from `libstdc++` [40] to parallelize the sort and merge stages in passes 1 and 2, respectively.

The number of sorted runs could be a few thousand even for moderate input sizes such as 64 GB. In pass 2, although we collapse all the read stages to a single stage using FG's virtual pipelines,[12] FG still maintains a separate buffer queue for each sorted run between the read and merge stages shown in Figure 3.5. In addition, each buffer queue requires its own semaphore. That is, despite the savings offered by FG's virtual pipelines, we might fall short of system resources to run the program. Furthermore, using a single read stage for thousands of sorted runs would be detrimental to the program's performance.

In our implementation, we use hierarchical merging to thwart such problems. When the number of sorted runs after pass 1 exceeds a certain limit, say k, we resort to hierarchical merging, wherein we merge k consecutive sorted runs at a time, thereby creating longer sorted runs, and we continue this process until the number of sorted runs is at most k. Once the number of sorted runs is at most k, we merge the sorted runs one last time for the final sorted output. In general, if there are r sorted runs, we require $\lceil \log_k r \rceil - 1$ passes to bring down the number of sorted runs to at most k. In each pass, we merge all the sorted runs available from the previous pass in batches of k consecutive sorted runs at a time. To merge any k consecutive sorted runs, we use the same set of pipelines as in pass 2, i.e., k vertical pipelines containing read and merge stages and one horizontal pipeline with merge and write stages. In general, if there were r sorted runs before hierarchical merging, there will be $\max(k, \lceil r/k^{\lceil \log_k r \rceil - 1} \rceil)$ sorted runs after we finish all required passes of hierarchical merging.

Before we set up and run the pipelines for pass 2, we first check whether the number of sorted

[12] Recall that FG automatically collapses all the source stages and all the sink stages of virtual pipelines to a single source stage and a single sink stage, respectively.

runs, say r, output by pass 1 exceeds k. If $r > k$, we finish executing all passes of hierarchical merging to bring down the number of sorted runs to at most k, and then we merge these sorted runs in pass 2. In our implementation, we set $k = 60$.

If the input is of size b bytes, and if we have to resort to hierarchical merging, then in each hierarchical merging pass, we read and write b bytes of data. Because each hierarchical merging pass entails extra I/O, we engineered our implementation of pass 1 to create longer in-memory sorted runs, thus writing longer sorted runs to disk in an attempt to restrict the number of sorted runs to at most k. We do not provide details here, but we refer the reader to Section 4.4.2 for an example.

We ran our experiments on a machine that has one quad-core, 2.8-GHz Intel i7 processor, 8 GB of RAM, and a 1.5-TB, 7200 rpm, SATA 3-Gb/s hard drive. The machine runs Fedora linux, release 13. We use the `read` and `write` system calls for unbuffered disk I/O. We compiled our FG-based implementation, written in C++, using the g++ 4.4.4 compiler at optimization level O3. We implemented a program to use STXXL's pipelined `sort` routine for out-of-core sorting and compiled it using g++ 4.4.4 (optimization level O3) and STXXL version 1.3.0 with parallel pipelines enabled, compiled in parallel mode.

We compared the two implementations on input sizes ranging from 8 GB to 128 GB, where we doubled the input size successively. In all our experiments, we used 16-byte records with 8-byte random keys. We restricted all experiments to use 1 GB of RAM. We used a block size of 4 MB in the STXXL code. In the FG implementation, we used a buffer size of 64 MB in pass 1. In pass 2, all vertical pipelines used a buffer size of 4 MB and the horizontal pipeline ran with a buffer size of 128 MB. Each pipeline was given as many buffers as the number of user-defined stages in it. This configuration allowed us to contain our memory usage to within 1 GB of RAM.

As Figure 3.9 shows, the FG-based implementation of shared-memory out-of-core sorting outperformed the STXXL-based implementation for all input sizes. The STXXL-based implementation was approximately 10.6%–19.5% slower than the FG-based implementation. Each result represents the average of three runs, where running times varied only slightly within each group of three.

Because we present our results in seconds, normalized by the number of gigabytes in the input, we would expect to see an almost straight-line graph for both implementations. On the contrary, as Figure 3.9 illustrates, we see jumps in running times for both implementations starting at 64 GB

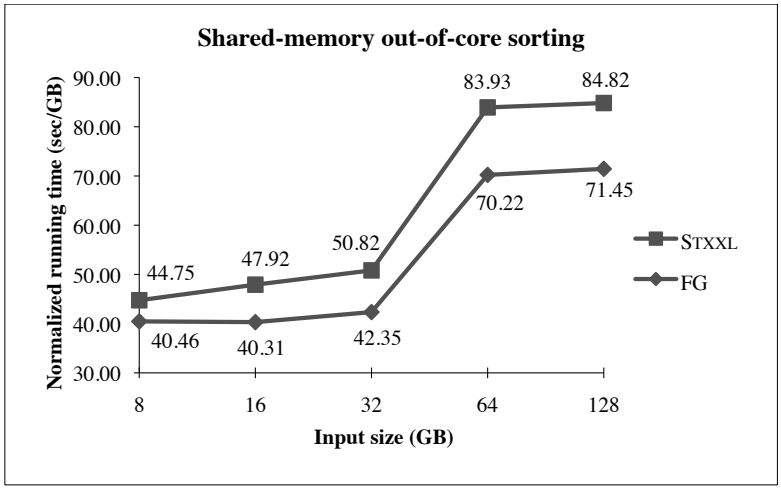


Figure 3.9: Performance results for the FG-based and STXXL-based implementations of out-of-core sorting on a single machine. All results are presented in seconds, normalized per gigabyte of input.

of input. Why do we see this jump in running times? By instrumenting our code, we found that although we never require more than one merging pass, we do resort to hierarchical merging when processing input sizes of 64 GB and more, which we believe explains the increase in running time. The I/O-volume generated by the STXXL-based code suggests that STXXL, too, follows a hierarchical merging approach in its sorting implementation. For an input of size b bytes that requires h passes of hierarchical merging, both implementations read a total of $(2 + h)b$ bytes and write the same amount. The number of hierarchical merging passes, h, could be different for the two implementations, however.

Therefore, an FG-based implementation of out-of-core sorting performs well both in the shared-memory and distributed-memory settings.

Chapter 4

External-Memory Suffix Arrays in FG

4.1 Introduction

The suffix array [26, 41], which is the lexicographically sorted array of the suffixes of a string, finds applications in string matching, bioinformatics, and text compression [10, 29]. For example, to find all occurrences of a pattern P in a text T, we could perform a binary search for P in the suffix array of T, because all suffixes that have P as a prefix will occur together in the suffix array of T. The Burrows-Wheeler transform [10] requires sorting the cyclic permutations of a string; if we append a character that is lexicographically smaller than all the other characters to the end of the string, then the problem of sorting cyclic permutations is equivalent to that of sorting suffixes.

The suffix array is usually represented as an array of starting indices of the lexicographically increasing suffixes. Let us denote the range of integers $i, i+1, \ldots, j$ by $[i, j]$ and let $[i, j) = [i, j-1]$. We assume that we are interested in constructing the suffix array of an n-character string $T = T[0, n) = t_0 t_1 \ldots t_{n-1}$ over some alphabet. For $i \in [0, n)$, let $S_i = t_i t_{i+1} \ldots t_{n-1}$ denote the ith suffix of T. Given any two suffixes S_i and S_j, where $i, j \in [0, n)$, we say that $S_i < S_j$ if S_i appears before S_j in lexicographic order. Then, the suffix array $SA[0, n)$ of T is a permutation of $[0, n)$ such that $S_{SA[0]} < S_{SA[1]} < \cdots < S_{SA[n-1]}$.

4.1.1 Background

The suffix tree [29] of a string is the compact trie of all its suffixes. The suffix tree of a string of length n over alphabet Σ can be constructed in $O(n \log |\Sigma|)$ time and $O(n)$ space, or using $O(n|\Sigma|)$

time and space. Although suffix trees are useful in a wide range of applications, their space-versus-time tradeoff hinders their usability when the alphabet size is large. In these situations, the suffix array presents itself as a more practical data structure. For a long time, however, the only known linear-time algorithm for constructing the suffix array of a string T was the lexicographic depth-first traversal of the suffix tree of T [29].

In 2003, Kärkkäinen and Sanders [33], Kim et al. [35], and Ko and Aluru [37] came up with three different in-memory, linear-time algorithms to construct suffix arrays. The DC3 algorithm [33, 34] is a simple algorithm for constructing suffix arrays, and it adapts to many models of computation. In the external-memory model [56, 57], the DC3 algorithm requires $O((n/(BD))\log_{M/B}(n/B))$ parallel I/Os and $O(n\log_{M/B}(n/B))$ internal work, where n is the number of characters in text T and D, B, and M are the number of disks, block size, and internal memory size of a machine, respectively. We describe the DC3 algorithm in Section 4.2.

Dementiev et al. [21] have implemented the DC3 algorithm in external memory using STXXL [20], which is based on the C++ Standard Template Library [50]. In their paper, the authors present the implementation of external-memory suffix arrays using techniques such as doubling [2] and doubling with discarding [17], in addition to the DC3 algorithm. By comparing the performance of the algorithms on various inputs, the authors conclude that the DC3 algorithm outperforms the other techniques, both in terms of running time and the I/O volume.

The STXXL library provides classes for data structures such as vectors, stacks, and queues, and for algorithms such as sorting, including when data resides on parallel disks on a single node. The library also supports pipelining of operations to reduce I/O [6, 22], which is used extensively by Dementiev et al. [21] in their implementation of external-memory suffix arrays.

This chapter describes how we implemented the DC3 algorithm for suffix arrays in external memory using the FG programming environment. As we shall see, the DC3 algorithm can be programmed efficiently using FG's pipeline structures. Experiments on real-world inputs and artificially constructed inputs demonstrate that FG as a software platform is well-suited for implementing external-memory suffix arrays. We found that the FG-based implementation of the external-memory DC3 algorithm ran comparably to the STXXL-based implementation for all types and sizes of inputs that we tested against.

4.2 The DC3 algorithm

In this section, we describe the in-memory, linear-time DC3 algorithm and provide pseudocode for its external-memory implementation.

The DC3 algorithm assumes that the input is a string T of length n over the alphabet $[1, n]$. The algorithm requires a few extra characters beyond the end of the string, for which it introduces a character, 0, which is lexically smaller than all characters in T. That is, the algorithm assumes that the input is a sequence of n integers in the range 1 to n, and that $t_j = 0$ for $j \geq n$.

The integer-alphabet requirement of the algorithm is not as restrictive as it seems. Given a string $T' = t'_0 t'_1 \ldots t'_{n-1}$ of length n over any alphabet, we can sort the unique characters of T', assign a rank to each character, and construct string T over the alphabet $[1, n]$ such that $t_i = \text{rank}(t'_i)$, for all $i \in [0, n)$. Because the character ranks are order preserving, the order of suffixes in T and T' is the same.

Throughout this chapter, we will compare strings and character-tuples lexicographically. We will use the % symbol for the modulus operator, and the / symbol will denote integer division. That is, for any i, we have $i \,\%\, 3 = i \mod 3$ and $i \,/\, 3 = \lfloor i / 3 \rfloor$.

We will expose the details of the DC3 algorithm with the help of the example string[1]

	0	1	2	3	4	5	6	7	8
$T[0,n) =$	m	a	l	a	y	a	l	a	m

with suffix array $SA = \langle 5, 1, 7, 3, 6, 2, 8, 0, 4 \rangle$.

The set of positions $\{i \in [0, n) : i \,\%\, 3 \neq 0\}$ is called the set of *sample positions*, and the suffixes starting at these positions are called *sample suffixes*. The remaining suffixes, starting at positions congruent to 0 mod 3, are *nonsample suffixes*. The DC3 algorithm proceeds in three steps:

1. **Sort sample suffixes.** Construct strings R_1 and R_2 such that $R_k = \langle [t_i t_{i+1} t_{i+2}] : i \,\%\, 3 = k \rangle$, for $i \in [0, n)$ and $k = 1, 2$. That is, for $k = 1, 2$, the characters of R_k are triples $[t_i t_{i+1} t_{i+2}]$ starting at positions i such that $i \,\%\, 3 = k$, for $i \in [0, n)$. Let $R = R_1 \odot R_2$ be the concatenation of R_1 and R_2. For our example string, we have

[1] Malayalam is a language spoken in southern India.

index in T	1	4	7	2	5	8
character in R	[ala]	[yal]	[am0]	[lay]	[ala]	[m00]
index in R	0	1	2	3	4	5

.

We define a *nonempty suffix* of R at position i in R to be all characters from i up to the first character that contains a 0 or the end of the string. In our example R above, the nonempty suffixes starting at indices 0, 2, and 4 are `alayalam0`, `am0`, and `alam00`, respectively. When $n \,\%\, 3 = 1$, we include a character [000] at position n in R_1. Therefore, we are always guaranteed to find a character containing a 0 in R_1. Note that, not counting the ending 0s, the nonempty suffixes of R correspond to the set of sample suffixes of T. In Lemma 1 below, we prove that the order of suffixes of R is the same as the order of its nonempty suffixes. Therefore, the order of suffixes of R gives us the order of the nonempty suffixes of R, which in turn gives us the order of the sample suffixes of T.

In order to sort the suffixes of R, the algorithm first sorts and ranks the unique characters of R, where ranks start from 1. In our example, R has five unique characters, which in sorted order are [ala], [am0], [lay], [m00], and [yal]. The algorithm then constructs a new string R' by renaming each character in R with its rank as a character in R'. If all characters of R' are different, we directly get the order of sample suffixes. Otherwise, the algorithm determines the suffix array of R' recursively. In our example,

index in T	1	4	7	2	5	8
character in R'	1	5	2	3	1	4
index in R'	0	1	2	3	4	5

,

and we recurse. The suffix array of R' is $SA_{R'} = \langle 4, 0, 2, 3, 5, 1 \rangle$; therefore, the list of sorted sample positions of T is $\langle 5, 1, 7, 2, 8, 4 \rangle$. Now that the order of sample suffixes is known, the algorithm assigns a rank, starting from 1, to each suffix S_i, where $i \in [0, n+2]$, and it denotes the rank of S_i by $rank(S_i)$. The ranks of suffixes at positions $[n, n+2]$ are set to 0, and $rank(S_i)$, such that $i \,\%\, 3 = 0$, is currently undefined and is indicated by $\perp$. For $T =$ `malayalam`, with $n = 9$, the ranks of suffixes at positions $[0, 11]$ are shown below:

i	0	1	2	3	4	5	6	7	8	9	10	11
character in T	m	a	l	a	y	a	l	a	m	0	0	0
$rank(S_i)$	$\perp$	2	4	$\perp$	6	1	$\perp$	3	5	0	0	0

.

2. **Sort nonsample suffixes.** Using the result of the previous step, we can sort the nonsample suffixes by comparing tuples $(t_i, rank(S_{i+1}))$. In our example, $S_3 < S_6 < S_0$ because $(a, 6) < (l, 3) < (m, 2)$.

3. **Merge the two sorted sets of suffixes.** For $k = 0, 1, 2$, let us define sets $P_k = \{i \in [0, n) : i \,\%\, 3 = k\}$. The algorithm defines the following comparison function for merging the sorted sets of sample and nonsample suffixes:

$$\begin{aligned}
i \in P_0,\ j \in P_1 \quad &: \quad S_i < S_j \Leftrightarrow (t_i, rank(S_{i+1})) < (t_j, rank(S_{j+1})), \\
i \in P_0,\ k \in P_2 \quad &: \quad S_i < S_k \Leftrightarrow (t_i, t_{i+1}, rank(S_{i+2})) < (t_k, t_{k+1}, rank(S_{k+2})), \\
j,\ k \in P_1 \cup P_2 \quad &: \quad S_j < S_k \Leftrightarrow rank(S_j) < rank(S_k)\ .
\end{aligned}$$

   For example, $S_1 < S_3$ because $(a, 4) < (a, 6)$; $S_6 < S_2$ because $(l, a, 5) < (l, a, 6)$, and $S_5 < S_1$ because $1 < 2$. For the string $T =$ `malayalam`, the sets of starting indices of the sorted suffixes at positions 0, 1, and 2 mod 3 are $\{3, 6, 0\}$, $\{1, 7, 4\}$, and $\{5, 2, 8\}$, respectively. Upon merging these sorted sets using the comparison function that was just described, we get $SA = \langle 5, 1, 7, 3, 6, 2, 8, 0, 4\rangle$.

Here is the lemma that allows us to consider the suffixes of R instead of the nonempty suffixes of R when determining the order of the sample suffixes of T.

Lemma 1 *For the string $R = R_1 \odot R_2$, where $R_k = \langle [t_i t_{i+1} t_{i+2}] : i \in [0, n), i \,\%\, 3 = k\rangle$ for $k = 1, 2$, the order of suffixes of R is the same as the order of nonempty suffixes of R.*

Proof. We will use lower-case Greek letters to represent zero or more contiguous characters of R, and lower-case English letters to denote a single character of R. Recall that each character of R is a character triple of T.

If all characters in R are unique, then the lemma is obvious, because then the rank of a suffix (nonempty or otherwise) is determined by its first character.

We now handle the case when R has duplicate characters, i.e., there exist at least two suffixes in R with some common prefix. Let $\alpha b\beta$ and $\alpha c\gamma$ be two suffixes[2] of R with a common prefix α, where α has at least one character and $b \neq c$. We will prove the lemma using three cases based on whether the suffixes start in R_1 or R_2.

1. **Both suffixes start in R_1.** Let y denote the character in R_1 that contains a 0; recall that R_1 is guaranteed to have such a character. Then, we can rewrite the suffixes $\alpha b\beta$ and $\alpha c\gamma$ as $\alpha b\delta_1 yR_2$ and $\alpha c\delta_2 yR_2$, respectively.[3] Now, $\alpha b\beta < \alpha c\gamma$ if and only if $b < c$. Because R_2 is common to both suffixes, we are, in effect, comparing the respective nonempty suffixes. That is, if both suffixes start in R_1, the order of the suffixes of R matches the order of its nonempty suffixes.

2. **Both suffixes start in R_2.** If both suffixes $\alpha b\beta$ and $\alpha c\gamma$ start in R_2, then they automatically correspond to the respective nonempty suffixes. Hence, if both suffixes start in R_2, the order of suffixes of R gives the order of the nonempty suffixes.

3. **One suffix starts in R_1 and the other starts in R_2.** Without loss of generality, let us assume that $\alpha b\beta$ starts in R_1; therefore, $\alpha c\gamma$ starts in R_2. Let us rewrite $\alpha b\beta$ as $\alpha b\delta_1 y\rho\alpha c\gamma$, where y is the character in R_1 that contains a 0, and b and y might be the same. As we noted in case 2, because suffix $\alpha c\gamma$ starts in R_2, it is the same as its corresponding nonempty suffix. The nonempty suffix of $\alpha b\beta$ is $\alpha b\delta_1 y$. Because $\alpha b\beta < \alpha c\gamma$ if and only if $b < c$, and because b is part of the nonempty suffix of $\alpha b\beta$, we can say that the order of the suffixes $\alpha b\beta$ and $\alpha c\gamma$ is determined by the order of the corresponding nonempty suffixes. ■

Figure 4.1 gives the pseudocode from [21] for an external-memory implementation of the DC3 algorithm. Lines 1–8 carry out step 1 of the DC3 algorithm. Lines 1 and 2 create the set of (character triple, position) tuples starting at sample positions and sort the set lexicographically by the first component. That is, at the end of line 2, all tuples with the same character-triple occur together in set S. The NAME procedure, for which we do not provide pseudocode, scans set S, assigns a fresh name, starting from 1, to each new triple that is found, and builds set R of (name, position) tuples

[2] Any two suffixes of a string must differ in at least one character.

[3] In the new representation, it is possible for b or c to equal y. If $b = y$, then rewrite $\alpha b\beta$ as αbR_2, and if $c = y$, then rewrite $\alpha c\gamma$ as αcR_2.

DC3(T)

// For all sample positions, store (character triple, position) tuples in set S

1 $S = \{(T[i, i+2], i) : i \,\%\, 3 \neq 0\}$

2 sort S by component 1

3 $R = \text{NAME}(S)$

4 **if** the names in R are not unique

// Use $i \,\%\, 3$ as the primary key and $i \,/\, 3$ as the secondary key to sort the tuples in R

5 sort the $(name, i) \in R$ by $(i \,\%\, 3, i \,/\, 3)$

6 $R'[0, 2n/3) = \langle name : (name, i) \in R \rangle$ // build recursive input

7 $SA12 = \text{DC3}(R')$ // recurse

8 $R = \langle (j + 1, SA12[j]) : j \in [0, 2n/3) \rangle$

9 sort R by the second component

10 $S_0 = \{(t_i, t_{i+1}, r', r'', i) : i \,\%\, 3 = 0, (r', i+1), (r'', i+2) \in R\}$

11 $S_1 = \{(r, t_i, r', i) : i \,\%\, 3 = 1, (r, i), (r', i+1) \in R\}$

12 $S_2 = \{(r, t_i, t_{i+1}, r'', i) : i \,\%\, 3 = 2, (r, i), (r'', i+2) \in R\}$

13 sort S_0 by components 1, 3 // sort nonsample suffixes by $(t_i, rank(S_{i+1}))$

14 sort S_1 and S_2 by component 1 // sort sample suffixes by rank

15 $SA = \text{MERGE}(S_0, S_1, S_2)$ using comparison function

$(v, \ldots) \in S_1 \cup S_2 \leq (w, \ldots) \in S_1 \cup S_2 \;\Leftrightarrow\; v \leq w$

$(c, c', u', u'', i) \in S_0 \leq (f, f', x', x'', j) \in S_0 \;\Leftrightarrow\; (c, u') \leq (f, x')$

$(c, c', u', u'', i) \in S_0 \leq (v, d, v', j) \in S_1 \;\Leftrightarrow\; (c, u') \leq (d, v')$

$(c, c', u', u'', i) \in S_0 \leq (w, e, e', w'', k) \in S_2 \;\Leftrightarrow\; (c, c', u'') \leq (e, e', w'')$

16 **return** $\langle$last component of $s : s \in SA\rangle$

Figure 4.1: Pseudocode by Dementiev et al. [21] for the external-memory implementation of the DC3 algorithm.

by replacing the character triple in each tuple of set S by its name. In lines 5–8, the pseudocode checks whether the order of sample suffixes has been completely determined, and then it recurses if necessary.

Lines 9–16 implement steps 2 and 3 of the algorithm. Sets S_0, S_1, and S_2 that are created in lines 10–12 contain enough information to allow the MERGE procedure in line 15 to correctly merge the sorted sets of sample and nonsample suffixes. Line 16 returns the suffix array of the input as the list of positions with lexicographically increasing suffixes.

The paper by Dementiev et al. [21] shows that the algorithm in Figure 4.1 can be implemented to run using 3(sort($10n$) + scan($2n$)) I/Os, where scan(x) and sort(x) are the number of I/Os required to scan and sort x words of data, respectively. The authors use words of size $\lceil \log n \rceil$ bits for inputs of size n. In the Parallel Disk Model [57], which assumes a machine with internal memory of size

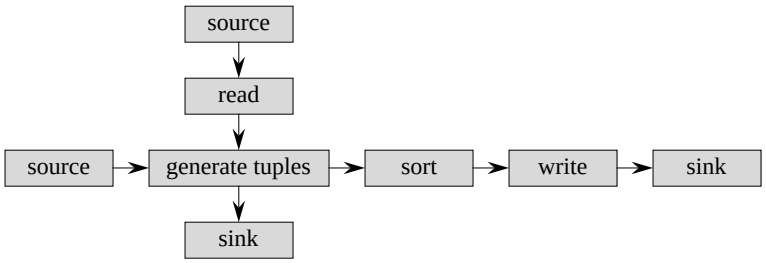


Figure 4.2: Intersecting pipelines in FG to generate sorted runs of (character triple, position) tuples starting at sample positions.

M and D disks, each of which can be accessed in blocks of size B, scan(x) = $\lceil x/(BD) \rceil$ and sort(x) = $(x/(BD)) \log_{M/B}(x/B)$.

4.3 Suffix-array implementation in FG

In this section, we present the design of the FG pipeline structures that we used to implement the external-memory DC3 algorithm. We will associate our pipelines with the line numbers of the pseudocode of Figure 4.1 that they implement. As we shall soon see, our implementation comprises a series of complex FG pipelines. We omit buffers and arrows connecting the respective sink and source stages in all the figures in this section.

4.3.1 Generate character triples at sample positions and create sorted runs

Lines 1 and 2 of the pseudocode generate (character triple, position) tuples starting at sample positions and sort them lexicographically by the first component, so that at the end of line 2, all duplicate character triples occur contiguously in set S. In our implementation, we generate sorted runs of (character triple, position) tuples starting at sample positions, sorted lexicographically by character triples, using the intersecting pipelines shown in Figure 4.2. When the pipelines of Figure 4.2 finish, we complete line 1 and part of line 2 of the pseudocode.

In the pipelines of Figure 4.2, the read stage accepts an empty buffer from the source stage on its pipeline, reads a buffer's worth of data from the input file, and conveys the buffer to the next stage, which generates tuples. Similar to the DC3 algorithm, we assume that the input is a sequence of n integers in the range $[1, n]$. We also assume that the size of the input file is n bytes; we interpret each

byte as an `unsigned char` in C++, and we use the character's numerical value in the machine's character set as its integer value. In each round except the last one, therefore, the read stage reads as many bytes from the input file as a buffer can hold. Because the input size n might not be an integral multiple of the buffer size, the buffer read into in the last round might be only partially full.

The first stage along the horizontal pipeline, which generates tuples, is common to two pipelines. This stage accepts a buffer with input characters from the read stage along the vertical pipeline and it accepts an empty buffer from the source stage along the horizontal pipeline. By using a static counter and the data in the vertical buffer, this stage fills the horizontal buffer with (character triple, position) tuples, starting at sample positions. After filling a horizontal buffer with as many tuples as the buffer can hold, this stage conveys the buffer to the next stage along the horizontal pipeline and accepts a fresh buffer from the source stage. When this stage has consumed all the data available in a vertical buffer, it conveys the vertical buffer to the sink stage along the vertical pipeline and accepts another buffer with input characters from the read stage. The last two stages, named sort and write, along the horizontal pipeline, sort a buffer of (character triple, position) tuples by the first component and write a sorted buffer to disk, respectively. Therefore, when the pipelines shown in Figure 4.2 finish, we have sorted runs of (character triple, position) tuples, starting at sample positions, available on disk.

In Figure 4.2, why did we require the stage that generates tuples to be an intersecting stage? Could we have flattened the pipelines of Figure 4.2 into a single, linear pipeline with the source, read, generate tuples, sort, write, and sink stages? No, because the stage that generates tuples consumes its input at a rate different from the rate at which it produces output. These rates differ because the sizes of the input and output data types of the stage differ—the stage consumes input characters and outputs (character triple, position) tuples. For example, an x-byte buffer can hold up to x input characters, but it can hold at most $x/7$ (character triple, position) tuples, assuming 4-byte position values. Hence, the stage cannot reuse the buffer that it accepts from the read stage to emit (character triple, position) tuples, which it would have to in case of a single, linear pipeline.

4.3.2 Merge sorted runs of character triples and name them

Before we proceed to name the character triples as required by line 3 of the pseudocode, we must merge the sorted runs of (character triple, position) tuples that we created earlier. We merge sorted

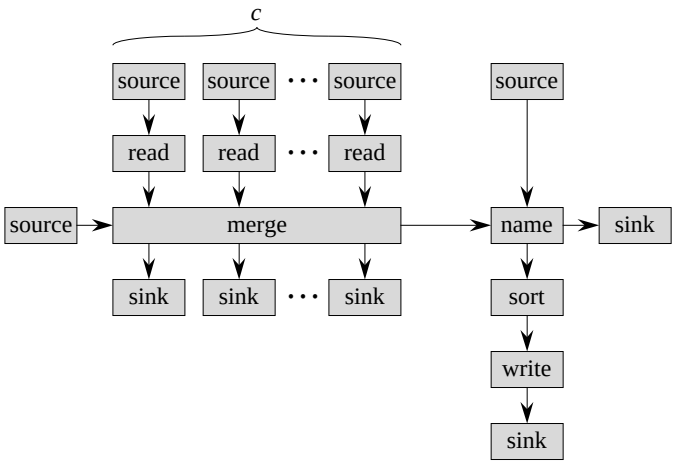


Figure 4.3: FG pipelines to merge sorted runs of (character triple, position) tuples and to generate sorted runs of (name, position) tuples.

runs of (character triple, position) tuples and produce (name, position) tuples using the pipelines shown in Figure 4.3. Assuming that the horizontal pipeline of Figure 4.2 writes out c sorted runs of (character triple, position) tuples, Figure 4.3 shows c vertical pipelines with the read and merge stages, one horizontal pipeline with the merge and name stages, and one vertical pipeline with the name, sort, and write stages.

Along all the c vertical pipelines with the read and merge stages, the read stage on pipeline i reads a block from sorted run i. The merge stage, which is common to $c + 1$ pipelines, accepts a block from the read stage along each of the c vertical pipelines, and it produces merged output along the horizontal pipeline. Each buffer that the merge stage conveys to the name stage along its horizontal pipeline lexicographically follows the previous buffer that it conveyed.

The name stage, which is the other intersecting stage in the pipelines of Figure 4.3, is common to two pipelines. Along its horizontal pipeline, this stage accepts buffers containing (character triple, position) tuples in globally sorted order. The name stage assigns a fresh name,[4] starting from 1, to each unique character triple and outputs (name, position) tuples[5] to an empty buffer, which it accepts along its vertical pipeline. Again, we chose to implement the name stage as an intersecting stage because the rate at which it consumes buffers along its horizontal pipeline differs from the rate at which it produces buffers along its vertical pipeline.

[4]The name of a character triple corresponds to its rank.

[5]The position value in each tuple the same as the second component of the corresponding (character triple, position) tuple.

For now, let us assume that the input does not require recursion; that is, we assume that all tuples in set R, which is created using the name stage, have unique names. Therefore, the condition in line 4 of the pseudocode will be false, which means that we will proceed from line 9 after the pipelines of Figure 4.3 finish. Based on this assumption, we sort the tuples in the buffers along the last vertical pipeline of Figure 4.3 by their second component, i.e., by position. Hence, after the pipelines of Figure 4.3 finish, we have sorted runs of (name, position) tuples available on disk, which when merged will produce the sorted set R per line 9 of the pseudocode.[6]

4.3.3 Sort nonsample suffixes

We continue with the assumption that the input does not require recursion, which means that we skip over lines 5–8 and proceed from line 9. As we proved in Section 4.2, because set R has unique names, it represents the suffix array of the sample suffixes.

The next step in the algorithm is to generate sets S_0, S_1, and S_2 and sort them (lines 10–14). In our implementation, however, we do not generate three distinct sets. Instead, we note that an element from any of the sets S_0, S_1, or S_2 always contains the position i in addition to at most two characters and two ranks. Therefore, we generate a single set whose elements are a 5-tuple: (t, t', r, r', i); we distinguish these identical-looking elements as being from sets S_0, S_1, or S_2 based on position i. For each $i \in [0, n)$, set S_k, where $k \in \{0, 1, 2\}$, contains tuples with positions $i \,\%\, 3 = k$. We will denote the single set containing these 5-tuples as set S_{012}.

Of course, we interpret the fields of the 5-tuple, (t, t', r, r', i), differently for the elements of sets S_0, S_1, and S_2. For example, for an element of set S_0 in set S_{012}, fields t and t' represent the characters at positions i and $i + 1$, fields r and r' represent the ranks of suffixes starting at positions $i + 1$ and $i + 2$ (which have just been uniquely determined), and $i \in [0, n)$ represents a position such that $i \,\%\, 3 = 0$. For an element of set S_1 in set S_{012}, field t represents the character at position i, fields r and r' represent the ranks of suffixes at positions i and $i + 1$, and $i \in [0, n)$ represents a position such that $i \,\%\, 3 = 1$; field t' is unused. Similarly, for an element of set S_2 in set S_{012}, fields t and t' represent the characters at positions i and $i + 1$, fields r and r' represent the ranks of suffixes starting at positions i and $i + 2$, and $i \in [0, n)$ represents a position such

[6] In a later subsection, where we deal with inputs that require recursion, we will see that, in fact, we do things slightly differently in our actual implementation.

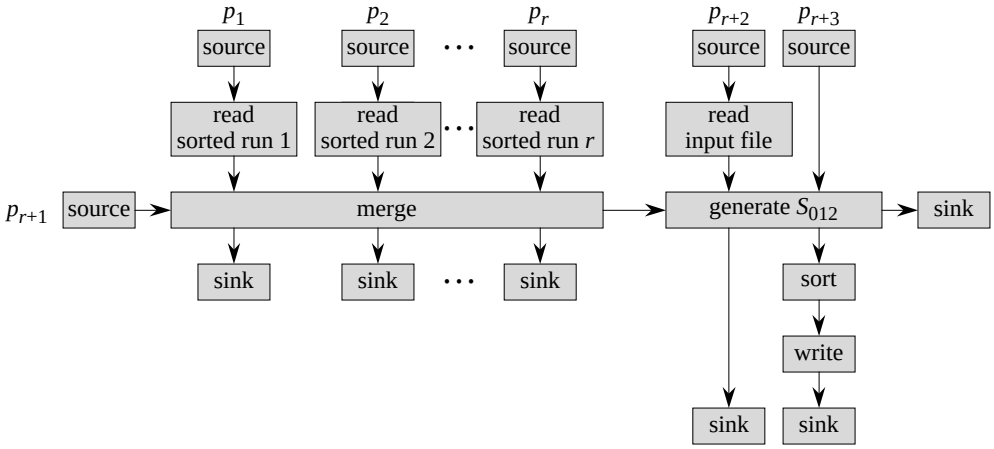


Figure 4.4: FG pipelines to merge sorted runs of (name, position) tuples and to generate sorted runs of set S_{012}. Each element of set S_{012} is a 5-tuple, (t, t', r, r', i). We distinguish an element as being originally from set S_0, S_1, or S_2, based on position i, and we sort the elements of set S_{012} using the comparison function given in line 15 of the pseudocode.

that $i \,\%\, 3 = 2$.

After the pipelines of Figure 4.3 finish, suppose we have r sorted runs of (name, position) tuples belonging to set R, which are sorted by position, available on disk. In order to create set S_{012}, we merge the r sorted runs to obtain sample suffixes sorted by position while also reading the input file, as Figure 4.4 illustrates. The pipelines in Figure 4.4 also create sorted runs of set S_{012}, thus inherently completing lines 13 and 14 of the pseudocode. For convenience, we have numbered the $r + 3$ pipelines in Figure 4.4 and we will refer to them as p_1 to p_{r+3}.

In Figure 4.4, the merge stage, which is common to $r+1$ pipelines, merges sorted runs of (name, position) tuples, which are fed to it along vertical pipelines p_1 to p_r, and it produces merged output along horizontal pipeline p_{r+1}. The contents of the buffers conveyed from the merge stage along pipeline p_{r+1}, when taken in sequence, form the sorted set R as in line 9 of the pseudocode.

The second stage in pipeline p_{r+1}, which generates set S_{012}, is common to three pipelines: p_{r+1}, p_{r+2}, and p_{r+3}. This stage accepts buffers with data along pipelines p_{r+1} and p_{r+2}, and it accepts empty buffers along pipeline p_{r+3}. The buffer that this stage accepts along the horizontal pipeline p_{r+1} contains (name, position) tuples globally sorted by position, and along the vertical pipeline p_{r+2}, the stage accepts a buffer's worth of input characters, which are read from the input file by the read stage. Using these two buffers, this stage generates tuples belonging to set S_{012} and copies these tuples to the empty buffer that it accepts along the vertical

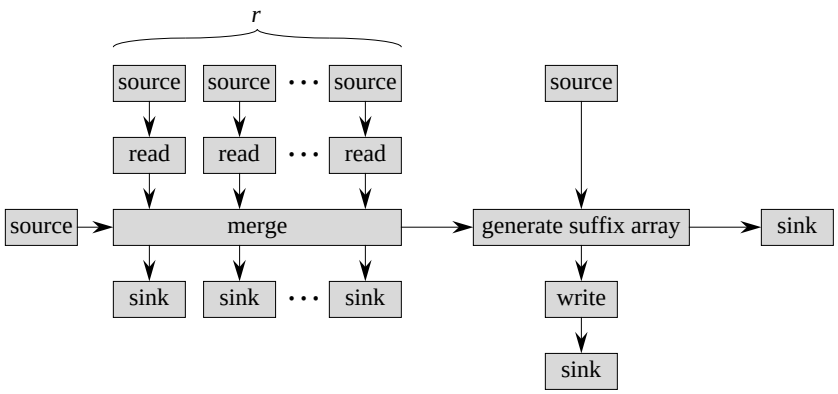


Figure 4.5: FG pipelines to generate the final suffix array.

pipeline p_{r+3}.

The sort stage in pipeline p_{r+3} sorts the tuples in its incoming buffer using the comparison function given in line 15 of the pseudocode in Figure 4.1. The write stage in the last pipeline of Figure 4.4 writes a sorted buffer of set S_{012} to disk.

4.3.4 Merge sorted sets of sample and nonsample suffixes

The sorted runs of set S_{012} must be merged, using the comparison function in line 15 of Figure 4.1, to create the suffix array of T. Figure 4.5 contains $r + 2$ pipelines, where r is the number of sorted runs of set S_{012}. The r vertical pipelines in Figure 4.5 containing the read and merge stages and the horizontal pipeline with the merge and generate suffix array stages work together to read and merge the sorted runs of set S_{012} and produce the merged output along the horizontal pipeline. In Figure 4.5, the stage that generates the suffix array of T consumes the merged output from the horizontal pipeline and it produces the suffix array of T along the last vertical pipeline. The write stage in the last vertical pipeline of Figure 4.5 writes the final indices representing the suffix array of T to disk.

4.3.5 Recursion

So far, we have assumed that the input does not require recursion. We now handle the case when the input requires recursion. Let us return to the pipelines of Figure 4.3, which merge sorted runs of

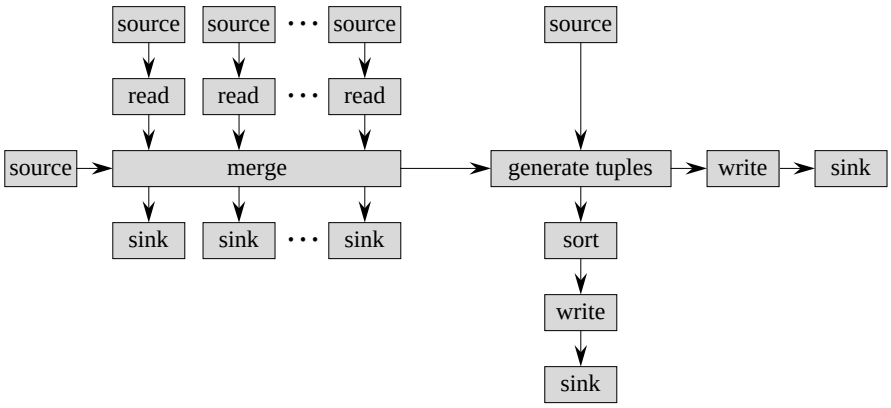


Figure 4.6: FG pipelines to generate sorted runs of (character triple, position) tuples in a recursive phase.

(character triple, position) tuples and create sorted runs of (name, position) tuples.[7] Earlier, we said that the sort stage in the last vertical pipeline of Figure 4.3 sorts the tuples in each incoming buffer by their second component. For the sake of simplicity, we mispresented the sorting criterion at the time. Indeed, our implementation always assumes that the condition in line 4 will be true, i.e., that another level of recursion will be required, and so we sort buffers along the last vertical pipeline of Figure 4.3 by (position % 3, position / 3). We decided to use this sorting criterion because we expect most inputs to require many levels of recursion. Hence, after the pipelines of Figure 4.3 finish, we have sorted runs of (name, position) tuples available on disk, which when merged will produce the sorted set R per line 5 of the pseudocode. The name component of the merged result of these sorted runs gives us recursive input, which can be used for the recursive call in line 7 of the pseudocode.

One of the benefits of pipelined programs is that we can avoid unnecessary I/O. As Figure 4.6 illustrates, in a recursive phase, we feed the merged output directly to the stage that generates (character triple, position) tuples. In Figure 4.6, the vertical pipeline with generate tuples, sort, and write stages is similar in structure and functionality to the horizontal pipeline in Figure 4.2. Although pipelined programming helps us avoid unnecessary I/Os, you might wonder why we output the recursive input to disk.[8] We store the recursive input because it is also required later while creating set S_{012}.

We finish sorting set S of (character triple, position) tuples and name them using the pipelines

[7] Recall that these tuples belong to set R of line 3 of the pseudocode.
[8] Notice the write stage in the horizontal pipeline in Figure 4.6.

in Figure 4.3. We reexecute the pipelines in Figures 4.6 and 4.3 until the recursion bottoms out. In the last recursive phase, however, our default sorting criterion of (position % 3, position / 3) for the last vertical pipeline of Figure 4.3 is incorrect, because we will execute line 9 after checking the condition in line 4, which will be false. Note that line 9 requires the tuples in set R to be sorted by their second component. Therefore, we recreate sorted runs of (name, position) tuples, consistent with line 9 of the pseudocode, by running a simple linear pipeline[9] with read, sort, and write stages. This pipeline reads the sorted runs of (name, position) tuples, sorts them by their second component, and writes these freshly sorted runs to disk. When this linear pipeline completes, we have sorted runs of (name, position) tuples, sorted by their second component, which when merged will create set R per line 9 of the pseudocode.

When the recursion bottoms out, and after we return from a recursive phase, we run the pipelines in Figure 4.4 to create sorted runs of set S_{012}. To generate the suffix array in a recursive phase, we use the pipelines in Figure 4.5, but we insert a sort stage before the write stage in the last vertical pipeline. This sort stage sorts each incoming buffer of (name, position) tuples by the second component. Therefore, in a recursive phase, the write stage outputs sorted runs of (name, position) tuples to disk; these sorted runs will be merged by the pipelines in Figure 4.4 acting on behalf of the previous recursive step. In a recursive phase, the stage function for the generate suffix array stage outputs (name, position) tuples, with names starting from 1; the stage function converts each position in the range $[0, 2n/3)$ to its corresponding 1 mod 3 or 2 mod 3 position of the previous recursive step. When the pipelines in Figure 4.5 are used to generate the suffix array of the original input, the stage function for the generate suffix array stage outputs only the positions of the sorted suffixes.

4.4 Implementation details

In this section, we outline some implementation decisions that we made to engineer the performance of our external-memory DC3 program.

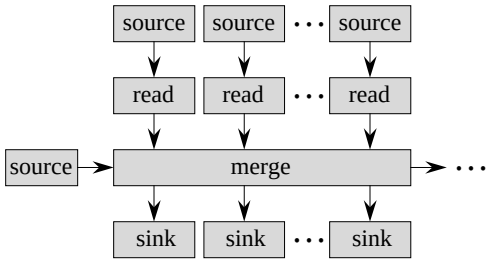


Figure 4.7: FG pipelines to read and merge sorted runs, followed by further processing.

4.4.1 Hierarchical merging

Until now, we have seen that the implementation of the external-memory DC3 algorithm frequently requires merging sorted runs available on disk. In order to merge sorted runs stored on disk, we use the pipeline structures shown in Figure 4.7, where the rightmost ellipsis in the figure could stand either for one or more stages following the merge stage in the horizontal pipeline or for additional pipelines, or both (see Figures 4.3–4.6).

Often, the number of sorted runs could be tens of thousands even for a moderate input size of half a million characters. Although FG's virtual pipelines spawn only a single thread for all the read, source, and sink stages, FG still maintains a separate buffer queue for each sorted run between the read and merge stages shown in Figure 4.7. In addition, each buffer queue requires its own semaphore. That is, despite the savings offered by FG's virtual pipelines, we fall short of system resources so that the program grinds to a halt.

In our implementation, we use hierarchical merging to thwart such problems. When the number of sorted runs exceeds a certain limit, say k, we resort to hierarchical merging, wherein we merge k consecutive sorted runs at a time, thereby creating longer sorted runs, and we continue this process until the number of sorted runs is at most k. Once the number of sorted runs is at most k, we merge sorted runs and use the sorted output in the manner shown in Figures 4.3–4.6. In general, if there are r sorted runs, we require $\lceil \log_k r \rceil - 1$ passes to bring down the number of sorted runs to at most k. In each pass, we merge all the sorted runs available from the previous pass in batches of k consecutive sorted runs at a time. To merge any k consecutive sorted runs, we use the pipelines shown in Figure 4.7, where just a write stage follows the merge stage in the horizontal

[9] We do not show the figure for this pipeline.

pipeline, thereby creating a longer sorted run of size ks, assuming that each sorted run was of size s before merging. In general, if there were r sorted runs before hierarchical merging, there will be $\max(k, \lceil r/k^{\lceil \log_k r \rceil - 1} \rceil)$ sorted runs after we finish all required passes of hierarchical merging.

Before we set up and run any of the pipelines in Figures 4.3–4.6, we first check whether the number of sorted runs, say r, output by the previous set of pipelines exceeds k. If $r > k$, we finish executing all passes of hierarchical merging to bring down the number of sorted runs to at most k, and then we set up and run the current set of pipelines. In our implementation of the external-memory DC3 algorithm, we set $k = 40$. In the interest of simplicity, we omitted these details while describing our implementation in Section 4.3.

4.4.2 Create longer, in-memory sorted runs

If we have to resort to hierarchical merging, and if the r sorted runs reside in a file of size b bytes, then in each hierarchical merging pass, we read and write b bytes of data. Because each hierarchical merging pass entails extra I/O, we engineered our implementation to create longer in-memory sorted runs, thus writing longer sorted runs to disk in an attempt to restrict the number of sorted runs to at most k. The next paragraph details our approach.

For all pipeline sets shown in Figures 4.2–4.6, we can calculate the number of sorted runs, say r, that will be written to disk before we begin executing the pipelines. Therefore, in our implementation, whenever we notice that $r > k$, we modify the pipeline structure such that longer sorted runs will get written to disk. Figure 4.8 shows how we modify the pipelines of Figure 4.2 to write longer sorted runs to disk. In Figure 4.8, for each round along the last vertical pipeline, the merge stage accepts m buffers along the horizontal pipeline and merges the contents of these buffers into an empty buffer that it accepts along the vertical pipeline. That is, if the horizontal pipeline has buffer size s, the last vertical pipeline has buffer size ms, which gets written to disk. Therefore, if the original pipeline structures of Figure 4.2 would have produced r sorted runs, the pipelines of Figure 4.8 would write only $\lceil r/m \rceil$ sorted runs to disk. We can make similar modifications to the pipelines in Figures 4.3–4.6 to write longer sorted runs to disk. In our implementation, we set $m = 10$. We have noticed that this approach often helps us avoid hierarchical merging before executing the next set of pipelines.

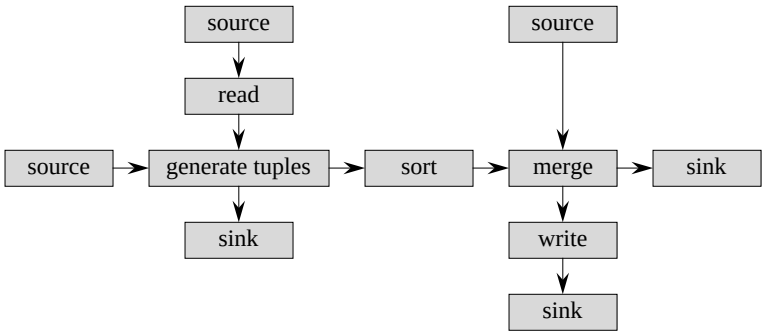


Figure 4.8: Modified FG pipelines from Figure 4.2 to create longer, in-memory sorted runs, thus writing longer sorted runs to disk. Similar modifications can be made to the pipelines of Figures 4.3–4.6.

4.4.3 Sort and merge stages

We parallelize the sort and merge stages in all pipelines, except for the sort stage in Figure 4.4.

Note that all sort stages we have seen in this chapter involve sorting tuples. In the sort stages in Figures 4.2, 4.3, 4.6, and in the sort stage that gets sandwiched between the generate suffix array and write stages in a recursive phase in Figure 4.5, all tuples share the same sorting criterion, and the rank of each tuple depends on a fixed set of fields of the tuple. Therefore, we use parallel radix sort, using 16-bit keys at a time, as the sort stage function for these sort stages. For example, let us focus on the sort stage in Figure 4.6, which sorts a buffer of (character triple, position) tuples. In our implementation, each character in the triple is a 32-bit value of type `unsigned int`, thus requiring six passes of radix sort through the buffer. Assuming that a buffer holds s (character triple, position) tuples, (a_i, b_i, c_i) for $i = 1, 2, 3 \ldots, s$, we sort the 16 least-significant bits of all c_i in one pass followed by the 16 most-significant bits of all c_i in another pass. Then, we sort the 16 least-significant bits and the 16 most-significant bits of all b_i in two passes and finally, we sort the 16 least-significant bits and the 16 most-significant bits of all a_i in the last two passes.[10] We use OpenMP's `parallel for` construct to parallelize each pass.

In the sort stage in Figure 4.4, however, any two tuples being compared could be from different sets (S_0, S_1, or S_2); that is, the inter-tuple rank depends on which combination of sets—of the six possible combinations—the tuples belong to. Similarly, for any two tuples belonging to the same

[10] In our code, whenever we use radix sort, we do not optimize for the most significant bits that remain zero and thus do not contribute to the result of the sort. We admit that this strategy is less efficient both in time and space, but considering that our input sizes range from 16 MB to 4 GB, i.e., from 2^{24} to 2^{32} bytes, we are unlikely to benefit much from the optimization for the input sizes that matter—512 MB and beyond.

set, the fields of the tuple that participate in the sorting criterion are different for each of the sets S_0, S_1, and S_2. Therefore, we perform simple, comparison-based sorting of set S_{012}, which subsumes the sets S_0, S_1, and S_2.

In all pipelines, we use the `multiway_merge` routine provided by `libstdc++` to parallelize the merge stages. The `multiway_merge` routine works in memory, but we want to merge sorted runs available on disk, with only buffer-sized portions of each sorted run available in memory at a time. Thus, to produce correctly merged output, we must ensure that none of the buffers run empty during a single call to `multiway_merge`. Therefore, we create partitions in the buffers for all sorted runs such that we can merge the sum of the sizes of these partitions in a single call to the routine. Below, we give pseudocode for the portion of the merge stage that uses the `multiway_merge` routine.

MERGE_STAGE_FUNCTION

```
    ... // deleted code
 1  while not all sorted runs have been fully exhausted
 2      assuming r non-empty sorted runs, let m_1, m_2, ..., m_r denote the respective
            maxima of the elements available in the buffers for the sorted runs
 3  m' = min{m_1, m_2, ..., m_r}     // minimum of all local maxima
 4  let s' be a block whose maximum element is m'
 5  rank m' in all sorted runs except s' using std::upper_bound()
 6  calculate t as the sum of the number of elements that can be merged from all blocks
    before the current buffer of s' is fully consumed
 7  call multiway_merge to merge t elements
    // current buffer for sorted run s' should be consumed now;
    // current buffers of some other sorted runs might also have been consumed
 8  for each sorted run
 9      if all elements in the current buffer for the sorted run have been consumed
10          convey the buffer
11          if this sorted run has not been fully exhausted
12              accept another buffer along the respective vertical pipeline
    ... // more deleted code
```

We will explain the above pseudocode using an example. As Figure 4.9 shows, let us assume that we want to merge five sorted runs and that we have a buffer's worth of data available from each of the sorted runs. Line 3 above defines m' as the minimum of the maximum elements of all buffers, and let us suppose that $s' = 2$ in line 4. Figure 4.9(a) shows a possible scenario after we rank m' in the buffers from all sorted runs. For each sorted run, the shaded region of its respective buffer contains elements that equal at most m'. Then, we calculate t to be the sum of sizes of all partitions,

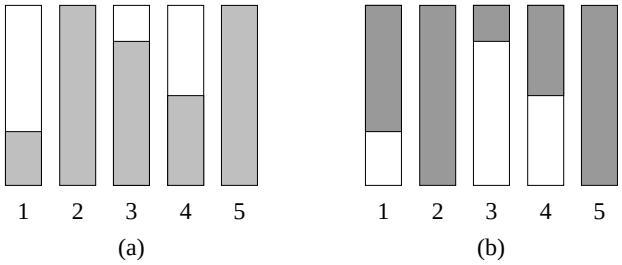


Figure 4.9: An example with five sorted runs to illustrate how we use the `multiway_merge` routine. Each vertical bar represents a buffer that is available in memory; run numbers are shown just below the buffers. We assume that in each buffer, the elements increase from the bottom of the buffer to the top. We calculate m' as in line 2 of the pseudocode. (a) The shaded region in each buffer represents the region of the buffer where elements equal at most m'. Because we assume that $s' = 2$, the buffer from sorted run 2 is fully shaded; the maximum element in the buffer from sorted run 5 also happens to be m' and hence its buffer is fully shaded, too. We calculate t to be the sum of the shaded regions in all buffers, and we call `multiway_merge` on t elements. After `multiway_merge` returns, we accept new buffers for sorted runs 2 and 5, whose buffers were fully consumed in the most recent call to the routine. (b) The shaded region within each buffer represents the portion of the buffer within which we should rank the new m' before the next call to `multiway_merge`.

i.e., the sum of sizes of all shaded regions in Figure 4.9(a). Line 6 of the above pseudocode calls `multiway_merge` to merge t elements. After the routine returns, we are sure to have consumed the buffer from sorted run s', which we assumed equals 2. In our example, we also happen to consume the buffer from sorted run 5, however. Therefore, line 12 above accepts the next buffer along the vertical pipelines for sorted runs 2 and 5, assuming that we haven't yet fully exhausted either of these sorted runs. Now, we are ready for another call to `multiway_merge`, so we can restart from line 1 above. Because the maximum elements of the buffers from sorted runs 2 and 5 have changed,[11] we reset m' and s' in lines 3 and 4, respectively. As Figure 4.9(b) shows, when we rank m' in all sorted runs in line 5, we should not use the portions of the buffers that we consumed in the previous call to `multiway_merge`. Then, we can continue as before from line 6.

4.5 Extensions to FG

In this section, we describe an extension to FG that came about from our original design of the external-memory DC3 algorithm. We extended FG to allow multiple sets of virtual pipelines to feed into an intersecting stage. As we shall soon see, such pipeline structures can prove useful in some scenarios, but FG did not support them earlier.

[11] We accepted a new buffer along the pipelines for these sorted runs. The buffers for sorted runs 1, 3, and 4 are unchanged, and so are the maximum elements in these buffers.

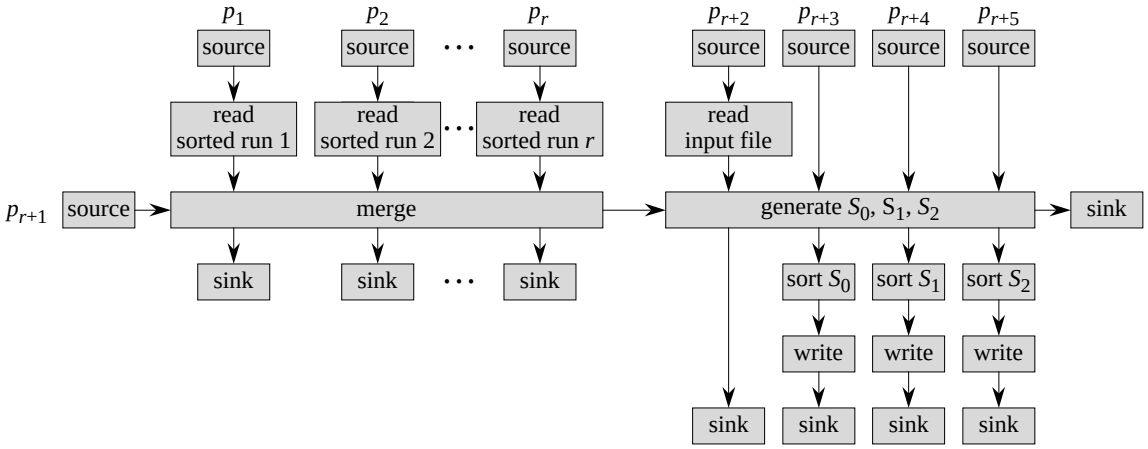


Figure 4.10: FG pipelines to merge sorted runs of (name, position) tuples and to generate sorted runs of sets S_0, S_1, and S_2.

We return the reader's attention to the pipelines of Figure 4.4, which generate sorted runs of set S_{012} (recall that each element of set S_{012} represents an element from one of the sets S_0, S_1, or S_2). Figure 4.10 shows our original design for generating sets S_0, S_1, and S_2. Earlier, we generated three distinct sets, using 5-tuples for elements of sets S_0 and S_2 and a 4-tuple for elements of set S_1 (see lines 10–12 of the pseudocode in Figure 4.1). Most of the description for the pipelines in Figure 4.10 is similar to that of Figure 4.4. Here, we briefly describe the working of the last three vertical pipelines of Figure 4.10. The stage that generated sets S_0, S_1, and S_2 accepted an empty buffer from each of the last three vertical pipelines and filled them with tuples of sets S_0, S_1, and S_2, respectively. The sort stages in pipelines p_{r+3}, p_{r+4}, and p_{r+5} sorted a buffer of set S_0, S_1, and S_2, respectively, and the write stages along these pipelines wrote out the sorted buffers to a distinct file. That is, earlier, we used a separate file to store the sorted runs of each of the three sets.

In order to merge the sorted runs of sets S_0, S_1, and S_2 to obtain the sorted sets of sample and nonsample suffixes and hence create the suffix array of T, we used the pipelines shown in Figure 4.11. Assuming that the pipelines of Figure 4.10 generated r_0, r_1, and r_2 sorted runs of sets S_0, S_1, and S_2, respectively, Figure 4.11 shows $r_0 + r_1 + r_2 + 4$ pipelines. Pipelines $p_{0,0}$ to p_{0,r_0} (pipeline numbers are shown in the figure) read and merge the sorted runs of set S_0 and produce the merged output along pipeline $p_{0,0}$. Similarly, the results of merging the sorted runs of sets S_1 and S_2 are output along pipelines $p_{1,0}$ and $p_{2,0}$, respectively. In Figure 4.11, the stage that generates the suffix array of T is a 4-way intersecting stage: it merges the incoming data along

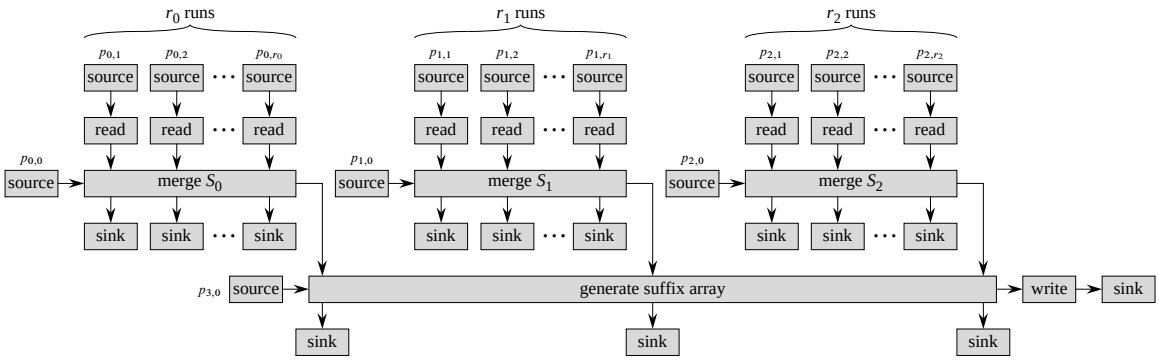


Figure 4.11: FG pipelines to generate the final suffix array in the original design. We assume that there are r_0, r_1, and r_2 sorted runs of sets S_0, S_1, and S_2, respectively. Starting from the left, we can say that there are three groups of vertical pipelines; in each such group, there is one horizontal pipeline with the merge stage and the stage that generates the suffix array, and some number of vertical pipelines with the read and merge stages. Therefore, we number pipelines using the group number and the pipeline number within the group. There are $r_0 + 1$ pipelines in the first group, with pipeline numbers $0,0$ to $0,r_0$, there are $r_1 + 1$ pipelines in the second group, with pipeline numbers $1,0$ to $1,r_1$, and there are $r_2 + 1$ pipelines in the third group, with pipeline numbers $2,0$ to $2,r_2$. For consistency, the horizontal pipeline with the write stage is assumed to have pipeline number $3,0$.

pipelines $p_{0,0}$, $p_{1,0}$, and $p_{2,0}$ using the comparison function shown in line 15 of the pseudocode, and it produces the suffix array of T along pipeline $p_{3,0}$. The write stage in pipeline $p_{3,0}$ of Figure 4.11 writes the final indices representing the suffix array of T to disk.

Figure 4.11 shows three sets of virtual pipelines, each of whose output feeds into an intersecting stage. We extended FG to allow such pipeline structures.

4.6 Experimental results

In this section, we compare the performance of our FG-based implementation of the external-memory DC3 algorithm with that of the STXXL-based implementation by Dementiev et al. [21].

We ran our experiments on a machine that has one quad-core, 2.8-GHz Intel i7 processor, 8 GB of RAM, and a 1.5-TB, 7200 rpm, SATA 3Gb/s hard drive. The machine runs Fedora linux, release 13. We use the `read` and `write` system calls for unbuffered disk I/O. We compiled our FG-based implementation, written in C++, using the g++ 4.4.4 compiler at optimization level O3. We contacted the STXXL implementors for the latest version of their suffix-array code and compiled it using g++ 4.4.4 (optimization level O3) and STXXL version 1.3.0 with parallel pipelines enabled, compiled in parallel mode.

We compared the two implementations using various types of inputs: an input comprising an $(n/2)$-length random string concatenated with itself (we refer to this input as Random2), an input with all characters equal, human genome data [24], and Gutenberg text [30]. We generated the first two types of inputs, and for the latter two types, we downloaded the input instances used in the STXXL implementation [21], which are available at http://algo2.iti.kit.edu/dementiev/esuffix/instances.shtml. For each type of input, we ran both implementations on input sizes ranging from 16 MB to 4 GB, doubling the input size successively. The maximum input size for Gutenberg and human genome inputs was approximately 3 GB, however. We restricted all experiments to use 1 GB of RAM. We used a block size of 4 MB in the STXXL code. In the FG implementation, all pipelines that had a read stage used a buffer size of 4 MB, and all other pipelines ran with a buffer size of 32 MB. Each pipeline was given as many buffers as the number of user-defined stages in it. This configuration allowed us to contain our memory usage within 1 GB of RAM.

As Figures 4.12(a)–(d) show, the FG implementation of the external-memory DC3 algorithm performed comparably to the STXXL implementation for all types of inputs and for each input size. As the figures show, the FG implementation is always faster for $n \leq 2^{31}$, and it is almost as good for the biggest input size, which is 4 GB in the random2 and all characters equal input types and 3 GB in case of Gutenberg and human genome inputs. The running times of the two implementations were approximately within 5%–37% of each other. Each result represents the average of three runs, where running times varied only slightly within each group of three.

Because we present our results in microseconds, normalized by the number of bytes in the input, we would expect to see an almost straight-line graph for both implementations. On the contrary, as Figures 4.12(a)–(d) illustrate, we see jumps in running times for all types of input, for both implementations. In the FG implementation, we see these jumps for input sizes 1 GB and beyond, whereas in the STXXL implementation, we see these jumps from 256 MB onwards. Why do we see this jump in running times? By instrumenting our code, we found that although we never require more than one merging pass, we do resort to hierarchical merging a number of times when processing input sizes of 1 GB and more, which we believe explains the increase in running time. The paper by Dementiev et al. [21] suggests that STXXL, too, follows a hierarchical merging approach in its sorting implementation; perhaps the STXXL-based implementation of the DC3 algorithm also requires many one-pass or multi-pass merges starting at 256 MB.

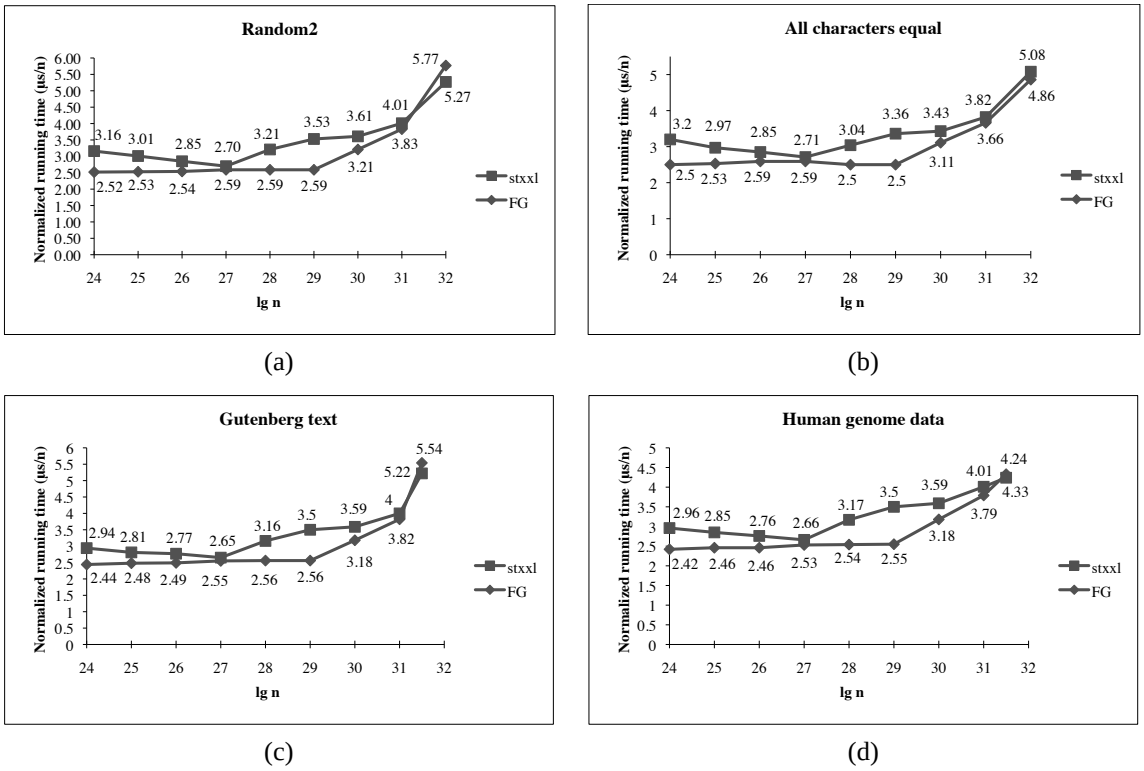


Figure 4.12: Performance results for the FG-based and STXXL-based implementations of the DC3 algorithm. All results are presented in microseconds, normalized per byte of input.

For an n-character input that requires d levels of recursion, our implementation reads a total of approximately $(38n + 96n(1-(2/3)^d)) \leq 134n$ bytes and writes the same amount plus another $2n$. The effect of hierarchical merging defies analysis, and therefore, these base expressions for the total volume of data read and written do not account for I/O induced by hierarchical merging. We measured the I/O volume due to hierarchical merging in our Random2 runs, however, finding that the additional volume ranged between 22.5% and 67.9% of the values given by the base expressions. The dominant computation time for each pipeline is $O(n \log B)$, where B is the buffer size.

4.7 Conclusions

In an earlier paper [43], where we used FG's intersecting pipelines for external-memory distribution sort, we speculated that we would be able to use intersecting pipelines in the design of external-memory algorithms other than sorting. Indeed, this project demonstrates that not only can we combine FG's intersecting pipelines in more innovative ways than we did for distribution sort, but

also that these pipelines are efficient in practice, as our experiments reveal. Most of the pipelines that we saw in this chapter are more complicated than those we implemented for out-of-core distribution sorting.

The DC3 algorithm presented us with interesting design and implementation opportunities. FG's design enabled us to reuse stage functions for stages such as read, sort, and write across different pipelines, with slight changes in function parameters. The FG-based implementation of the DC3 algorithm ran comparably to that of the STXXL-based implementation, which, too, is a well engineered implementation.

As we saw in section 4.5, our original design of external-memory suffix arrays also offered the idea of extending FG to allow multiple sets of virtual pipelines. Such pipeline structures can be useful when we wish to merge sorted runs of a limited number of different types of data, with each of their outputs feeding into other stages.

The pipelines in this project, which are implemented using FG version 1.4, also proffered ideas for some design features for FG's next version. We realized that auxiliary buffers might be redundant, after all. Whenever a stage requires auxiliary buffers, we could make it an intersecting stage with the source stage feeding it a pipeline buffer that the stage can use as an auxiliary buffer. In the current version of FG, auxiliary buffers in a pipeline are restricted to be of the same size as the pipeline buffers flowing through the pipeline. Our latest design idea for replacing auxiliary buffers will not bind their size based on other buffers.

Chapter 5

Fast Gauss Transform

5.1 Introduction

The discrete Gauss transform at a target point $t \in \mathbf{R}^d$ due to n source points $s_j \in \mathbf{R}^d$ for $j = 1, 2, \ldots, n$ is defined as

$$g(t) = \sum_{j=1}^{n} e^{-|t-s_j|^2/\delta} q(s_j) \ , \tag{5.1}$$

where $q(s_j)$ for $j = 1, 2, \ldots, n$ is a charge distribution function defined at the n source points, $|x - y|$ is the Euclidean distance between x and y, and $\delta > 0$ is the Gaussian parameter.

In many disciplines such as computational physics [39, 48, 55], computational finance [9], computer graphics, and machine learning [23, 36], we are interested in computing transformations of the form shown in equation (5.1) at m target locations t_i for $i = 1, 2, \ldots, m$. We can evaluate the sum of n Gaussians at m targets using a matrix-vector product

$$g = Gq \ ,$$

where we define the matrix G and the vectors q and g as

$$\begin{aligned} G_{ij} &= e^{-|t_i-s_j|^2/\delta} \text{ for } i = 1, 2, \ldots, m \text{ and } j = 1, 2, \ldots n \ , \\ q_j &= q(s_j) \text{ for } j = 1, 2, \ldots, n \ , \\ g_i &= g(t_i) \text{ for } i = 1, 2, \ldots, m \ . \end{aligned} \tag{5.2}$$

Given the sources, targets, charges at the sources, and the Gaussian parameter δ, constructing the matrix G of Gaussian interactions and computing the matrix-vector product Gq requires $O(mn)$ time, which becomes prohibitive for large m and n. The fast Gauss transform (FGT), introduced by Greengard and Strain [27], reduces the time complexity of evaluating equation (5.1) at m target locations to $O(m + n)$. The authors were able to improve the running time by approximating the Gaussian shown in equation (5.1) by using a truncated Hermite expansion, where the truncation point depends on the required precision. That is, the fast Gauss transform computes approximate values, albeit with high precision.

Later, Greengard and Sun [28] replaced the Hermite expansions in the fast Gauss transform with plane-wave expansions, without changing the asymptotic running time of the transform. In fact, in this new version of the fast Gauss transform, they were able to use a sweeping algorithm, which drastically reduces the running time of one of the computation steps.

In this chapter, we present our implementation of the plane-wave version of the fast Gauss transform, both in shared memory and in distributed memory. Because the distributed-memory implementation requires interprocessor communication, we use FG to help us overlap communication with computation. We compare the performance of our implementation against a distributed-memory implementation by Sampath, Sundar, and Veerapaneni [44]. Experimental results show that under certain assumptions that hold for both implementations, our implementation outperforms the other implementation. We begin by introducing the Hermite and plane-wave versions of the fast Gauss transform in Sections 5.2 and 5.3. Section 5.4 outlines some performance improvements for the FGT steps described by Spivak, Veerapaneni, and Greengard [47]. Section 5.5 discusses some obvious but useful observations that we made to improve our implementation, and Section 5.6 details our shared-memory and distributed-memory implementations. The final sections of this chapter present our experimental results and offer some concluding remarks.

5.2 Hermite version of FGT

Greengard and Strain [27] approximate the Gaussian field at a target t due to a source s, in one dimension, as a Hermite expansion centered at a point s_0:

$$g(t) = e^{-|(t-s)|^2/\delta} = \sum_{k=0}^{\infty} \frac{1}{k!} \left(\frac{s - s_0}{\sqrt{\delta}} \right)^k h_k \left(\frac{t - s_0}{\sqrt{\delta}} \right) , \tag{5.3}$$

where

$$\begin{aligned} h_k(x) &= e^{-x^2} H_k(x) , \\ H_k(x) &= 2x H_{k-1}(x) - 2(k-1) H_{k-2}(x) , \\ H_1(x) &= 2x , \\ H_0(x) &= 1 ; \end{aligned} \tag{5.4}$$

$H_k(x)$ and $h_k(x)$ are the Hermite polynomials and the associated Hermite functions, respectively.

For multidimensional FGT, we can split the d-dimensional Gaussian into the product of d one-dimensional Gaussians. If $t, s \in \mathbf{R}^d$, then

$$\begin{aligned} e^{-|t-s|^2/\delta} &= e^{(-(t_1-s_1)^2-(t_2-s_2)^2-\cdots-(t_d-s_d)^2)/\delta} \\ &= e^{-(t_1-s_1)^2/\delta} \cdot e^{-(t_2-s_2)^2/\delta} \cdots e^{-(t_d-s_d)^2/\delta} . \end{aligned} \tag{5.5}$$

The authors describe the higher-dimensional analog of equation (5.3) using multi-index notation. For any multi-index $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d)$ and any $t \in \mathbf{R}^d$, define

$$\begin{aligned} |\alpha| &= \alpha_1 + \alpha_2 + \cdots + \alpha_d , \\ \alpha! &= \alpha_1! \alpha_2! \cdots \alpha_d! , \\ t^\alpha &= t_1^{\alpha_1} t_2^{\alpha_2} \cdots t_d^{\alpha_d} . \end{aligned} \tag{5.6}$$

If p is an integer, we say that $\alpha \geq p$ if $\alpha_i \geq p$ for $1 \leq i \leq d$, and that $\alpha \leq p$ if $\alpha_i \leq p$

for $1 \leq i \leq d$. The multidimensional Hermite polynomials and Hermite functions are defined as

$$\begin{aligned} H_\alpha(x) &= H_{\alpha_1}(x_1)\, H_{\alpha_2}(x_2) \cdots H_{\alpha_d}(x_d)\,, \\ h_\alpha(x) &= e^{-|x|^2} H_\alpha(x) \\ &= e^{-(x_1^2+x_2^2+\cdots+x_d^2)} H_{\alpha_1}(x_1)\, H_{\alpha_2}(x_2) \cdots H_{\alpha_d}(x_d) \\ &= (e^{-x_1^2} H_{\alpha_1}(x_1))\,(e^{-x_2^2} H_{\alpha_2}(x_2)) \cdots (e^{-x_d^2} H_{\alpha_d}(x_d)) \\ &= h_{\alpha_1}(x_1)\, h_{\alpha_2}(x_2) \cdots h_{\alpha_d}(x_d)\,, \text{ by applying equation (5.4)}\,. \end{aligned} \tag{5.7}$$

Using equations (5.5), (5.6), and (5.7), Greengard and Strain show that

$$e^{-|t-s|^2/\delta} = \sum_{\alpha \geq 0} \frac{1}{\alpha!} \left(\frac{s - s_0}{\sqrt{\delta}} \right)^\alpha h_\alpha \left(\frac{t - s_0}{\sqrt{\delta}} \right)\,. \tag{5.8}$$

Greengard and Strain [27] prove that if N_B sources lie in a box B with center s_B and side length $\sqrt{\delta}$, and if t is a target point in a box C with center t_C, then the Gaussian in equation (5.1) due to the sources in box B can be evaluated using the Hermite expansion

$$g(t) = \sum_{\beta \geq 0} R_\beta \left(\frac{t - t_C}{\sqrt{\delta}} \right)^\beta\,,$$

where

$$\begin{aligned} R_\beta &= \frac{(-1)^{|\beta|}}{\beta!} \sum_{\alpha \geq 0} A_\alpha\, h_{\alpha+\beta} \left(\frac{s_B - t_C}{\sqrt{\delta}} \right)\,, \\ A_\alpha &= \frac{1}{\alpha!} \sum_{j=1}^{N_B} q_j \left(\frac{s_j - s_B}{\sqrt{\delta}} \right)^\alpha\,, \end{aligned}$$

with a suitable error bound for truncating the series after p^d terms. In the above equations, α and β are multi-indices. In later equations for the Hermite version, we will sum α and β up to p.

With this background, we are ready to look at a simple, fast algorithm for evaluating

$$g(t_i) = \sum_{j=1}^{n} e^{-|t_i - s_j|^2/\delta} q_j$$

at m target locations in $O(m+n)$ time, as given by Greengard and Strain [27]. The required precision ϵ determines the variable p, which denotes the number of expansion terms for the Hermite expansions. The authors assume that the sources s_j and targets t_i all lie in the unit box $[0,1]^d$. As a preprocessing step, they subdivide $[0,1]^d$ into smaller boxes of length $\sqrt{\delta}$ parallel to each axis and assign each source s_j and target t_i to its respective box. If N_B sources lie in box B, then $n = \sum_B N_B$. The FGT algorithm proceeds in three steps:

1. For each source box B, create a Hermite expansion with a term for each α_i such that $1 \leq \alpha_i \leq p$, totaling p^d terms:

$$\sum_{s_j \in B} q_j \frac{1}{\alpha!} \left(\frac{s_j - s_B}{\sqrt{\delta}} \right)^{\alpha} .$$

   For source box B, this step requires $O(p^d N_B)$ time and, therefore, over all the source boxes, this step requires $O(p^d n)$ time.

2. Now, consider a target box C. For each $t_i \in C$, we need to accumulate the total field from all source boxes. Because of the exponential decay of the Gaussian field, however, it suffices to include only the $(2b+1)^d$ nearest boxes, where b depends on the required precision ϵ. These nearest $(2b+1)^d$ boxes are defined as the *interaction region* for box C, denoted by $I(C)$. Therefore, in this step, each target box C accumulates the Hermite expansions from all source boxes within its interaction region:

$$\begin{aligned} R_\beta &= \frac{(-1)^{|\beta|}}{\beta!} \sum_{B \in I(C)} \sum_{\alpha \leq p} A_\alpha(B)\, h_{\alpha+\beta} \left(\frac{s_B - t_C}{\sqrt{\delta}} \right) , \qquad (5.9) \\ A_\alpha(B) &= \frac{1}{\alpha!} \sum_{s_j \in B} q_j \left(\frac{s_j - s_B}{\sqrt{\delta}} \right)^{\alpha} . \end{aligned}$$

   Because of the product form of $h_{\alpha+\beta}$ (see equation (5.7)), it turns out that computing the p^d coefficients R_β involves only $O(dp^{d+1})$ operations. For any target box, because we collect expansions from the $(2b+1)^d$ boxes in its interaction region, this step requires $O(b^d dp^{d+1})$ time in total for each target box, and $O(b^d dp^{d+1} N_{box})$ time over all boxes, where N_{box} is the total number of boxes.

3. Finally, for any target t_i in box C, approximate $g(t_i)$ by

$$\begin{aligned} g(t_i) &= \sum_{B} \sum_{s_j \in B} e^{-|t_i - s_j|^2/\delta} q_j \\ &= \sum_{\beta \le p} R_\beta \left(\frac{t_i - t_C}{\sqrt{\delta}} \right)^\beta + O(\epsilon) \,. \end{aligned} \tag{5.10}$$

Over all targets t_i, where $1 \le i \le m$, evaluating expression (5.10) requires $O(p^d m)$ time in total.

The above algorithm, therefore, requires $O(b^d dp^{d+1} N_{box}) + O(p^d n) + O(p^d m) = O(p^d (b^d dp N_{box} + n + m))$ time, where $N_{box} = (1/\sqrt{\delta})^d$ is the total number of boxes.

5.3 Plane-wave version of FGT

As mentioned earlier, Greengard and Sun [28] replaced the Hermite expansions in the fast Gauss transform with plane-wave expansions, which reduced the running time of box-to-box translations (step 2 in the algorithm from Section 5.2) without changing the overall running time of the algorithm. The authors also show how to decompose the computation as a product of matrices. In this section, we elaborate on the plane-wave version of the fast Gauss transform, and we introduce the matrix factorization representation of the fast Gauss transform. Our parallel implementation of the fast Gauss transform, which we will describe in a later section, is based on the plane-wave version of the FGT.

Greengard and Sun proved that if N_B sources lie in a box B with center s_B and side length $\sqrt{\delta}$, and if t is a target point in a box C with center t_C, then we can approximate the Gaussian in equation (5.1) using plane-wave expansions as

$$g(t) = \sum_{-p \le \beta \le p} D_\beta \, e^{i\lambda\beta \cdot (t - t_C)} + O(\epsilon) \,,$$

where

$$D_\beta = R_\beta \, e^{i\lambda\beta \cdot (t_C - s_B)} \,,$$

$$R_\beta \;=\; \left(\frac{L}{2p\sqrt{\pi}}\right)^d e^{-\lambda^2|\beta|^2\delta/4} \sum_{j=1}^{N_B} q_j e^{i\lambda\beta\,\cdot\,(s_B - s_j)} \,.$$

In the above equations, $i = \sqrt{-1}$ is the square root of negative unity, β is a multi-index, the number of expansion terms p and variable L depend on the required precision ϵ, we define $\lambda = L/(p\sqrt{\delta})$, and $\beta \cdot (t - t_C)$ in the first equation above represents the dot product of the d-element vectors β and $(t - t_C)$. Similar terms in the expressions for D_β and R_β above, and in the rest of this chapter, also stand for dot products.

Similar to the Hermite version, Greengard and Sun assume that the unit box $[0, 1]^d$ is subdivided into smaller boxes with length $\sqrt{\delta}$ along each axis. They describe the algorithm for the plane-wave version of FGT in three steps, replacing the Hermite expansions with suitable plane-wave expansions:

1. For each source box B, accumulate the influence of all sources in the box into a plane-wave expansion:

$$R_\beta(B) = \left(\frac{L}{2p\sqrt{\pi}}\right)^d e^{-\lambda^2|\beta|^2\delta/4} \sum_{j=1}^{N_B} q_j e^{i\lambda\beta\,\cdot\,(s_B - s_j)} \,, \text{ for } -p \le \beta \le p \,.$$

   Over all the source boxes, this step takes $O(p^d n)$ time.

2. As in the Hermite version, for each target box C, collect the plane-wave expansions from the nearest $(2b+1)^d$ boxes in C's interaction region:

$$D_\beta(C) = \sum_{B \in I(C)} R_\beta(B)\, e^{i\lambda\beta\,\cdot\,(t_C - s_B)} \,, \text{ for } -p \le \beta \le p \,.$$

   This step requires $O(b^d\, p^d)$ time for one box, and $O(b^d\, p^d\, N_{box})$ for all boxes.

3. Finally, for any target t in box C, approximate $g(t)$ using plane-wave expansions as

$$g(t) = \sum_{-p \le \beta \le p} D_\beta(C)\, e^{i\lambda\beta\,\cdot\,(t - t_C)} \,. \tag{5.11}$$

   This step requires $O(p^d m)$ time over all m targets.

Approximating a Gaussian using plane-wave expansions, therefore, takes $O(p^d n)+O(b^d p^d N_{box})+O(p^d m) = O(p^d(b^d N_{box}+n+m))$ operations. Note that the time required for box-to-box translations (step 2) is less than that for the Hermite version by a factor of dp.

In the literature [44, 47], step 1 of the algorithm is nicknamed the S2W step because for each source box B, this step accumulates the effect of all sources in the box into a single plane-wave expansion. Step 2 of the algorithm is called the W2L step because for each target box C, this step adds the plane-wave expansions from all source boxes lying in C's interaction region to form a "local" expansion for C. Similarly, the last step of the algorithm is labeled the L2T step because it uses the local expansion for each target box to generate the final answer for each target.

Greengard and Sun [28] also show how to represent the matrix G of Gaussian interactions (see equation (5.2)) in a factored form. Suppose there are S source boxes, $B_1, B_2, \ldots, B_S$, and T target boxes, $C_1, C_2, \ldots, C_T$. Also, let N_j sources lie in box B_j, M_i targets lie in box C_i, and let the points be ordered such that

$$\begin{aligned}
\{s_1, \ldots, s_{N_1}\} &\subset B_1 \,, \\
\{s_{N_1+1}, \ldots, s_{N_1+N_2}\} &\subset B_2 \,, \\
&\cdots \\
\{s_{n-N_S+1}, \ldots, s_n\} &\subset B_S \,, \\
\{t_1, \ldots, t_{M_1}\} &\subset C_1 \,, \\
\{t_{M_1+1}, \ldots, t_{M_1+M_2}\} &\subset C_2 \,, \\
&\cdots \\
\{t_{m-M_T+1}, \ldots, t_m\} &\subset C_T \,.
\end{aligned}$$

Then, the approximation G_ϵ of the matrix G of Gaussian interactions can be factorized as

$$G_\epsilon = DEF \,, \tag{5.12}$$

where F and D are block diagonal matrices with $S \times S$ and $T \times T$ blocks, respectively, and E is a block matrix with $T \times S$ blocks and at most $(2b+1)^d$ nonzero blocks per row. Figure 5.1 shows

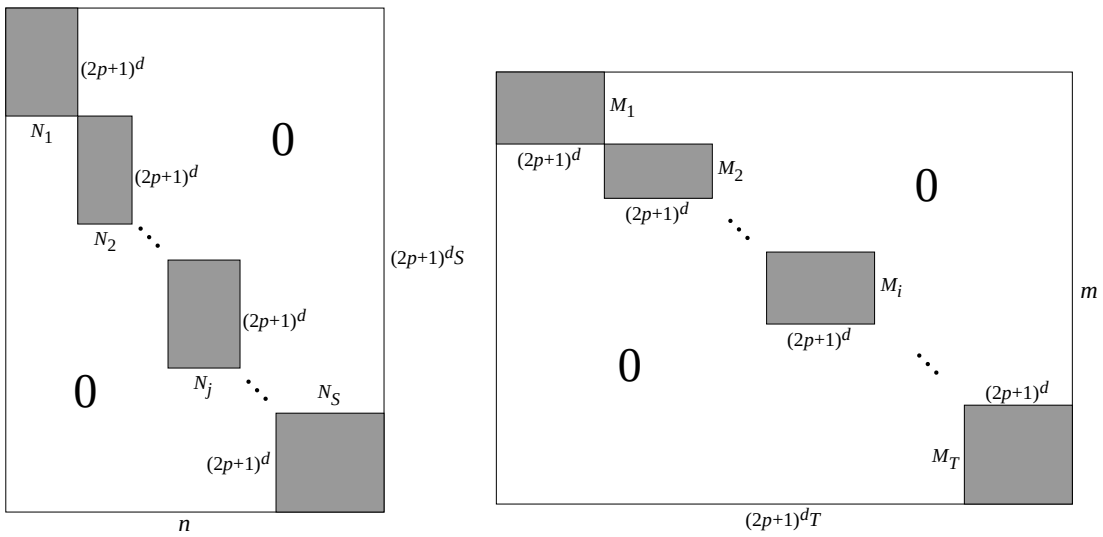


Figure 5.1: The structures of block diagonal matrices F and D. (a) The $(2p+1)^d S \times n$ matrix F has $S \times S$ blocks, where each diagonal block corresponds to a distinct source box. The jth diagonal block of matrix F has dimensions $(2p+1)^d \times N_j$ for $j = 1, 2, \ldots, S$; each column of the jth diagonal block corresponds to the plane-wave expansion of a distinct source within box B_j. (b) The $m \times (2p+1)^d T$ matrix D has $T \times T$ blocks, where each diagonal block corresponds to a distinct target box. The ith diagonal block of matrix D has dimensions $M_i \times (2p+1)^d$ for $i = 1, 2, \ldots, T$; each row of the ith diagonal block corresponds to the plane-wave expansion of a distinct target within box C_i.

the structures of matrices F and D.

Each diagonal block of matrix F corresponds to a distinct source box, and each column of the jth diagonal block $F(j)$ with dimensions $(2p+1)^d \times N_j$ for $j = 1, 2, \ldots, S$ of matrix F corresponds to the plane-wave expansion of a distinct source lying within box B_j. Matrix F has dimensions $(2p+1)^d S \times n$. The block diagonal matrix D with $T \times T$ blocks is similar to matrix F, except that each diagonal block in D corresponds to a distinct target box. In matrix D, the ith diagonal block, where $i = 1, 2, \ldots, T$, has dimensions $M_i \times (2p+1)^d$, and each row of the ith diagonal block corresponds to the plane-wave expansion of a distinct target lying within box C_i. Matrix D has dimensions $m \times (2p+1)^d T$.

As mentioned earlier, E is a block matrix with $T \times S$ blocks and at most $(2b+1)^d$ nonzero blocks per row. Block $E(i, j)$ for $1 \leq i \leq T$ and $1 \leq j \leq S$ of matrix E is nonzero when source box B_j lies in the interaction region of target box T_i. Each nonzero block $E(i, j)$ is a diagonal matrix with dimensions $(2p+1)^d \times (2p+1)^d$, whose diagonal entries are given by $e^{i\lambda\beta\cdot(t_{C_i} - s_{B_j})}$, where $-p \leq \beta \leq p$.

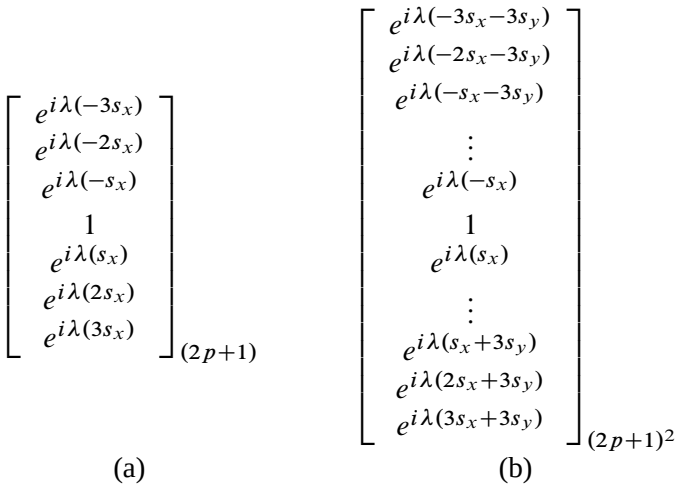


Figure 5.2: Vectors showing plane-wave coefficients for a single source s, assuming that $p = 3$. (a) The vector for $s = (s_x)$ with $d = 1$. (b) The vector for $s = (s_x, s_y)$ with $d = 2$. This vector gets scaled by q_s for further calculations.

We can utilize the factorized notation of G_ϵ from equation (5.12) to compute the fast Gauss transform using matrix-vector products. The S2W step of the algorithm corresponds to multiplying the $(2p+1)^d S \times n$ matrix F with the transpose of the row vector $\{q_1, q_2, \ldots, q_n\}$, resulting in the $(2p+1)^d S$-vector F'. The W2L step of the algorithm corresponds to multiplying the $(2p+1)^d T \times (2p+1)^d S$ matrix E with vector F', resulting in the $(2p+1)^d T$-vector E'. The final step of the algorithm corresponds to multiplying the $m \times (2p+1)^d T$ matrix D with vector E' in order to approximate the Gaussian (5.1) at all m target locations.

5.4 Performance improvements

In this section, we elaborate on some methods to improve the performance of computing the fast Gauss transform described by Spivak, Veerapaneni, and Greengard [47]. Because our parallel implementation is based on the plane-wave version of the fast Gauss transform, we will focus on improving the steps of the algorithm presented in Section 5.3.

5.4.1 S2W step

Figures 5.2 (a) and (b) show the $(2p+1)^d$ plane-wave coefficients for a single source $s = (s_x)$ with $d = 1$, and $s = (s_x, s_y)$ with $d = 2$, respectively, assuming that $p = 3$. As we noted in

Section 5.3, step 1 requires $O(p^d n)$ operations to complete because we have to evaluate $(2p+1)^d$ plane-wave terms for each source. As Figures 5.2 (a) and (b) illustrate, the terms that are equidistant from the middle element (which always evaluates to 1) are complex conjugates of each other. We can use this observation to easily save a factor of 2 in the computation. We can do better, however, if we know something about the source points.

Regular grids

In each source box B, if the source points belong to a regular grid (known as a tensor grid in the literature), we can apply the technique of separation of variables to accelerate the S2W step. Let $d = 3$ and suppose that the vertices of a $\nu \times \nu \times \nu$ regular grid form the source points of a box B, such that $N_B = \nu^3$. Spivak, Veerapaneni, and Greengard [47] show how to form the plane-wave expansions in each box in $O(p\nu^3 + p^2\nu^2 + p^3\nu)$ operations, so that if $\nu > p$, the total cost of forming the expansions for box B reduces to $O(pN_B)$ operations. Assuming that the source points in each box lie on a regular grid, and because $n = \sum_B N_B$, the net cost of the S2W step reduces to $O(pn)$ operations [47]. In d dimensions, computing the S2W step for a single box takes $O(dpN_B)$ time, and over all boxes it takes $O(dpn)$ time.

Let the ν^3 source points in box B, centered at $(s_{B_x}, s_{B_y}, s_{B_z})$, lie on the regular grid $\{(s_{j_1}, s_{j_2}, s_{j_3}) \ : \ j_1, j_2, j_3 = 1, \ldots, \nu\}$. For box B, we would like to compute the $(2p+1)^3$ plane-wave terms, which we denote by $\{w_{\beta_1\beta_2\beta_3} \ : \ \beta_1, \beta_2, \beta_3 = -p, \ldots, p\}$, using the source charges $\{q_{j_1 j_2 j_3} \ : \ j_1, j_2, j_3 = 1, \ldots, \nu\}$. As Figure 5.3 shows, we can achieve the reduced running time in three stages, where we unroll one dimension of the expansion formation in each stage.

Note that in each of the three stages in the pseudocode of Figure 5.3, the dot product in the exponentiation involves two scalar quantities. For example, β_1 is the first component of the multi-index β and $(s_{B_x} - s_{j_1})$ is the difference between the x-components of a source's box center and itself. Both β_1 and $(s_{B_x} - s_{j_1})$ are scalar values. Similarly, $\beta_2, \beta_3, (s_{B_y} - s_{j_2})$, and $(s_{B_z} - s_{j_3})$ are scalars. Clearly, stages 1, 2, and 3 in the pseudocode of Figure 5.3 require time $O(p\nu^3)$, $O(p^2\nu^2)$, and $O(p^3\nu)$, respectively. Computing the plane-wave expansions for all source boxes, therefore, requires $O(pn)$ operations, assuming that $\nu > p$ and $d = 3$. In general, the S2W step takes $O(dpn)$ time if $\nu > p$.

S2W-STAGE1

for $\beta_1 = -p$ to p

  for $j_2, j_3 = 1$ to ν

    $w_{\beta_1}(j_2, j_3) = \sum_{j_1=1}^{\nu} q_{j_1 j_2 j_3} e^{i\lambda\beta_1 \cdot (s_{B_x} - s_{j_1})}$

S2W-STAGE2

for $\beta_1, \beta_2 = -p$ to p

  for $j_3 = 1$ to ν

    $w_{\beta_1\beta_2}(j_3) = \sum_{j_2=1}^{\nu} w_{\beta_1}(j_2, j_3) e^{i\lambda\beta_2 \cdot (s_{B_y} - s_{j_2})}$

S2W-STAGE3

for $\beta_1, \beta_2, \beta_3 = -p$ to p

  $w_{\beta_1\beta_2\beta_3} = \left(\frac{L}{2p\sqrt{\pi}}\right)^3 \sum_{j_3=1}^{\nu} w_{\beta_1\beta_2}(j_3) e^{i\lambda\beta_3 \cdot (s_{B_z} - s_{j_3})}$

Figure 5.3: Pseudocode from [47] for the S2W step for a box, assuming that points in the box lie on a regular grid.

As an aside, for each source box B, the pseudocode in Figure 5.3 computes the product of the corresponding diagonal block in matrix F with the charge values of the sources in this box. That is, executing the pseudocode in Figure 5.3 for a source box B gives us a set of $(2p+1)^d$ continuous values in vector F' that correspond to the pane-wave expansion for box B.

5.4.2 Sweeping algorithm for W2L step

Recall from Section 5.3 that in the W2L step, for each target box, we accumulate the plane-wave expansions from all source boxes that lie in the interaction region of the target box. In the naive method, we simply sum the plane-wave expansions of all the source boxes in a target box's interaction region, which requires $O(b^d\, p^d\, N_{box})$ operations, where N_{box} is the total number of boxes. Greengard and Sun [28], however, came up with a sweeping algorithm for speeding up this step, which we now describe. For convenience, we will assume that the source and target boxes coincide so that we can denote all boxes by B_j for $j = 0, \ldots, N_{box} - 1$, and their respective centers by s_{B_j}.

Figure 5.4 depicts the idea behind the sweeping algorithm when boxes lie in one dimension. For the leftmost box, B_0, we sum the plane-wave expansions of the $(b+1)$ boxes—box B_0 and the b boxes to the right of B_0—using the direct method, which takes $O(bp)$ time. Let us denote the

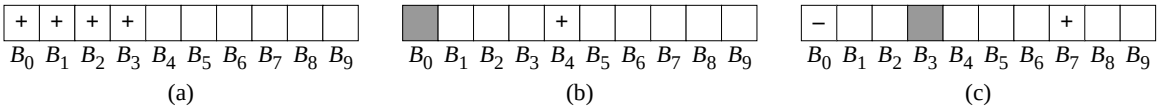


Figure 5.4: Sweeping algorithm for box-to-box translations when boxes lie in one dimension, for $b = 3$. (a) For box B_0, we calculate the local expansion directly by adding the plane-wave expansions of boxes B_0–B_3, indicated by plus signs in these boxes. (b) To calculate the local expansion of box B_1, we can reuse the local expansion of box B_0, which we indicate by shading box B_0, and add the plane-wave expansion of box B_4. (c) When we reuse the local expansion of box B_3 to calculate the local expansion for box B_4, we must subtract the plane-wave expansion of box B_0 (which participates in the local expansion of box B_3), shown using a minus sign in box B_0, and add the plane-wave expansion of box B_7. For all boxes B_1–B_9 shown in the figure, we must appropriately adjust for box centers.

local expansion of box B_0 by l_0 and an individual term in the local expansion[1] of box B_0 by $l_{0,\beta}$ for $-p \leq \beta \leq p$. Then, if we assume that $b = 3$, we can write

$$\begin{aligned} l_{0,\beta} \;=\; & w_{B_0,\beta}\, e^{i\lambda\beta \cdot (s_{B_0} - s_{B_0})} + w_{B_1,\beta}\, e^{i\lambda\beta \cdot (s_{B_0} - s_{B_1})} + w_{B_2,\beta}\, e^{i\lambda\beta \cdot (s_{B_0} - s_{B_2})} + \\ & w_{B_3,\beta}\, e^{i\lambda\beta \cdot (s_{B_0} - s_{B_3})}\,, \end{aligned}$$

where w_{B_j} denotes the plane-wave expansion of box B_j computed in the S2W step. We ignore all boxes that lie outside the FGT grid, which is why there are no boxes to the left of box B_0 that can be included.

To calculate the local expansion for box B_1, we can reuse l_0 (after adjusting for box centers) and just add in the contribution from box B_4, as shown in Figure 5.4(b). Thus, we have

$$l_{1,\beta} = l_{0,\beta}\, e^{i\lambda\beta \cdot (s_{B_1} - s_{B_0})} + w_{B_4,\beta}\, e^{i\lambda\beta \cdot (s_{B_1} - s_{B_4})}\,, \tag{5.13}$$

for $-p \leq \beta \leq p$.

The above observation forms the basis for the sweeping algorithm. In general, box B_{j-1}'s interaction list consists of boxes $\{B_{j-b-1}, \ldots, B_{j-1}, \ldots, B_{j+b-1}\}$ and that of box B_j consists of boxes $\{B_{j-b}, \ldots, B_j, \ldots, B_{j+b}\}$ (again, we ignore all box numbers that lie outside the range $[1, N_{box}]$). Therefore, we can use the standard stenciling technique to compute the local expansion for box B_j from that of box B_{j-1}: reuse the local expansion of box B_{j-1}, but subtract the contribution from box B_{j-b-1} (the leftmost member of box B_{j-1}'s interaction list), and add

[1] In one dimension, each box has $(2p + 1)$ terms in its local expansion.

the contribution from box B_{j+b} (the rightmost member of box B_j's interaction list). Of course, we must adjust appropriately for box centers, as we showed in our calculation for box B_1 above. Assuming that the vectors required for adjusting the box centers are precomputed, the cost of forming the local expansion of a box is reduced from $(2b+1)p$ operations to $3p$ operations, for a total cost of $O(pN_{box})$ over all boxes, when the boxes lie in one dimension. For a complete example, we show the calculation for box B_4 below and illustrate it pictorially in Figure 5.4(c); as before, if we assume that $b = 3$, then for $-p \leq \beta \leq p$, we have

$$l_{4,\beta} = l_{3,\beta}\, e^{i\lambda\beta\cdot(s_{B_4}-s_{B_3})} - w_{B_0,\beta}\, e^{i\lambda\beta\,\cdot\,(s_{B_4}-s_{B_0})} + w_{B_7,\beta}\, e^{i\lambda\beta\,\cdot\,(s_{B_4}-s_{B_7})}\,.$$

We can extend the sweeping algorithm to multiple dimensions by performing a sequence of d one-dimensional sweeps [47]. As mentioned earlier, we assume that the unit box $[0,1]^d$ is subdivided into smaller boxes and that each box has side length $\sqrt{\delta}$ in each dimension. Therefore, we have $1/\sqrt{\delta}$ boxes along each dimension, for a total of $(1/\sqrt{\delta})^d$ boxes. Let $\gamma = 1/\sqrt{\delta}$ denote the number of boxes in each dimension. Then, we can number the γ^d FGT boxes as $B_j = B_{(j_1,j_2,\ldots,j_d)}$, where $0 \leq j_k \leq \gamma - 1$, for $k = 1,\ldots,d$. Box B_j's interaction region consists of at most $(2b+1)^d$ boxes $B_{(r_1,r_2,\ldots,r_d)}$, such that $j_k - b \leq r_k \leq j_k + b$ and $0 \leq r_k \leq \gamma - 1$, for $k = 1,2,\ldots,d$. In the sweep along the first dimension, each box collects the expansions from $(2b+1)$ boxes in its interaction region. After the sweep along the second dimension, each box has gathered the necessary expansions from $(2b+1)^2$ boxes in its interaction region, so that after the dth sweep, each box has the required expansions from all the $(2b+1)^d$ boxes in its interaction region. The sweep along each dimension requires $3p^d N_{box}$ operations, where $N_{box} = \gamma^d$ and, therefore, the W2L step requires $O(dp^d N_{box})$ time in total, using the sweeping algorithm. This running time is considerably less than the running time of $O(b^d p^d N_{box})$ using the naive method.

Note that vectors of the form $e^{i\lambda\beta\cdot(s_{B_i}-s_{B_j})}$ that get multiplied with w_{B_j} correspond to the diagonal entries of nonzero block $E(i,j)$ of matrix E. In the sweeping algorithm, we avoid computing all $(2b+1)^d$ nonzero blocks in a row of matrix E, however.

5.4.3 L2T step

Similar to the S2W step, we can speed up the L2T step if we know that the targets lie on a regular grid. Again, if we assume that $d = 3$ and that the targets in each box lie on the vertices of a $\nu \times \nu \times \nu$ regular grid, then we can reduce the running time for the L2T step from $O(p^3 m)$ to $O(pm)$, assuming that $\nu > p$.

As before, we suppose that in target box C, with center $(t_{C_x}, t_{C_y}, t_{C_z})$, the ν^3 target points lie on the regular grid $\{(t_{j_1}, t_{j_2}, t_{j_3}) \ : \ j_1, j_2, j_3 = 1, \ldots, \nu\}$. In box C, for each target t, we would like to compute the sum of the elementwise product of the $(2p+1)^3$ terms of the plane-wave expansion of this target with the local expansion for this box (see equation (5.11)). We compute $g(t)$ for all targets in box C in three stages, as we did in the S2W step. Below, we provide pseudocode for all the three stages, in which we denote the $(2p+1)^3$ terms of the local expansion of box C by $l_{C,\beta_1\beta_2\beta_3}$, where each component β_1, β_2, and β_3 of multi-index β ranges between $-p$ and p.

L2T-STAGE1

for $j_1 = 1$ to ν
  for $\beta_2, \beta_3 = -p$ to p
    $g_{j_1}(\beta_2, \beta_3) = \sum_{\beta_1=-p}^{p} l_{C,\beta_1\beta_2\beta_3} e^{i\lambda\beta_1 \cdot (t_{j_1} - t_{C_x})}$

L2T-STAGE2

for $j_1, j_2 = 1$ to ν
  for $\beta_3 = -p$ to p
    $g_{j_1 j_2}(\beta_3) = \sum_{\beta_2=-p}^{p} g_{j_1}(\beta_2, \beta_3) e^{i\lambda\beta_2 \cdot (t_{j_2} - t_{C_y})}$

L2T-STAGE3

for $j_1, j_2, j_3 = 1$ to ν
  $g_{j_1 j_2 j_3} = \sum_{\beta_3=-p}^{p} g_{j_1 j_2}(\beta_3) e^{i\lambda\beta_3 \cdot (t_{j_3} - t_{C_z})}$

For any target box C, stages 1, 2, and 3 take time $O(p^3\nu)$, $O(p^2\nu^2)$, and $O(p\nu^3)$, respectively, to execute.[2] Because we assumed that $\nu > p$, we can say that the total running time for computing the fast Gauss transform for all targets in a single box is $O(p\nu^3)$ and, therefore, the total running time for all target boxes is $O(pm)$ when $d = 3$. In general, the total running time for the L2T step when the target points in each box are assumed to lie on a regular grid is $O(dpm)$ if $\nu > p$.

[2] Similar to the S2W step, the dot products involve scalar quantities.

Referring back to our matrix-vector-product notation, for any target box C, the pseudocode above computes the product of the diagonal block corresponding to box C in matrix D with the local expansion for box C stored in vector E'. Executing all three stages of the L2T step for a target box C thus approximates the Gaussian (5.1) for all targets in box C.

5.5 Implementation details

In this section, we discuss some basic details of our fast Gauss transform implementation. We begin by stating a few assumptions that we make in our implementation before continuing with some specifics of the implementation.

In the rest of this chapter, we assume that $d = 3$, the source and target points coincide, and that all points lie within the unit cube $[0, 1]^3$. Along each axis, we divide the domain into $\gamma = 1/\sqrt{\delta}$ equal parts, so that the side length of each box along each dimension is $\sqrt{\delta}$ and the total number of boxes is γ^3. Furthermore, we assume that within each box, the sources and targets lie on a regular grid. We will assume that the grid of points in any box has ν points along each dimension for a total of ν^3 points, and we will denote the fixed distance between any two vertices on the grid along any dimension by $\Delta = \sqrt{\delta}/\nu$.

Given the regularity of the boxes and the points within them, we do not store the source and target points but calculate them at run time, though we do store the random charge values for each source point. In our code, we use BLAS routines [7] wherever applicable. Although BLAS routines support operations on complex numbers and we generate complex values in all the FGT steps,[3] we use the `double` counterparts of these routines because in our code we store the real and imaginary parts of a complex vector separately. In all the FGT steps, we use the well-known formula

$$e^{ix} = \cos x + i \sin x$$

in our calculations.

[3]Note, however, that in the last step, L2T, although we use the imaginary parts of vectors for intermediate calculations, we retain only the real part of the calculations as the final answer for each target.

5.5.1 S2W and L2T steps

In addition to the methods presented for improving the running time of the FGT steps in Section 5.4, we can save some additional computations in the implementation. Note that computing $\cos x$ and $\sin x$ is usually costly, and given that we have to compute many sine and cosine values, we would like to reuse values whenever possible. Again, our assumption that the sources and targets lie on a regular grid proves helpful in achieving this goal in the S2W and L2T steps. Let us see how, by taking the pseudocode for stage 1 of the S2W step for some box B as an example. For convenience, we replicate the pseudocode below.

S2W-STAGE1
  for $\beta_1 = -p$ to p
    for $j_2, j_3 = 1$ to ν
      $w_{\beta_1}(j_2, j_3) = \sum_{j_1=1}^{\nu} q_{j_1 j_2 j_3} e^{i\lambda\beta_1 \cdot (s_{B_x} - s_{j_1})}$

In the above pseudocode, we exhaust all calculations based on a single value of β_1 before moving further. The summation in the innermost loop, however, seems to require ν calculations of $\cos\theta_{j_1}$ and $\sin\theta_{j_1}$, where $\theta_{j_1} = \lambda\beta_1(s_{B_x} - s_{j_1})$ for $j_1 = 1, \ldots, \nu$. As we noted in Section 5.4.1, because β_1 and $(s_{B_x} - s_{j_1})$ are scalars, $e^{i\lambda\beta_1 \cdot (s_{B_x} - s_{j_1})} = e^{i\lambda\beta_1(s_{B_x} - s_{j_1})}$. By our assumption, however, $s_{j_1+1} = s_{j_1} + \Delta$ for $j_1 = 1, \ldots, \nu - 1$. The first point s_1 can be easily calculated for any box, as we now describe. We number the γ^3 boxes from 0 to $\gamma^3 - 1$. If we imagine the γ^3 boxes to be arranged in γ layers, with γ^2 boxes in each layer, then we can represent each box by its x-, y-, and z-coordinates, from (0, 0, 0) to $(\gamma - 1, \gamma - 1, \gamma - 1)$. For any box $B = (B_x, B_y, B_z)$, we calculate $B_x = B \bmod \gamma$; then, if we assume that j_1 spans the x-dimension, we get $s_1 = B_x\sqrt{\delta}$. Clearly, the x-coordinate of box B's center is $s_{B_x} = s_1 + \sqrt{\delta}/2$. With these observations, we can implement stage 1 above for any box B with just eight sine and cosine computations, as depicted below.

S2W-STAGE1

1 $B_x = B \bmod \gamma$
2 $s_1 = B_x * \sqrt{\delta}$
3 $s_{B_x} = s_1 + \sqrt{\delta}/2$
// store $e^{i(-p\lambda(s_{B_x}-s_1))}$ in two variables, where $(s_{B_x} - s_1) = \sqrt{\delta}/2$
4 *src_real* $= \cos(\lambda * -p * (s_{B_x} - s_1))$
5 *src_imag* $= \sin(\lambda * -p * (s_{B_x} - s_1))$

// store $e^{i\lambda(s_{B_x}-s_1)}$ in two variables
6 *inc_src_real* $= \cos(\lambda * (s_{B_x} - s_1))$
7 *inc_src_imag* $= \sin(\lambda * (s_{B_x} - s_1))$

// compute $e^{i(-p\lambda(-\Delta))}$ and $e^{i\lambda(-\Delta)}$
8 *step_real* $= \cos(\lambda * -p * -\Delta)$
9 *step_imag* $= \sin(\lambda * -p * -\Delta)$
10 *inc_step_real* $= \cos(\lambda * -\Delta)$
11 *inc_step_imag* $= \sin(\lambda * -\Delta)$

12 **for** $\beta_1 = -p$ to p
13 **for** $j_2, j_3 = 1$ to ν
14 *temp_real* = *src_real* // start with $e^{i\lambda\beta_1(s_{B_x}-s_1)}$
15 *temp_imag* = *src_imag*
16 *sum_real* = 0.0
17 *sum_imag* = 0.0
// expand the summation
18 **for** $j_1 = 1$ to ν
19 *sum_real* += $q_{j_1 j_2 j_3}$ *temp_real*
20 *sum_imag* += $q_{j_1 j_2 j_3}$ *temp_imag*
// update for the next source point $s_{j_1+1} = s_{j_1} + \Delta$
// $e^{i\lambda\beta_1(s_{B_x}-s_{j_1+1})} = e^{i\lambda\beta_1(s_{B_x}-s_{j_1})} e^{i\lambda\beta_1(-\Delta)}$
21 *temp* = *temp_real* $*$ *step_real* $-$ *temp_imag* $*$ *step_imag*
22 *temp_imag* = *temp_real* $*$ *step_imag* $+$ *temp_imag* $*$ *step_real*
23 *temp_real* = *temp*
// store the result
24 $w_{\beta_1}(j_2, j_3) = (sum\_real, sum\_imag)$ // representing a complex number as a tuple
// update for the next value of β_1
25 *temp* = *src_real* $*$ *inc_src_real* $-$ *src_imag* $*$ *inc_src_imag*
26 *src_imag* = *src_real* $*$ *inc_src_real* $+$ *src_imag* $*$ *inc_src_imag*
27 *src_real* = *temp*

28 *temp* = *step_real* $*$ *inc_step_real* $-$ *step_imag* $*$ *inc_step_imag*
29 *step_imag* = *step_real* $*$ *inc_step_real* $+$ *step_imag* $*$ *inc_step_imag*
30 *step_real* = *temp*

In the above pseudocode, we determine the x-coordinate of the box, the x-coordinate of its first source point, and the x-coordinate of its center in lines 1–3. In lines 4–11, we use eight

variables to store the values of $e^{i(-p\lambda(s_{B_x}-s_1))}$, $e^{i\lambda(s_{B_x}-s_1)}$, $e^{i(-p\lambda(-\Delta))}$, and $e^{i\lambda(-\Delta)}$. Each iteration of the **for** loop of line 18 gathers a portion of the result and computes the next value of $e^{i\lambda\beta_1(s_{B_x}-s_{j_1})}$ for a given β_1. After the **for** loop of line 18 finishes, the code stores the computed result into $w_{\beta_1}(j_2, j_3)$ in line 24, and before starting the next iteration of the **for** loop of line 12, the code calculates $e^{i\lambda(\beta_1+1)(s_{B_x}-s_1)}$ and $e^{i\lambda(\beta_1+1)(-\Delta)}$. Note that we use $-\Delta$ in the exponentiation because in the S2W step, we translate source points with respect to the center of their box. Therefore, for any $j_1 = 1, \ldots, \nu - 1$, we have $e^{i\lambda\beta_1(s_{B_x}-s_{j_1+1})} = e^{i\lambda\beta_1(s_{B_x}-(s_{j_1}+\Delta))} = e^{i\lambda\beta_1(s_{B_x}-s_{j_1})}e^{i\lambda\beta_1(-\Delta)}$. As highlighted earlier, the code computes only a few sines and cosines to complete stage 1. We can apply similar techniques to compute stages 2 and 3 of the S2W step, for which we do not show pseudocode. Note, however, that for a box $B = (B_x, B_y, B_z)$, we calculate $B_y = \lfloor B/\gamma \rfloor \bmod \gamma$ and $B_z = \lfloor B/\gamma^2 \rfloor$.

The pseudocode above works well for a regular grid. If, however, the grid within each box is adaptive—i.e., if the step value, denoted by Δ, is not constant between each successive source point along any dimension—then we would require a few more sine and cosine computations. Let us suppose that the step values between successive source points along any dimension are given by $\Delta_{j_1} = s_{j_1+1} - s_{j_1}$ for $j_1 = 1, \ldots, \nu$, and let $\Delta_\nu = 0$. Now, if we use four vectors of size ν each to store the real and imaginary parts of $e^{i(-p\lambda(-\Delta_{j_1}))}$ and $e^{i\lambda(-\Delta_{j_1})}$, we can proceed similar to the pseudocode above for constant Δ, but we must use appropriate vector values to update variables in lines 21–23 and lines 28–30. Therefore, we now have to compute 4ν sine and cosine values in stage 1 instead of four earlier (see lines 8–11) to accommodate the irregular step sizes.

We can employ similar methods to compute the L2T step, though there are slight differences, which we now highlight. Below, we replicate the pseudocode for stage 1 of the L2T step.

L2T-STAGE1
for $j_1 = 1$ to ν
  for $\beta_2, \beta_3 = -p$ to p
    $g_{j_1}(\beta_2, \beta_3) = \sum_{\beta_1=-p}^{p} l_{C,\beta_1\beta_2\beta_3} e^{i\lambda\beta_1 \cdot (t_{j_1}-t_{C_x})}$

In the pseudocode above, we exhaust all calculations based on a single target coordinate along the x-dimension before updating the target coordinate along the same dimension. Similar to stage 1 of the S2W step, calculating eight sine and cosine values will be enough. The pseudocode below, though less elaborate than what we presented for the S2W step, clearly establishes our point.

L2T-STAGE1

1 $C_x = C \bmod \gamma$
2 $t_1 = C_x * \sqrt{\delta}$
3 $t_{C_x} = t_1 + \sqrt{\delta}/2$ // box center
4 store the real and imaginary parts of $e^{i(-p\lambda(t_1 - t_{C_x}))}$ in two variables
5 store the real and imaginary parts of $e^{i\lambda(t_1 - t_{C_x})}$ in two variables
6 store the real and imaginary parts of $e^{i(-p\lambda\Delta)}$ in two variables
7 store the real and imaginary parts of $e^{i\lambda\Delta}$ in two variables

8 **for** $j_1 = 1$ to ν
9 **for** $\beta_2, \beta_3 = -p$ to p
10 $temp = e^{i(-p\lambda(t_{j_1} - t_{C_x}))}$ // complex *temp*
11 $sum = (0, 0)$ // complex *sum*
// expand the summation
12 **for** $\beta_1 = -p$ to p
13 $sum \mathrel{+}= l_{C,\beta_1\beta_2\beta_3} * temp$ // complex multiplication and addition
// update for next β_1
14 $temp = temp * e^{i\lambda(t_{j_1} - t_{C_x})}$ // complex multiplication
// store the result
15 $g_{j_1}(\beta_2, \beta_3) = sum$
// update for the next target
16 $next = j_1 + 1$
17 $e^{i(-p\lambda(t_{next} - t_{C_x}))} = e^{i(-p\lambda(t_{j_1} - t_{C_x}))} * e^{i(-p\lambda\Delta)}$
18 $e^{i\lambda(t_{next} - t_{C_x})} = e^{i\lambda(t_{j_1} - t_{C_x})} * e^{i\lambda\Delta}$

Above, we use $+\Delta$ in the exponentiation because in the L2T step, we translate the box center with respect to the target points, so $(t_{j_1+1} - t_{C_x}) = (t_{j_1} + \Delta - t_{C_x})$. As we can see, we were able to implement stage 1 using just eight sine and cosine computations instead of 4ν computations. If, however, the grid of points within a box is not uniform, so that $\Delta_{j_1} = t_{j_1+1} - t_{j_1}$ for $j_1 = 1, \ldots, \nu$, we might be tempted to think that we will not benefit much by computing four vectors with ν values each in lines 6–7 above and using these vectors in lines 17–18.[4] We would be wrong, however, because we can reuse these values in stages 2 and 3.

5.5.2 W2L step

In the W2L step, we require many vectors, each of size $(2p+1)^3$, to store $e^{i(\lambda\beta\cdot(s_B - s_{B'}))}$ for $-p \le \beta \le p$. Because we sweep one dimension at a time, exactly one of the components of $(s_B - s_{B'})$ is nonzero. Furthermore, because all boxes have side lengths $\sqrt{\delta}$, the nonzero component is a mul-

[4]Note that even if we were not to pre-calculate $e^{i(-p\lambda\Delta_{j_1})}$ and $e^{i\lambda\Delta_{j_1}}$ for $j_1 = 1, \ldots, \nu$, we would still have to compute only 4ν sine and cosine values in lines 17–18.

tiple of $\sqrt{\delta}$. For example, let's look at Figure 5.5, which shows 512 boxes arranged in an $8 \times 8 \times 8$ grid and numbered from 0–511. As we can see from the figure, box B_0's immediate neighbor in the x-, y-, and z-directions are boxes B_1, B_8, and B_{64}, respectively (box B_{64} is hidden from view in the figure). Box B_0 has center $(\sqrt{\delta}/2, \sqrt{\delta}/2, \sqrt{\delta}/2)$, box B_1 has center $(3\sqrt{\delta}/2, \sqrt{\delta}/2, \sqrt{\delta}/2)$, box B_8 has center $(\sqrt{\delta}/2, 3\sqrt{\delta}/2, \sqrt{\delta}/2)$, and box B_{64} has center $(\sqrt{\delta}/2, \sqrt{\delta}/2, 3\sqrt{\delta}/2)$. Therefore, $(s_{B_0} - s_{B_1}) = (-\sqrt{\delta}, 0, 0)$, $(s_{B_0} - s_{B_8}) = (0, -\sqrt{\delta}, 0)$, and $(s_{B_0} - s_{B_{64}}) = (0, 0, -\sqrt{\delta})$. In fact, for all boxes $B = (B_x, B_y, B_z)$ such that $B_x = 0$, the difference between B's center and that of any of its b neighbors to the right is $(s_B - s_{B'}) = ((B_x - B_{x'})\sqrt{\delta}, 0, 0)$, where $B' = (B_{x'}, B_y, B_z)$ and $B_{x'} = B_x + 1, \ldots, B_x + b$. That is, the x-component of $(s_B - s_{B'})$ ranges from $(-\sqrt{\delta})$ to $(-b\sqrt{\delta})$ for the specific boxes under consideration. Using this observation, we calculate vector $V = e^{i\lambda\beta\cdot(s_B - s_{B'})}$ for $-p \leq \beta \leq p$, with $(2p+1)^3$ elements, such that $(s_B - s_{B'}) = (-\sqrt{\delta}, 0, 0)$. Note that for all γ^2 boxes with $B_x = 0$, vector V provides the required adjustment of each box center with its immediate right neighbor during the x-sweep. Let us define vector V^2 with $(2p+1)^3$ elements to be the elementwise product of vector V with itself; define vectors $V^3, \ldots, V^b$, similarly. For all boxes with $B_x = 0$, vectors $V^2, V^3, \ldots, V^b$ provide the necessary adjustments of box centers with their remaining $(b-1)$ neighbors to the right. Recall that in the sweeping algorithm, during the sweep along the x-dimension, the vectors $V, V^2, \ldots, V^b$ will be multiplied with the respective plane-wave expansions of the boxes that we calculated in the S2W step, as shown in the pseudocode below.

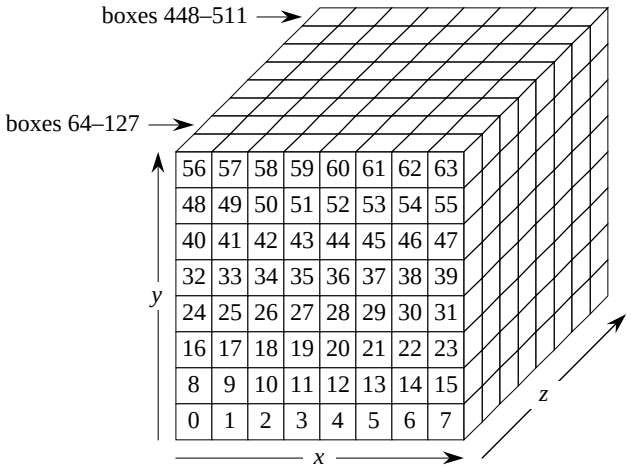


Figure 5.5: An $8 \times 8 \times 8$ grid of boxes numbered from 0 through 511. The numbers shown inside the first layer of boxes represent their respective box numbers. To reduce clutter, we don't label the boxes as B_0, B_1, etc., but simply show the box numbers.

MULTIDIMENSIONAL-SWEEP

// sweep along the first dimension
Calculate $V = e^{i\lambda\beta \cdot (-\sqrt{\delta},0,0)}$ for $-p \le \beta \le p$
Calculate $V^2, \ldots, V^b$
Calculate V^{b+1}, $V_{compensate} = \overline{V}$, and $V_{sub} = \overline{V^{b+1}}$
// process each horizontal row
for $j_2, j_3 = 0$ to $\gamma - 1$
    // Associate box number j with $(0, j_2, j_3)$
    $l_j = w_{B_j}$
    // accumulate the plane-wave expansions from all b boxes to the right of
    // box number $j = (0, j_2, j_3)$
    for $j_1 = 1$ to b
        // associate box number j' with (j_1, j_2, j_3)
        $l_j \mathrel{+}= w_{B_{j'}} * V^{j_1}$

    // compute the local expansion of box $(j_1 + 1, j_2, j_3)$
    for $j_1 = 0$ to $\gamma - 2$
        // associate box number j' with $(j_1 + 1, j_2, j_3)$ and
        // box number j with (j_1, j_2, j_3)
        $l_{j'} = l_j * V_{compensate}$
        // subtract influence from box $u = (j_1 - b, j_2, j_3)$, if $(j_1 - b) \ge 0$
        $l_{j'} \mathrel{-}= w_{B_u} * V_{sub}$
        // add influence from box $v = (j_1 + 1 + b, j_2, j_3)$, if $(j_1 + 1 + b) \le \gamma - 1$
        $l_{j'} \mathrel{+}= w_{B_v} * V^b$

Repeat analogous loops along the y- and z-dimensions; recalculate vectors $V, V^2, \ldots, V^b$, and use the appropriate l-vectors calculated in the previous sweep, instead of the w-vectors.

From the pseudocode, we note that during the sweep along the x-dimension, for all boxes such that $0 < B_x \leq \gamma - 1$, we simply need two additional vectors, namely $V_{compensate} = \overline{V}$, where $\overline{V}$ denotes the complex conjugate of vector V, and $V_{sub} = \overline{(V^{b+1})}$. Let's see why we need these complex conjugate vectors using boxes $B_0 = (0,0,0)$, $B_1 = (1,0,0)$, and $B_4 = (4,0,0)$ from Figure 5.5 as examples, and assuming that $b = 3$. Recall from Section 5.4.2 that we calculate l_0 directly, after which we can use l_0 to calculate l_1 (see equation (5.13)), but we must adjust for box centers. To calculate l_0, we translated the centers of all boxes in box B_0's interaction region with respect to the center of box B_0, denoted by s_{B_0}, whereas to calculate l_1, we must translate the box centers with respect to box B_1's center, denoted by s_{B_1}. Because $(s_{B_1} - s_{B_0}) = (\sqrt{\delta}, 0, 0)$, the vector $V_{compensate} = \overline{V}$ gives us the correct adjustment. In the case of box B_1, there is no box to subtract, and the center of box B_4, which should be added, is $-3\sqrt{\delta}$ away and so we can use $V^3 = V^b$. In the case of box B_4 where we have to subtract box B_0's contribution, we observe that $(s_{B_4} - s_{B_0}) = (4\sqrt{\delta}, 0, 0)$, and so we calculate $V_{sub} = \overline{V^4} = \overline{V^{b+1}}$.

Hence, during the x-sweep, for any box $B = (B_x, B_y, B_z)$ such that $0 < B_x \leq \gamma - 1$, if we label $B_c = (B_x - 1, B_y, B_z)$, $B_u = (B_x - b - 1, B_y, B_z)$, and $B_v = (B_x + b, B_y, B_z)$, ignoring any B_u and B_v whose x-component lies outside $[0, \gamma - 1]$, we can calculate

$$l_B = l_{B_c} V_{compensate} - l_{B_u} V_{sub} + l_{B_v} V^b .$$

During the x-sweep, we require just 10 sine and cosine values to calculate vector V (we do not provide details here except noting that we can exploit the pattern that the elements of vector V follow, as Figure 5.2(b) shows for $d = 2$). All the other vectors—$V^2, \ldots, V^{b+1}, V_{compensate}$, and V_{sub}—require only multiplications and additions. We can use similar observations and techniques for the sweeps along the y- and z-dimensions. While sweeping along the y-dimension, we calculate $V = e^{i\lambda\beta\cdot(0,-\sqrt{\delta},0)}$ for $-p \leq \beta \leq p$, and along the z-dimension, we calculate $V = e^{i\lambda\beta\cdot(0,0,-\sqrt{\delta})}$ for $-p \leq \beta \leq p$.

In this section, we saw how we can save many computations in all the three steps of the fast Gauss transform. The techniques highlighted in this section are used in our parallel implementation of the fast Gauss transform, discussed next.

5.6 Parallel implementation of fast Gauss transform

In this section, we discuss how we implemented the fast Gauss transform in parallel. We continue with our assumption that $d = 3$. We have implemented a shared-memory version and a distributed-memory version of the algorithm, both of which will be discussed in this section.

5.6.1 Shared-memory implementation

We have implemented the fast Gauss transform in shared memory in C++ and using OpenMP. In the shared-memory model, parallelizing the steps of the fast Gauss transform is fairly straightforward because in the S2W and L2T steps, we can work on different boxes in parallel. We implement the W2L step serially. We use OpenMP's `parallel for` loop to parallelize the S2W and L2T steps, and we use the `omp_set_num_threads()` function to specify the number of threads that the `parallel for` region should use. All assumptions that we made about the input points and the performance accelerations that we discussed in the previous two sections hold. Of course, because we are working with shared memory, we must be careful to provide non-overlapping pieces of memory to the regions that execute in parallel. Overall, we have a clean, parallel implementation of the fast Gauss transform in shared memory, as shown below. In the pseudocode below, all **parallel for** loops stand for OpenMP `parallel for` regions.

SHMEM-PARALLEL-FGT
1 **parallel for** *box_id* $= 0$ to $\gamma^3 - 1$
2 execute all three stages of S2W for this box
3 call MULTIDIMENSIONAL-SWEEP to compute the local expansion for all boxes
4 **parallel for** *box_id* $= 0$ to $\gamma^3 - 1$
5 execute all three stages of L2T for this box

5.6.2 Distributed-memory implementation

We have implemented the fast Gauss transform in a distributed-memory setting using FG, MPI, and OpenMP. Although we distribute data across the nodes of a cluster, the computation on each node takes place entirely in memory. That is, in this application, data does not reside on disk, which makes this application different from the other FG programs that we have seen till now. A distributed-memory implementation of FGT requires interprocessor communication for which we use MPI, and FG helps overlap computation with interprocessor communication. Within each node,

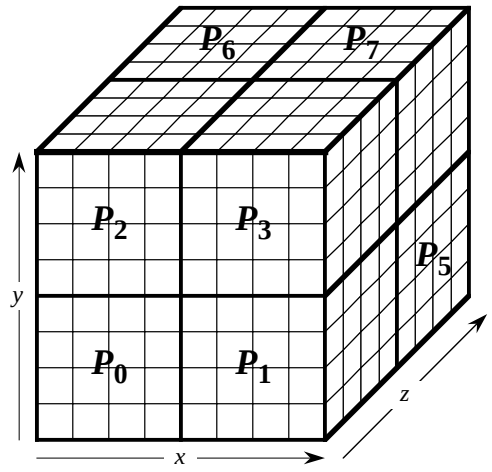


Figure 5.6: Distributing boxes across 8 nodes of a cluster, assuming that the unit cube $[0, 1]^3$ contains 512 boxes arranged as an $8 \times 8 \times 8$ grid. Each node becomes responsible for a $4 \times 4 \times 4$ subgrid of boxes. As the thicker lines demarcate, we can imagine the nodes themselves to be arranged as a $2 \times 2 \times 2$ grid, where each node owns 64 boxes. Node P_4 and its boxes are completely hidden from view.

some work can be done in parallel for which OpenMP proves useful. As we shall soon see, in this application, we also use FG as a signaling mechanism for events.

As with any distributed-memory application, we must first determine how to distribute the data so as to load-balance the work across all the available nodes. As we have seen, the S2W and L2T steps carry out box-wise computations, and so we would not like to split the data of a box across nodes. Therefore, as Figure 5.6 shows, we partition the unit cube $[0, 1]^3$ with γ^3 boxes arranged as a $\gamma \times \gamma \times \gamma$ grid across the available nodes such that each node is responsible for a subgrid of boxes. As mentioned earlier, all computation on a node takes place entirely in memory. With this setup, nodes obviously don't require to communicate with each other during the S2W and L2T steps because these steps work on data contained within a box. For each box, however, the W2L step accumulates data from at most $(2b + 1)^3$ neighboring boxes, some of which might belong to a different node, thus requiring interprocessor communication during this step.

A basic implementation of the fast Gauss transform with the setup that we just described could proceed as follows: during the S2W step, each node computes the plane-wave expansions of all boxes that it owns. Indeed, if a node has multiple cores, it could execute the S2W step on multiple boxes in parallel, as we described in the shared-memory implementation. For the W2L step, each node uses interprocessor communication to obtain the plane-wave expansions of all boxes that lie

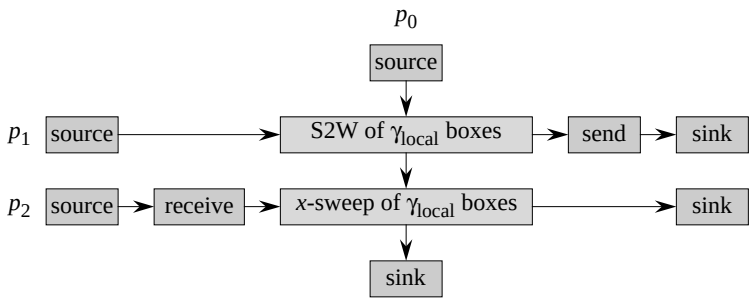


Figure 5.7: FG pipelines that each node executes to calculate the plane-wave expansions of all local boxes for the S2W step, and to sweep along the x-dimension for the W2L step.

in the interaction region of some box that this node owns. After the data exchange, each node can use the sweeping algorithm to compute the local expansions of all its boxes to complete the W2L step. The last step, L2T, is again completely local to a node and, therefore, each node can work on its set of boxes (in parallel, if the node is so equipped), to complete the algorithm. In the prospective design that we just described, only node-local computations are executed in parallel. We can, however, overlap computation and interprocessor communication for better performance, as we describe next.

5.6.3 S2W step and partial W2L step

In our implementation, each node executes the FG pipelines shown in Figure 5.7 to compute the plane-wave expansions of all local boxes, hence completing the S2W step, and to finish sweeping along the x-dimension, as part of the W2L step. Unlike the FG pipelines that we have seen so far, buffers along all pipelines of Figure 5.7 don't necessarily carry useful data; some buffers are dummy and are used only to signal the next stage to begin its computation. For example, the buffers that flow along pipelines p_0 and p_1 are dummy. The buffers along pipeline p_2 carry useful data, however: these buffers contain data that a node receives from other nodes. Before we elaborate on how these pipelines work, we introduce a few assumptions.

In our implementation, we assume that the number of nodes P is a perfect cube, so that $P = \rho^3$ for some ρ that divides γ. Since γ denotes the number of boxes in each dimension, each node works on $(\gamma/\rho)^3$ boxes. Let us denote $\gamma_{local} = (\gamma/\rho)$ to be the number of boxes in each dimension on any node. We also assume that $\gamma_{local} \geq b$ so that any node will have to communicate with at most two

other nodes for the W2L step.

On each node, the S2W stage, which is the intersecting stage along pipelines p_0 and p_1, generates the plane-wave expansions for all boxes local to a node. As Figure 5.6 shows, we can perceive all boxes local to a node to be arranged as a $\gamma_{local} \times \gamma_{local} \times \gamma_{local}$ grid. In each round, the S2W stage generates the plane-wave expansions of all boxes in a single row of the grid, for a total of γ_{local}^2 rounds. This stage processes the grid layer by layer, starting with the bottommost row and continuing toward the top in each layer. The buffers that this stage accepts along pipelines p_0 and p_1 are dummy buffers. The buffer along pipeline p_1 merely acts as a signal for this stage to start computing the next row of boxes. Along pipeline p_0, the S2W stage accepts a buffer and immediately conveys it just before returning from its stage function. One might wonder, then, what role the S2W stage plays along pipeline p_0. As Figure 5.7 shows, the S2W stage feeds into the x-sweep stage, and we shall soon see that in each round, the x-sweep stage can proceed only after the S2W stage has finished its computation. The buffer that the S2W stage conveys along pipeline p_0 acts as a go-ahead signal for the x-sweep stage. In the stage function for the S2W stage, we use OpenMP's `parallel for` loop to compute the plane-wave expansions of multiple boxes in parallel in an effort to completely utilize the multiple cores (if any) of a machine. Given that the S2W stage accepts and conveys only dummy buffers, where does it read its input from and write its output to? Recall that all input[5] and the results of all computations on a node are stored in memory. Therefore, in each round, the S2W stage knows exactly which preallocated area of memory to read its input from and write its output to.

Using an example, we first describe a prospective method to compute the x-sweep of all boxes. In the next paragraph, we detail our actual implementation. Let us assume, as in Figure 5.8, that $P = 64$ so that $\rho = 4$, $\delta = 1/256$ so that $\gamma = 16$ and $\gamma_{local} = 4$, and $b = 3$. In this global $16 \times 16 \times 16$ grid of boxes, let us focus on the bottommost row in the first layer, whose boxes have numbers 0–15. If we were working on a single machine, we would process this row during the x-sweep by directly calculating the result for box 0 and using the result of the previous box (adding and subtracting boxes as necessary) for boxes 1 through 15. In the distributed-memory setting, however, boxes 0–3 belong to node P_0, boxes 4–7 belong to node P_1, boxes 8–11 belong

[5] As mentioned earlier, we compute the source and target points at run time, so we store only the charge value corresponding to each source as input.

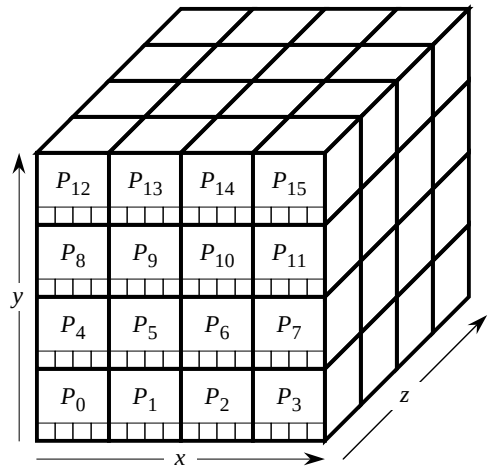


Figure 5.8: 64 nodes assumed to be arranged as a $4 \times 4 \times 4$ grid, with 64 boxes in each node. This figure shows the bottommost row of boxes in the first 16 nodes. For clarity, we have not numbered the boxes in each node. Node P_0 contains boxes 0–3, node P_1 contains boxes 4–7, node P_4 contains boxes 64–67, etc.

to node P_2, and boxes 12–15 belong to node P_3, as Figure 5.8 shows. Therefore, node P_0 would have to receive the S2W results of boxes 4–6 from node P_1 in order to complete its x-sweep of the first row. Node P_1, in turn, would have to wait to receive the x-sweep results of box 3 and the S2W results of boxes 0–3 from node P_0, and the S2W results of boxes 8–10 from node P_2, to complete its sweep of boxes 4–7. Similarly, for each row, nodes P_2 and P_3 would have to wait for nodes P_1 and P_2, respectively, to finish their sweeps. On the whole, we would end up serializing the process to some extent instead of using all the available resources.

In our implementation, therefore, on each node, in each round, the x-sweep stage *independently* sweeps each row of boxes. By independently, we mean that for each row, each node initiates a sweep starting from the first box that it owns in that row. In each round, this stage processes a row of boxes, following the same order of rows as its predecessor along pipeline p_0. Going back to our example in the previous paragraph, node P_0 works as before. Node P_1, however, receives the S2W results of boxes 1–3 (the last b boxes of node P_0; recall, we assumed that $b = 3$) and the S2W results of boxes 8–10 (the first b boxes of node P_2). Using data received from its neighboring nodes and the S2W results that it computes locally, node P_1 is ready to compute the sweep of box 4 directly, after which it can follow the usual scheme to sweep boxes 5–7. That is, node P_1 initiates a sweep starting at box 4, which is the first box that it owns in the row under consideration. Node P_2, similarly, receives the S2W results of boxes 5–7 and boxes 12–14 from nodes P_1 and P_3,

respectively, and initiates a sweep at box 8, and node P_3 receives the S2W results of boxes 9–11 from node P_2 and initiates a sweep at box 12. Nodes P_4–P_7 require similar communication patterns for each row of boxes that they own, and so on.

Referring again to the node arrangement shown in Figure 5.8, we can say that for the x-sweep, each node requires data from its immediate neighbors along the x-dimension. Because we assume that $\gamma_{local} \geq b$, each node requires data from at most two other nodes. Boundary nodes along the x-dimension require data from only one other node, whereas interior nodes require data from two other nodes. In Figure 5.8, nodes P_0, P_3, P_4, and P_7 are examples of boundary nodes, and nodes P_1, P_2, P_5, and P_6 are examples of interior nodes. Boundary nodes require the S2W results of b boxes from their left or right neighbors, and interior nodes require the S2W results of b boxes from each of their left and right neighbors in the x-direction. For the x-sweep, a node P_i, where $i = 0, \ldots, P-1$, is a boundary node if $i \bmod \rho = 0$ or $i \bmod \rho = \rho - 1$. We will refer to a boundary node P_i such that $i \bmod \rho = 0$ as a left boundary node and a boundary node P_i such that $i \bmod \rho = \rho - 1$ as a right boundary node.

Because we process one row per round, we require γ_{local}^2 rounds to complete the x-sweep. As mentioned earlier, the buffer along pipeline p_0 is a dummy buffer, which signals that the local S2W computation for the next row to be swept has finished, and the buffer along pipeline p_2 carries S2W results, received from neighboring nodes, that are required to sweep the next row.

On each node, the send and receive stages in pipelines p_1 and p_2 of Figure 5.7 take care of sending and receiving data, respectively, through interprocessor communication. From our previous discussion, we know that both these stages repeat for γ_{local}^2 rounds. In each round, the send stage of a left boundary node sends the S2W results of the last b boxes in the current row to its right neighbor, and a right boundary node sends the S2W results of the first b boxes in the current row to its left neighbor. An interior node sends the S2W results of the first and last b boxes in the current row to its immediate left and right neighbor, respectively. Now that we know what data each node sends in a round, we can easily infer what data each node receives in a round. In each round, the receive stage of a left boundary node receives the S2W results of its right neighbor's first b boxes in the current row and each right boundary node receives the S2W results of its left neighbor's last b boxes in the current row. An interior node receives the S2W results of its immediate right and left neighbor's first and last b boxes in the current row, respectively.

Given that the send stage always accepts a dummy buffer from the S2W stage, where does it send data for interprocessor communication from? Similar to the S2W stage, the send stage knows exactly which preallocated area of memory to send the data from. Indeed, the send stage accesses the most recent area of memory that the S2W stage used to write its output. Unlike the dummy buffers in pipelines p_0 and p_1, buffers flowing through pipeline p_2 carry useful data. The receive stage uses the buffer to receive S2W results from other nodes and passes the buffer along to the x-sweep stage, which uses the contents of the buffer to sweep the current row.

To aid performance in the send and receive stages, we distinguish between boundary and interior nodes, and we order the sends and receives in order to minimize communication time. We now explain how we ordered the sends and receives, assuming that the nodes are arranged as shown in Figure 5.8. A boundary node sends data to the single other node that requires its data. An even-numbered interior node first sends data to its right neighbor and then to its left neighbor, whereas an odd-numbered interior node first sends data to its left neighbor and then to its right neighbor. In the receive stage, all nodes follow the same order that we just described for the send stage.

The chronology of events that follows summarizes the working of the pipelines in Figure 5.7 in a single round:

1. The S2W stage accepts a dummy buffer from the source stage along pipeline p_1, reads its input from a predefined region of memory, computes the S2W results for the current row of boxes, and writes its output to another known area of memory. The S2W stage then conveys the dummy buffer along pipeline p_1 to signal the send stage to proceed. Finally, this stage accepts and immediately conveys a dummy buffer along pipeline p_2 to signal the x-sweep stage.

2. After the send stage accepts a dummy buffer along its pipeline, this stage sends data from the S2W stage's output region for this round of pipeline p_1, through interprocessor communication, to the required nodes. Hence, no copying of data occurs in this stage. The send stage conveys the dummy buffer before returning from its stage function.

3. The receive stage accepts an empty buffer from the source stage along pipeline p_2 and fills the buffer with S2W results received through interprocessor communication from other nodes, before conveying the buffer.

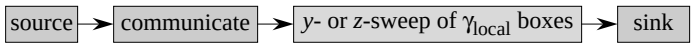


Figure 5.9: FG pipelines that each node executes to sweep along the y- and z-dimensions for the W2L step.

4. The x-sweep stage accepts a dummy buffer along pipeline p_0, which indicates that S2W stage's output region for this round of pipeline p_0 is available for reading. The buffer that this stage accepts along pipeline p_2 contains relevant data to sweep the current row of boxes. After sweeping the current row and writing the sweep results to a preallocated region of memory, which is disjoint from S2W stage's output regions, this stage conveys both buffers that it accepted along their respective pipelines.

Given that each node sends and receives data at exactly the same rate in each round,[6] one might wonder why we decided to use disjoint pipelines for interprocessor communication. Also, unlike dsort, where the amount of data and the set of nodes that a node sends data to and receives data from might differ from one round to another, here, each node knows exactly how much data to communicate and which other nodes to communicate with, in each round. The pipelines of Figure 5.7 represent our existing design for implementing the S2W and x-sweep phases of distributed-memory FGT. In retrospect, we could have collapsed the pipelines of Figure 5.7 into a single, linear pipeline. We realized this possibility after having performed the experiments reported in Section 5.7. We were sufficiently satisfied with the experimental results that we felt it unnecessary to reimplement the pipelines. To avoid confusion, we refrain from describing this alternate, single-pipeline approach.

5.6.4 Completing the W2L step and the L2T step

Next, we carry out the y- and z-sweeps in two phases on each node to complete the W2L step. For both of the remaining sweeps, we implemented the FG pipelines shown in Figure 5.9. Buffers flowing through this pipeline carry useful data. The first user-defined stage in the pipeline participates in interprocessor communication, and its successor stage processes either a column of boxes (during the y-sweep) or a row of boxes along the z-dimension (during the z-sweep).

During the y-sweep, each node processes a column of boxes in each round, for a total of γ_{local}^2 rounds. Similar to the x-sweep, each node communicates data required for the y-sweep with other

[6] Each node either sends and receives $b(2p+1)^3$ complex doubles, or it sends and receives $2b(2p+1)^3$ complex doubles.

nodes and again, because we assume that $\gamma_{local} \geq b$, each node communicates data with at most two other nodes. Also as before, for each column, each node initiates a sweep for this column starting at the bottommost box, after the node has received the necessary data to complete the y-sweep for this column. For the y-sweep, a node P_i such that $\lfloor i/\rho \rfloor \bmod \rho = 0$ or $\lfloor i/\rho \rfloor \bmod \rho = \rho - 1$ is designated a boundary node and other nodes are interior nodes. In Figure 5.8, nodes P_0–P_3 and P_{12}–P_{15} are examples of boundary nodes and nodes P_4–P_{11} are examples of interior nodes for the y-sweep.

The communicate stage in Figure 5.9 uses the MPI routine `MPI_Sendrecv_replace` to send and receive data through interprocessor communication. The `MPI_Sendrecv_replace` routine uses the same region of memory to send data from and receive data into, though the node that it sends data to can differ from the node that it receives data from. Because each node sends and receives data at the same rate in each round,[7] we decided to use `MPI_Sendrecv_replace` in the communicate stage in the pipeline of Figure 5.9. To satisfy the specifications of the `MPI_Sendrecv_replace` routine, each node copies the relevant data to be sent in a round into the pipeline buffer, before invoking `MPI_Sendrecv_replace`. Here again, we order the communication so that a boundary node immediately communicates with its top or bottom neighbor, and an interior node communicates first with its top neighbor and then its bottom neighbor, or vice-versa. Of course, during the y-sweep, each node uses the results of the x-sweep that it calculated earlier for its set of boxes, along with the data that it receives from its neighboring nodes along the y-dimension.

As an aside, we briefly note that we could have collapsed the send and receive stages in the pipelines of Figure 5.7 into a single communicate stage that would have worked similar to the communicate stage that we just described. Hence, we could have implemented the S2W and the x-sweep phases using a single, linear pipeline.

After completing the y-sweep, we re-execute the pipelines of Figure 5.9 to complete the z-sweep and hence the W2L step. During the z-sweep, each node processes a row of boxes along the z-dimension in each round, for a total of γ_{local}^2 rounds. We define a node P_i, for $i = 0, \ldots, P-1$, such that $\lfloor i/\rho^2 \rfloor = 0$ or $\lfloor i/\rho^2 \rfloor = \rho - 1$ to be a boundary node, and we categorize other nodes as

[7] Similar to the x-sweep, each node sends and receives either $b(2p+1)^3$ complex double values or $2b(2p+1)^3$ complex double values.

interior nodes. In Figure 5.8, nodes P_0–P_{15} and nodes P_{48}–P_{63} (hidden) are boundary nodes, and nodes P_{16}–P_{47} are interior nodes. In the z-sweep, we use the results from the y-sweep, and we order the communication similar to the earlier sweeps.

Thus, executing the pipelines of Figure 5.7, followed by two executions of the pipelines of Figure 5.9 completes the first two steps of the fast Gauss transform in distributed memory.

All computations required for the last step, the L2T step, are entirely local to a node and, therefore, we use OpenMP's `parallel for` loop (see lines 4–5 of the pseudocode for the shared-memory implementation) to approximate the Gaussian of equation (5.1) for the targets in multiple boxes in parallel.

5.6.5 Error checking

Given that the fast Gauss transform computes approximate values, we should take the time to determine its numeric accuracy. Ideally, we would like to calculate the Gaussian of equation (5.1) directly for each target, evaluate the error as the difference of the actual value with the respective FGT value for the target, compute the root mean square error over all targets, and ensure that the root mean square error is at most ϵ, which is the required precision. Computing the actual value for all targets is prohibitive, however.[8] Therefore, we compute the sampled maximum relative error,[9] defined in the pseudocode below, using a sample of target points and compare it against ϵ.

SAMPLED-MAX-RELATIVE-ERROR
1 let max_{FGT} = maximum of all FGT values
2 let S = a sample of target points
3 **for** each $v \in S$
4 compute v_{direct} = Gauss transform of v calculated directly using equation (5.1)
5 let v_{FGT} = FGT value for v
6 compute the relative error as $|v_{direct} - v_{FGT}|/max_{FGT}$
7 sampled maximum relative error = maximum of the relative errors computed for sample S

5.7 Experimental results

In this section, we present some performance results of our FGT implementation. As mentioned earlier, we have implemented the fast Gauss transform in shared memory using OpenMP and in

[8] Recall that improving the O(mn) running time was the basis for FGT.
[9] We were in contact with Shravan K. Veerapaneni [44, 47], who suggested this method.

distributed-memory using FG, MPI and OpenMP. Our shared-memory and distributed-memory versions are both implemented in C++. We compare our implementation against a distributed-memory implementation in C++ by Sampath, Sundar, and Veerpaneni [44], which uses PETSc 3.0 [4, 5] for MPI. We will refer to this other implementation as SSV-FGT.

We ran both implementations on a cluster with many nodes, of which we used either one or eight nodes. Each of the nodes that we used has two dual-core, 2.3-GHz AMD Opteron 280 processors, and 8 GB of RAM. Each node runs Linux version 2.6.33.3 and the nodes are connected using gigabit ethernet. We use MPICH2 [42] for MPI because it is a thread-safe implementation of MPI. We compiled our shared-memory and distributed-memory implementations using g++ version 4.1.2 and mpicxx, respectively, at optimization level O3. We requested the authors of SSV-FGT for their code and compiled it using PETSc 3.0, as suggested by them; here again, we set the compiler optimization level to O3.

In our experiments, we assume that $d = 3$, that all points lie within the unit cube $[0, 1]^3$, and that the sources and targets coincide. We also assume that within each box, the points lie on a regular grid. As mentioned earlier, we store only the charge values associated with the source points, and we generate the source/target points at run time. Similar to our implementation, SSV-FGT, too, seems to generate input at run time. To the best of our knowledge, the SSV-FGT implementation that we compared against generates data on a regular grid in each box. For fairness, however, in our implementation, we generate experimental results with points lying on a regular grid in each box and with points not assumed to lie on a regular grid,[10] using separate executions of our program.

On a single node, we generate 29.2 million source/target points and set $\delta = 1/16$, for a total of 64 boxes arranged as a $4 \times 4 \times 4$ grid. On eight nodes, we generate 233.6 million source/target points distributed equally across the nodes and set $\delta = 1/64$, for a total of 512 boxes. Each node, therefore, owns a $4 \times 4 \times 4$ subgrid of boxes, for a total of 64 boxes. Note that for $P = 8$, we generate 8 times as many points as for $P = 1$, so that we use the same number of points per node. In both cases, we set $\epsilon = 10^{-6}$, for which the required values of L, b, and p are 7, 4, and 10, respectively (see Table 2.1 in the paper by Spivak, Veerapaneni, and Greengard [47]). We provided suitable command-line parameters to SSV-FGT to match our input sizes and the FGT parameter

[10] To satisfy this assumption, we repeatedly calculate sine and cosine values in the S2W and L2T steps instead of applying our performance enhancements. That is, we did not actually generate points on a non-regular grid.

	$P = 1$		$P = 8$	
	29.2 million points		233.6 million points	
implementation	regular grid	non-regular grid	regular grid	non-regular grid
PN-FGT	12.63	21.12	5.68	8.97
SSV-FGT	101.90	101.90	117.03	117.03

Table 5.1: Experimental results for our implementation, nicknamed PN-FGT, and SSV-FGT. All results are presented in seconds and each result represents the average of three runs.

values in both the 1-node and the 8-node cases.

Table 5.1 summarizes the results of our experiments. In the table, we nickname our implementation as PN-FGT. All results are presented in seconds; each result is the average of three runs, where the running times differed only slightly within each group of three. Because we are not entirely sure whether SSV-FGT generates inputs on a regular or a non-regular grid within each box, we show the same running time for both types of grids in each case for SSV-FGT. In our implementation, when $P = 1$, we use only one core out of the available cores on a node, similar to SSV-FGT. That is, the results for $P = 1$ effectively give us the serial running time for both implementations. When $P = 8$, we use all available cores on a node for maximum parallelism; we are not sure whether SSV-FGT uses all available cores on a node. When $P = 1$, we get a maximum relative error of 10^{-8} based on the first 100 source/target points, and when $P = 8$, we get a maximum relative error of 0.0000025 based on the first 100 source/target points; we don't know these numbers for SSV-FGT.

As the results in Table 5.1 show, our FGT implementation ran considerably faster than SSV-FGT in all cases. Our implementation was between 4.8–20.6 times faster than SSV-FGT. We attribute our performance to the judicious use of available parallelism: both shared-memory and distributed-memory parallelism. In shared-memory, we use OpenMP's `parallel for` regions to compute the results for multiple boxes in parallel in the S2W and L2T phases.[11] In addition to distributing the computation over multiple nodes in our distributed-memory implementation, FG pipelines help us to overlap interprocessor communication with computation, and we use OpenMP's `parallel for` regions for in-core parallelism.

For completeness, we must also ensure that our implementation scales well. To check for scal-

[11] Our code has these parallel regions though we restrict ourselves to use only one core in our experiments for a fair comparison with SSV-FGT.

implementation	regular grid	non-regular grid
$P = 1$, using 4 cores	4.46	6.30
$P = 8$, using 4 cores on each node	5.68	8.97

Table 5.2: Scalability results for our implementation. All results are presented in seconds and each result represents the average of three runs.

ability, we ran our shared-memory implementation by allowing it to access all available cores on a single node and compared these results with our results for $P = 8$. As before, we use the same number of points per node, i.e., we generate 29.2 million source/target points on a single node and 233.6 million source/target points on eight nodes. Table 5.2 shows the results of our comparison. As we can see from the table, our code scales reasonably well, though we admit that just eight nodes are not enough to test for scalability. We were unable to test our code further because of hardware limitations.

5.8 Conclusions

In this chapter, we saw how to approximate the Gaussian of equation (5.1) using Hermite and plane-wave expansions at m target locations, as well as our implementation of the plane-wave version of the fast Gauss transform. Although we did not store any data on disk for this application, we still had to think about mitigating the effects of interprocessor communication on the performance of our implementation, for which FG pipelines proved useful. Furthermore, we were able to use FG stages as signaling mechanisms for the first time here. This application also provided us with ample opportunity for in-core parallelism for which we used OpenMP. In addition to exploiting the available parallelism, we analyzed the structure of the computation and identified ways to reuse costly computations to improve performance.

The experimental results presented in Section 5.7 are highly encouraging, both in terms of performance and scalability. That being said, our implementation does make a few assumptions. In particular, we assume that all source/target points within a box lie on a regular grid. In general, if the source and target points are randomly distributed within the unit cube $[0, 1]^3$, some boxes are likely to contain more points than the others. Going through all the steps of the FGT for lightly-populated boxes is costly. Hence, for general point distributions, we would want to segregate the

densely-populated boxes from the lightly populated ones, and follow a different approach in each case. Sampath, Sundar, and Veerapaneni [44] address the parallel implementation of this more general case. Our experiments with a regular grid inspire us to try the general case in future. We believe that FG pipelines will prove useful in the general case, too. As in the regular-grid case, understanding the underlying computation and the sequence of data exchanges will be vital to designing efficient FG pipelines. FG, in turn, will provide multithreaded parallelism to help overlap interprocessor communication with computation, in addition to allowing access to other sources of parallelism in its stage functions. As mentioned earlier, we were able to run our experiments on at most eight nodes, which served as a good starting point. As a next step, we should endeavor to run our implementation on more nodes to be able to better assess the performance and scalability of our implementation. In Section 5.6.3, we mentioned the possibility of collapsing the pipelines of Figure 5.7 into a single, linear pipeline. What effect this alternate design has on performance might be worth looking into.

On the whole, the favorable results from our experiments for the fast Gauss transform encourage us to look for more applications in the field of scientific computing.

Chapter 6

Related Work

In this chapter, we compare FG with related libraries such as UNIX pipes, StreamIt, TPIE, and TBB, and we discuss the similarities and differences of each of these libraries with respect to FG. We also compare FG's design with the concept of dataflow programming.

6.1 UNIX Pipes

FG shares its conceptual design of a pipeline acting on data streams with UNIX pipes [3]. Because pipes transfer data between processes, not threads, pipes are a more heavyweight mechanism than the buffer queues that sit between FG stages. Furthermore, because low-level system interactions in UNIX occur through a higher-level file-like abstraction, UNIX pipes interact through the file-system interface. FG and UNIX pipes both make use of shared memory, but whereas it is at the thread level in FG, it is at the process level in pipes. Although it is possible for a process to read from or write to multiple pipes and thus emulate intersecting-pipelines-like flow of data, the underlying programming effort is considerable. We have to invoke the `pipe` system call multiple times as required, be careful not to accidentally shut off some direction of a pipe in a process, avoid deadlocks, and close file descriptors and release memory associated with all pipes that were created. Moreover, we are likely to run out of system resources in situations where we use FG's virtual pipelines.

6.2 StreamIt

The StreamIt [51, 52] project from MIT is a Java-like language and compiler for stream programs. Stream-based applications are found in the embedded domain such as cell phones and software routers. StreamIt and FG have much in common. StreamIt defines a *filter* to be the basic unit of computation, with the fine-grained execution steps described in a filter's *work* method; a filter and its work function resemble FG's stage and stage function, respectively. Just as a stage operates on a fixed-size buffer, a filter has static input and output sizes, known as pop and push rates, respectively, in StreamIt terminology.

StreamIt allows filters to be joined together to form a Pipeline, SplitJoin, and FeedbackLoop. The simple linear pipeline constructs in StreamIt and FG are fundamentally similar. A SplitJoin in StreamIt diverges at a *splitter* and converges at a *joiner*; a splitter can be Duplicate, RoundRobin, or Null. A Duplicate splitter sends a copy of each data item to the various output streams, a RoundRobin splitter sends data in a round-robin manner, and a Null splitter sends out no data at all (useful if its successor filters expect no input). A joiner can be either RoundRobin or Null. FG's fork-join construct is similar in flavor, except that FG allows only the FG-equivalent of a RoundRobin splitter but a joiner in FG can be either RoundRobin or first-come first-served (FCFS). As in FG, the FeedbackLoop construct in StreamIt introduces cycles in a stream graph. FG and StreamIt share the common goal of making programs easier to write and faster to execute.

We now highlight some of the major differences between StreamIt and FG. Unlike FG, which is a library, StreamIt is a language with its own compiler. The StreamIt language specification for version 2.1 [49] suggests that StreamIt now supports dynamic I/O rates for filters, that is, a filter can consume and emit data items at a varying rate; buffer sizes in FG remain fixed over the entire execution of the pipeline. FG allows disjoint and intersecting pipeline structures, which are absent from StreamIt.

StreamIt seems to assume an infinite sequence of data items, whereas FG has an explicit notion of shutting down a pipeline. When the number of rounds are not known in advance and when buffers might flow nonlinearly through a pipeline, it can be difficult to determine when to shut down the pipeline correctly.

6.3 TPIE

Templated Parallel I/O Environment, or TPIE [1] is a library that provides templated interfaces for efficient implementations of external-memory algorithms. TPIE is implemented as a set of templated classes and functions in C++. Unlike FG, users never make explicit calls to I/O functions in TPIE; its Block Transfer Engine takes care of transferring data to and from the disk. The user is provided access to a data stream that can be read from and written to sequentially. Although the user can choose from three Block Transfer Engines (one each for C `stdio`, `ufs` I/O, and I/O using `mmap`) depending on her desired method for performing I/O, TPIE streams do not provide the flexibility of non-sequential data access. In FG, however, because the user has full control over how I/O is performed, she can access data sequentially or otherwise.

TPIE provides templates for several computation patterns such as scanning, permutation, and merging in the form of *operation management objects*. The exact names of methods in an operation management object and their required functionality are specified by TPIE, and the user is required to supply the application-specific code. Usually, each operation management object requires at least two methods: `initialize()` and `operate()`. The user defines the exact computation that she wants for the pattern in the `operate()` method, just as she defines the exact operation of a stage in an FG stage function. The `operate()` method can request access to multiple streams if need be, similar to how the user can accept buffers from multiple pipelines in an intersecting stage in FG. Unlike FG, where stage functions operate on large data buffers, the `operate()` method of an operation management object seems to function at the granularity of a single data element per input stream. In FG terminology, this level of granularity is equivalent to calling a stage function for every data element in the input. The user is also responsible for informing TPIE of the end of the computation by returning a special TPIE-defined constant from the `operate()` method, which is similar to setting the caboose flag on the fly in an FG pipeline.

It is not clear from the documentation whether the user can perform interprocessor communication in an `operate()` method. TPIE, therefore, seems to be designed for executing external-memory algorithms on a single machine, and its documentation suggests that the Block Transfer Engines only support streams stored on a single disk.

6.4 Dataflow Programming

In the dataflow execution model [32], a program is represented as a directed graph where the nodes are primitive instructions such as arithmetic operations, and edges between the nodes represent data dependencies between the instructions. Data flows as tokens along the edges, which behave as unbounded FIFO queues. A node's incoming edges are called its *input arcs* and its outgoing edges are called *output arcs*. Whenever data is available on all the input arcs of a node, it becomes *fireable* and is executed at some later time. When a fireable node executes, it removes data from each of its input arcs, performs its operation, and places data on one or more of its output arcs. That is, an operation becomes ready to execute as soon as all its required input is available. Because multiple instructions might be executed in parallel, a dataflow program presents the potential for a high degree of parallelism. In contrast, an instruction in the von Neumann execution model executes only when the program counter reaches it, irrespective of whether the instruction could have been executed earlier.

Dataflow principles automatically allow for *pipelined* dataflow if the computation is to be performed on more than one dataset. Note, however, that a subset of nodes in a dataflow graph that form a chain[1] can never be executed in parallel for a particular wave of data, and operating such subgraphs at the fine granularity of the pure dataflow model is inefficient. Dataflow programming, therefore, evolved to define *large-grain dataflow* [54] wherein a fine-grained dataflow graph is analyzed to identify subgraphs that form a chain, whose instructions are then grouped to execute as a sequential process. These coarse-grained nodes are termed *macroactors*. Even in large-grain dataflow, all parts of the dataflow graph, including macroactors, execute under the rules of the dataflow execution model. In any of its forms, the dataflow model does not address high-latency operations.

In principle, FG shares some features with dataflow programs. An FG pipeline can be seen as a directed graph with data buffers flowing from one stage to another as specified by the user. The thread corresponding to an FG stage becomes ready to execute when a buffer is available in its incoming queue. Similar to a node in a dataflow program, an FG stage accepts a buffer, executes its stage function, and conveys the buffer to its outgoing queue. Conceptually, we, too, "overload" arcs in our pipeline illustrations to represent both the direction of data flow and the buffer queues that

[1] By a chain, we mean that nodes are linked one after another such that the output of one node is the input for the next.

sit between stages. Also, multiple stages in an FG pipeline might run in parallel, depending on the available resources.

In practice, however, FG and dataflow programs share little in common. Dataflow graphs are said to be the "machine language" of dataflow programs, whereas FG's pipeline "graphs" are meant purely for illustrative purposes. In the pure dataflow model, nodes represent primitive instructions, whereas stage functions in FG usually span multiple lines of code in C++ and, therefore, involve hundreds or thousands of instructions. At first glance, an FG pipeline seems to resemble a dataflow graph in the large-grain dataflow model. Note, however, that in the large-grain dataflow model, a linear pipeline in FG would be executed as one big serial stage whose stage function is formed by combining all the stage functions of the original pipeline, resulting in zero parallelism. As we know, all FG programs are multithreaded and have the potential of running in parallel. FG's pipelines are coarse-grained in another sense, too: FG stages operate on large buffers of data, instead of single data tokens. The coarse granularity of data "tokens" in FG is important for efficiently implementing programs that run on massive datasets. FG can be seen as a combination of dataflow ideas and multithreading features available in the von Neumann world.

6.5 Threading Building Blocks (TBB)

Intel's Threading Building Blocks [53] is a C++ template library that supports many parallel-programming constructs, including software pipelines. Similar to FG, TBB relieves the programmer from having to parallelize her code using native threads and is offered as a library; thus, it does not require learning a new language or using a new compiler. Just as FG encourages a programmer to design an FG pipeline based on the high-latency operations in her program, TBB encourages a programmer to express the required computation using a number of tasks. The TBB runtime system takes care of efficiently mapping tasks to threads. Instead of statically allocating work to threads, the Threading Building Blocks library recursively splits the user's tasks until the right number of parallel tasks are reached. Breaking down the problem recursively also suits the library's task-stealing approach for load-balancing tasks among threads and enables the library to scale the parallelism based on the number of cores available in a machine.

The Threading Building Blocks library offers a number of parallel-programming constructs

such as `parallel_for`, `parallel_scan`, and `parallel_reduce` and some concurrent containers such as `concurrent_queue`, `concurrent_vector`, and `concurrent_hashmap`. Additionally, TBB provides a pipeline template for programs that can be modeled as a linear sequence of stages. Each stage in a TBB pipeline is written as a C++ class that inherits from TBB's `filter` class; an implementation of `operator ()` within the class serves as the corresponding stage function. Public methods in TBB's `pipeline` class allow the user to set up and run the pipeline. Analogous to FG buffers, *tokens* flow through a TBB pipeline, and as with FG, the user is required to set the maximum number of tokens that can be in flight in a TBB pipeline. In TBB, each stage can be designated as serial or parallel; a serial stage processes tokens one after the other, in order, whereas a parallel stage might process multiple tokens concurrently or out of order. TBB requires the first stage in the pipeline to also manage buffers; effectively, the first stage acts like FG's source stage but the code to manage buffers comes from the user and is part of the stage function. That is, unlike FG, the user shares some of the burden of buffer management in TBB. The first stage is also responsible for deciding when the pipeline has finished its computation; the stage returns `NULL` instead of a valid buffer to indicate the equivalent of FG's caboose round. It is not clear from the TBB documentation as to whether a middle stage in a pipeline can inform TBB's runtime system that the pipeline has finished its computation. Tokens arrive as stage function parameters and are sent out of the stage function as return values; TBB internally ensures that tokens are processed in order at the serial stages. TBB's design allows disjoint linear pipelines to run concurrently, but TBB requires that each pipeline be started from its own thread. TBB, however, does not support non-linear pipeline constructs similar to FG's fork-join or intersecting pipelines. That is, it would not be possible to implement dsort pass 2 in TBB using the pipeline structures shown in Figure 3.4.

Unlike FG, TBB is not designed for programs that involve high-latency operations along with computation operations; TBB's task scheduler works best on algorithms composed of non-blocking tasks. Although tasks offer scalable parallelism, a blocked task can neither be split further nor be scheduled on a processor; TBB documentation suggests using full-blown threads for tasks that block. That is, FG's design is suited for the kinds of programs we target.

6.6 MapReduce

MapReduce, introduced by Google, is a programming model inspired by the *map* and *reduce* primitives present in Lisp and other functional languages. The main focus of Google's MapReduce is to apply the paradigm to extremely large datasets, in the order of terabytes, running on thousands of nodes on a cluster. In a paper introducing the model [19], Dean and Ghemawat bring to light several interesting problems, such as distributed grep, count of URL access frequency, and reverse web-link graph, that can be effectively expressed using MapReduce. In order to realize these problems on inputs spanning terabytes, the computations need to be distributed and carried out in parallel across hundreds or thousands of machines. The conceptual simplicity of many such problems is obscured by the much greater, and more complicated, code required to deal with the issues related to distributing the data and parallelizing the computation. The peripheral code, although tedious, repetitive, and usually unrelated to the underlying problem, is necessary for a practical implementation. MapReduce was, therefore, designed as an abstraction to allow users to express their computations in a simple, uniform manner, without having to worry about the details of parallelization. Recall from Section 2.1 that similar observations regarding out-of-core programs led to the genesis of FG. That is, both MapReduce and FG share the goal of wanting to make it simpler for users to write efficient parallel programs for large datasets. The users of MapReduce express the computation using two functions: *Map* and *Reduce*. These functions are analogous to FG's stage functions, except that, in FG, the user provides a stage function for each unique stage that she would like FG to spawn.

For each input pair, the `Map` function outputs a list of *intermediate* key-value pairs. The `Reduce` function merges all values that belong to the same key k to produce a smaller set of values (typically zero or one output values). Similar to the input files, the output files are also distributed on the nodes of a cluster; the user is required to specify the number R of output files that should be written by MapReduce. Because each reduce task outputs one file, R is also the number of reduce tasks that work on the results of the map function. In addition to the map and reduce functions, the library allows users to specify a *partition* function for intermediate keys to override the default partition method that the library provides for ensuring well-balanced partitions.

The input files, stored on the local disks of the nodes of a cluster, are managed by the Google File System (GFS) [25] which splits the files into 64-MB blocks and stores a copy of each block on

three different nodes. MapReduce invokes multiple copies of the program on a cluster. One of the copies becomes the master; the remaining copies are designated as workers and are assigned map or reduce tasks by the master. In order to save network bandwidth, the master tries to schedule a map task on a node that has a replica of the corresponding input. MapReduce re-executes halted tasks resulting from node failures, which are common in large clusters. If a worker node fails, the master reassigns the failed worker's task to a new node; if the master fails, the program is aborted. Also, because each reduce worker sorts its data by the intermediate keys before starting its work, the library guarantees a sorted output file within each partition.

While enumerating the many applications of MapReduce in the introductory paragraph earlier, we deliberately left out one important application—distribution sort. In an implementation of distribution sort using MapReduce, the map function emits a <key, record> pair and the reduce function emits all pairs unchanged. Clearly, this implementation seems much simpler than the FG implementation that we described in Chapter 3. Recall, however, that the library ensures sorted output for all applications; therefore, it is possible that a dsort-like algorithm is built into the library. The MapReduce paper [19] mentions that an external sort is used when the intermediate data output by the map function is too big to fit in main memory, but it does not provide any details about the underlying sorting mechanism.

Just as FG is a framework for pipeline-structured programs, MapReduce is a programming model for problems that can express their computations within the map and reduce functions. We listed several applications of MapReduce within our limited scope, and the literature on FG presents a number of applications that are implementable using software pipelines. Whereas FG spawns as many threads on each node as the number of stages in the pipeline, this information is not apparent in the MapReduce literature; we conjecture that MapReduce handles one task per node. MapReduce also provides support for recovering from faults and has a designated master that dynamically allocates tasks to workers; FG relies on the operating system's abilities to dynamically schedule threads. Both FG and MapReduce provide a simple interface for applications that can fit within their respective frameworks and make it simpler for users to parallelize their applications in this era of multicore machines.

6.7 STXXL

The Standard Template Library for Extra Large Data Sets (STXXL) was introduced by Dementiev [20] to allow easy implementation of algorithms based on the Parallel Disk Model [57]. STXXL's interface resembles that of the C++ Standard Template Library (STL) [50]. The STXXL library provides classes for data structures such as vectors, stacks, and queues, and for algorithms such as sorting, including when data reside on parallel disks on a single node. The library's AIO layer provides I/O-related operations and abstracts away details of how asynchronous I/O is performed. Recently, the authors of STXXL added pipelining capabilities to the library [22] to overlap I/O and computation, and to reduce the number of I/O operations.

In STXXL, a pipelined computation is represented as a directed, acyclic *flow graph*, where the nodes of the flow graph represent data-processing components and its edges indicate the direction of data flow. The nodes of a flow graph are analogous to FG stages and edges play a similar role in FG. Like an FG pipeline, an STXXL flow graph is set up in source code, i.e., there is no graphical interface for setting up a pipeline.

STXXL defines three types of nodes in a flow graph: a *scanning* node, a *sorting* node, and a *file* node. In traditional STXXL pipelining, each node processes *elements* of possibly different types and numbers, and the type dependencies are resolved at compile time. Scanning nodes process one element after the other, do all work in order, and their functionality is defined by the programmer. Sorting nodes sort elements with respect to some sorting criterion, and file nodes read elements from disk to feed their successors or consume elements from their predecessors and write them to disk. What role the programmer plays in these two kinds of nodes is not clear from the literature. Nodes also seem to require definitions of `operator*()` and `operator++()`, but here again we are not sure as to who implements these operators. As we know, FG does not distinguish between stage functions; an FG user is free to implement each stage function per her requirement. We do acknowledge that scanning, sorting, and file nodes cover most operations in an external-memory computation on a single machine.

Each flow graph is required to have a *primary sink*, which is usually a file node, and pipeline execution is triggered by the primary sink when it invokes the `materialize()` function. As in an FG pipeline, STXXL requires that all nodes have a path to the primary sink. In FG, however, it is

the source stage of a pipeline that triggers the computation by emitting an empty buffer to the first user-defined stage in the pipeline.

A sorting node in an STXXL flow graph seems to be internally implemented using a three-node sequence: a run-creator scanning node followed by a file node to store sorted runs on disk, and a merging node to merge the sorted runs. Though each sorting node internally involves further complexity as we just described, it has a simple representation in an STXXL flow graph. The implementation of external-memory sorting in STXXL uses MCSTL [46] to parallelize the in-memory computations; MCSTL seems to inherently use OpenMP's parallel routines. For the task parallelism that pipelining requires, STXXL relies on OS threading mechanisms, and POSIX threads in particular, similar to FG.

Because STXXL assumes responsibility for providing some basic data structures and algorithms, the glue code for a pipeline involving all three types of nodes looks fairly short and clean (see Appendix A in [6]), but it is not intuitive for a novice STL programmer. We admit that a similar pipeline in FG would require more coding effort, but in our opinion, the APIs that FG provides for constructing a pipeline are much more intuitive. Furthermore, if FG were to internally support some basic algorithms in its next version, the code for setting up an FG pipeline would shrink, too.

Later, the STXXL pipelining framework was augmented with *asynchronous* nodes [6], which process data by spawning a worker thread to communicate data between their predecessors and successors. Each asynchronous node requires two element buffers: a producer buffer to absorb elements from the predecessor and a consumer buffer to serve the successor. Routines executing on behalf of nodes may block when no data is available. When the producer buffer is full and the consumer buffer is empty, the two buffers are swapped. These buffers work using OS-supported synchronization mechanisms. The buffers associated with asynchronous nodes were introduced to amortize the cost of thread synchronization because synchronizing for every single element would be far too costly. The buffer size can be tuned as required, and introducing asynchronous nodes in an existing implementation requires extra coding effort. From the above description, we best understand asynchronous nodes as being akin to the buffer queues that sit between FG stages. Both FG buffers and the buffers associated with asynchronous nodes provide coarse-grained data parallelism. Whether the primary sink of an STXXL flow graph recycles buffers is unclear.

On the whole, the fundamental designs of FG and STXXL pipelines seem to have more similar-

ities than differences. The paper by Beckmann, Dementiev, and Singler [6] shows an example of a distribute-collect flow graph that resembles FG's intersecting pipelines, though support for disjoint pipelines seems to be absent from STXXL. We conjecture that because of these similarities, the FG-based and STXXL-based implementations of external-memory suffix array ran comparably, as we saw in Chapter 4. To the best of our knowledge, STXXL does not support distributed-memory computations that involve interprocessor communication. A stage that performs interprocessor communication cannot be categorized as a scanning, sorting, or a file node, which further validates our speculation.

Chapter 7

Conclusions

In this thesis, we explored FG's software-pipeline model for parallel computing. In particular, the multithreaded platform that FG provides is useful for overlapping high-latency operations with other operations in out-of-core and distributed-memory applications. In this thesis, we saw how to use FG to tackle latency with the help of three program instances: an out-of-core, distribution-based sorting program; an external-memory suffix-array program; and a scientific-computing application called the fast Gauss transform (FGT). In the latter two applications, we tried to utilize all the cores available in a machine by combining FG's multithreading feature with libraries such as OpenMP that help with in-core parallelism. In all three instances, we achieved good performance results, which we attribute to well-designed FG pipelines and programming techniques that leveraged distributed-memory and shared-memory parallelism, combined with FG's quality implementation. This thesis shows that we can use FG to model applications from a variety of disciplines and implement them efficiently. As always, more work can be done, which we discuss next.

Both in dsort and in FGT, we used a distributed-memory cluster for our experiments. In both these programs, we were limited by the available hardware, however. In dsort, we were able to sort only 64 GB of data distributed across 16 nodes of a cluster. Considering the hardware that is available today, 64 GB of data is relatively small. Similarly, in FGT, we were able to test our scalability results only on 8 nodes of a cluster, which again is not enough. FG and our implementations are capable of handling bigger datasets in both these cases, so we should endeavor to look for hardware on which we can test these programs better.

As we discussed in the last section of Chapter 4, the FG pipelines that we designed for im-

plementing external-memory suffix arrays made us realize that auxiliary buffers available in FG version 1.4 might be redundant, after all. Usually, a stage that cannot perform its work in-place requires auxiliary buffers. We propose that whenever a stage requires an auxiliary buffer, we make it an intersecting stage.[1] Along the extra pipeline, only the source stage should precede this stage and the sink stage should succeed this stage. A pipeline buffer along the extra pipeline can then serve as an auxiliary buffer for this stage. Using the proposed design, we can allow auxiliary buffers to be of a different size than the pipeline buffer, which is not possible in the existing FG implementation. In this design, each stage that requires an auxiliary buffer will add an extra pipeline, each of which will require its own source and sink stages, which amounts to as many extra threads. As we saw in Section 3.2.3, we could soon overwhelm the available system resources. In order to stay within the system thread-limit, we could probably collapse all the source stages and all the sink stages, such that there is only one source stage and one sink stage for all the pipelines. We could take the lead from virtual pipelines to implement the single-source, single-sink idea. Recall that the current FG implementation provides a swap method for auxiliary and pipeline thumbnails, which swaps the buffers associated with the thumbnails so that the auxiliary buffer becomes the pipeline buffer and vice-versa. The main advantage of the swap method is that it helps us avoid having to copy data from the auxiliary buffer to the pipeline buffer.[2] Thanks to FG's thumbnail design, we should be able to implement the swap method now just as simply as before, i.e., by swapping only the buffer pointers, keeping the rest of the two pipeline thumbnails the same.

Another FG feature that might be worth exploring is whether FG should provide built-in support for routine operations such as reading from and writing to disk and sorting. Aided with such a functionality, the user will be saved from writing a function for each user-defined stage in the pipeline. To implement the disk-based operations, for example, the user might tell FG the buffer size and the file name to read from or write to, and FG could execute the command for reading or writing. The sorting functionality might be more tricky, but will perhaps be doable with the help of a user-provided comparison function and C++ template programming.

In the last section of Chapter 5, we identified some directions for future work, which we reiterate here. The FGT implementation that we presented in this thesis assumes that the source/target

[1] If the said stage is already an intersecting stage, we add another pipeline.

[2] Recall that because a stage uses the auxiliary buffer as temporary space, the useful results of the computation might be in the auxiliary buffer.

points within each box lie on a regular grid, which is surely not the most general case. Our FGT implementation should serve as a good starting point for a future implementation that would handle the general point distribution. We might even identify some hidden FG facet that we haven't come across yet; for example, while designing pipelines for the FGT, we realized that FG stages could also serve as signaling mechanisms. Another small but interesting experiment would be to try the single, linear pipeline design for the S2W and x-sweep phases of distributed-memory FGT and see how the performance of this alternate design compares with the existing results.

This thesis shows that we can use FG to model applications from different computing disciplines and implement them efficiently. FG continues to deliver its promise of making multithreaded programming easier for users, without compromising on performance. Our work has helped to extend FG in meaningful ways, and it provides some suggestions for the next version of FG (version 2.0), which is under way. The performance results from this thesis can serve as initial benchmarks for the same programs implemented using FG version 2.0; additionally, we have outlined some application-specific future work.

To conclude, we reiterate the contributions of this thesis:

1. While designing an out-of-core, distribution-based sorting program using FG (nicknamed "dsort"), we identified ways to advance FG from supporting just single, linear pipelines to multiple disjoint pipelines and multiple pipelines that intersect at a common stage. Using these new pipeline structures, we were able to implement dsort efficiently, despite its disadvantages of having dynamic I/O and communication patterns.

2. Our implementation of out-of-core sorting in a shared-memory setting using FG is faster by 9.6%–16.3% (approximately) compared with an STXXL-based implementation.

3. Our implementation of external-memory suffix arrays exercised FG's intersecting pipelines in ways that had never been attempted before. The complex FG pipeline structures that we designed for this algorithm also performed well. For the first time, we implemented a recursive algorithm using FG, and we used OpenMP along with FG to fully utilize all the cores of a multicore machine.

4. We have implemented the fast Gauss transform in a distributed-memory setting that uses

FG to overlap communication with computation. In this project, we used FORTRAN-based BLAS routines for vector computation, and we used OpenMP to leverage the in-core parallelism offered by the algorithm. We saw speedups in running time by factors of up to 20 compared with an alternate FGT implementation.

5. We have used our experience in designing out-of-core programs using FG to extend FG with additional pipeline structures. We have also identified some FG features that might be redundant, and thus can be removed from the next version of FG.

Bibliography

[1] Lars Arge, Rakesh Barve, Andrew Danner, David Hutchinson, Thomas Mølhave, Octavian Procopiuc, Jörg Rotthowe, Laura Toma, Jan Vahrenhold, Darren Erik Vengroff, Markus Vogel, and Rajiv Wickeremesinghe. *TPIE User Manual and Reference, Draft of May 2006*. http://cs.au.dk/~thomasm/tpie.pdf.

[2] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory (extended abstract). In *STOC '97: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 540–548, 1997.

[3] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[4] Satish Balay, William D. Gropp, Lois C. McInnes, and Barry F. Smith. Petsc home page. http://www.mcs.anl.gov/petsc, 2008.

[5] Satish Balay, William D. Gropp, Lois C. McInnes, and Barry F. Smith. Petsc users manual. Technical Report ANL-95/11 - Revision 3.0, Argonne National Laboratory, 2008.

[6] Andreas Beckmann, Roman Dementiev, and Johannes Singler. Building a parallel pipelined external memory algorithm library. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS)*, 2009.

[7] Basic linear algebra subprograms. http://www.netlib.org/blas.

[8] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31:135–167, 1998.

[9] Mark Broadie and Yusaku Yamamoto. Application of the fast gauss transform to option pricing. *Management Science*, 49(11):1071–1088, 2003.

[10] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, SRC, May 1994.

[11] Geeta Chaudhry and Thomas H. Cormen. Oblivious vs. distribution-based sorting: An experimental evaluation. In *13th Annual European Symposium on Algorithms (ESA 2005)*, volume 3669 of *Lecture Notes in Computer Science*, pages 317–328. Springer, October 2005.

[12] 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.

[13] Thomas H. Cormen and Elena Riccio Davidson. *Asynchronous Buffered Computation Design and Engineering Framework Generator (ABCDEFG) Tutorial and Reference, Version 1.4*.

[14] Thomas H. Cormen and Elena Riccio Davidson. FG: A framework generator for hiding latency in parallel programs running on clusters. In *17th International Conference on Parallel and Distributed Computing Systems (PDCS 2004)*, pages 137–144, September 2004.

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

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

[17] Andreas Crauser and Paolo Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[18] Elena Riccio Davidson and Thomas H. Cormen. The FG programming environment: Reducing source code size for parallel programs running on clusters. In *Second Workshop on Productivity and Performance in High-End Computing (P-PHEC 2005)*, pages 27–34, February 2005.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 137–150, December 2004.

[20] Roman Dementiev. STXXL *Tutorial*, June 2006. Available from stxxl.sourceforge.net.

[21] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. *Journal of Experimental Algorithmics*, 12:1–24, June 2008.

[22] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: standard template library for XXL data sets. *Software Practice and Experience*, 38(6):589–637, May 2008.

[23] Ahmed Elgammal, Ramani Duraiswami, and Larry S. Davis. Efficient kernel density estimation using the fast gauss transform with applications to color modeling and tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(11):1499–1504, 2003.

[24] UCSC Genome Browser. http://www.hgdownload.cse.ucsc.edu/downloads.html (status May, 2004).

[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, December 2003.

[26] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. *New indices for text: PAT trees and PAT arrays*. Prentice-Hall Inc., 1992.

[27] Leslie Greengard and John Strain. The fast Gauss transform. *SIAM Journal on Scientific and Statistical Computing*, 12(1):79–94, 1991.

[28] Leslie Greengard and Xiaobai Sun. A new version of the fast Gauss transform. *Documenta Mathematica*, III:575–584, 1998.

[29] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[30] Project Gutenberg. http://www.gutenberg.org.

[31] IEEE. Standard 1003.1-2001, Portable operating system interface, 2001. http://www.unix.org/version3/ieee_std.html.

[32] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[33] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Conference on Automata, Languages, and Programming*, pages 943–955. Springer-Verlag, 2003.

[34] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, November 2006.

[35] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer, 2003.

[36] Junmo Kim, John W. Fisher III, Anthony Yezzi, Mujdat Cetin, and Alan S. Willsky. A nonparametric statistical method for image segmentation using information theory and curve evolution. *IEEE Transactions on Image Processing*, 14(10):1486–1502, 2005.

[37] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2003.

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

[39] Jing-Rebecca Li and Leslie Greengard. High order accurate methods for the evaluation of layer heat potentials. *SIAM Journal on Scientific Computing*, 31:3847–3860, October 2009.

[40] libstdc++ parallel mode. http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html.

[41] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[42] Mpich2 home page. http://www.mcs.anl.gov/research/projects/mpich2/.

[43] Priya Natarajan, Thomas H. Cormen, and Elena Riccio Strange. Out-of-core distribution sort in the FG programming environment. In *Workshop on Multithreaded Architectures and Applications, Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010.

[44] Rahul S. Sampath, Hari Sundar, and Shravan K. Veerapaneni. Parallel fast Gauss transform. In *ACM/IEEE Conference on Supercomputing*, 2010.

[45] S. Seshadri and Jeffrey F. Naughton. Sampling issues in parallel database systems. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *3rd International Conference on Extending Database Technology (EDBT '92)*, volume 580 of *Lecture Notes in Computer Science*, pages 328–343. Springer-Verlag, March 1992.

[46] Johannes Singler, Peter Sanders, and Felix Putze. MCSTL: The multi-core standard template library. In *Euro-Par: Parallel Processing*, volume 4641 of *LNCS*, pages 682–694, 2007.

[47] Marina Spivak, Shravan K. Veerapaneni, and Leslie Greengard. The fast generalized Gauss transform. *SIAM Journal on Scientific Computing*, 32(5), 2010.

[48] John Strain. Fast adpative methods for the free-space heat equation. *SIAM Journal on Scientific Computing*, 15(1):185–206, Jan 1994.

[49] *StreamIt Language Specification, Version 2.1*, September 2006. http://cag.csail.mit.edu/streamit/papers/streamit-lang-spec.pdf.

[50] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[51] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. Technical Report LCS-TM-622, Massachusetts Institute of Technology, August 2001.

[52] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002.

[53] Threading building blocks. http://www.threadingbuildingblocks.org/documentation.php.

[54] Jurij Šilc, Borut Robič, and Theo Ungerer. *Asynchrony in parallel computing: From dataflow to multithreading*, pages 1–33. Nova Science Publishers, Inc., Commack, NY, USA, 2001.

[55] Shravan K. Veerapaneni and George Biros. The Chebyshev fast Gauss and nonuniform fast Fourier transforms and their application to the evaluation of distributed heat potentials. *Journal of Computational Physics*, 227:7768–7790, August 2008.

[56] Jeffrey Scott Vitter. *Algorithms and Data Structures for External Memory, Foundations and Trends® in Theoretical Computer Science*. now Publishers Inc., 2008.

[57] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.