# RaftLib: A C++ Template Library for High Performance Stream Parallel Processing

Jonathan C. Beard
Peng Li
Roger D. Chamberlain

**Original Article:**
Jonathan C Beard, Peng Li, and Roger D Chamberlain
RaftLib: A C++ Template Library for High Performance Stream Parallel Processing
International Journal of High Performance Computing Applications 1094342016672542, first published on October 19, 2016
**doi:**10.1177/1094342016672542

# RaftLib: A C++ Template Library for High Performance Stream Parallel Processing

**Abstract**

Stream processing is a compute paradigm that has been around for decades, yet until recently has failed to garner the same attention as other mainstream languages and libraries (e.g., `C++`, OpenMP, MPI). Stream processing has great promise: the ability to safely exploit extreme levels of parallelism to process huge volumes of streaming data. There have been many implementations, both libraries and full languages. The full languages implicitly assume that the streaming paradigm cannot be fully exploited in legacy languages, while library approaches are often preferred for being integrable with the vast expanse of extant legacy code. Libraries, however are often criticized for yielding to the shape of their respective languages. RaftLib aims to fully exploit the stream processing paradigm, enabling a full spectrum of streaming graph optimizations, while providing a platform for the exploration of integrability with legacy `C/C++` code. RaftLib is built as a `C++` template library, enabling programmers to utilize the robust `C++` standard library, and other legacy code, along with RaftLib's parallelization framework. RaftLib supports several online optimization techniques: dynamic queue optimization, automatic parallelization, and real-time low overhead performance monitoring.

# Introduction and background

Decries touting the end of frequency scaling and the inevitability of a massively multi-core future are found frequently in current literature [21]. Equally prescient are the numerous papers with potential solutions to programming multi-core architectures [31, 39]. One of the more promising programming modalities to date, and one of the few to break out of the limiting fork-join model, is a very old one: stream processing [18]. The term "stream processing" is also synonymous with data-flow programming and is a natural superset of the more limited map-reduce modality. Until recently stream processing has garnered little attention. RaftLib aims to change that by enabling performant and automatically tuned stream processing within the highly popular `C++` language.

Stream processing is a compute paradigm that views an application as a set of compute kernels (also sometimes termed "filters" [48]) connected by communication links that deliver streams of data. Each compute kernel is typically programmed as a sequentially executing unit. Each stream is abstracted as a first-in, first-out (FIFO) queue, whose exact allocation and construction is dependent upon the link type (and largely transparent to the user). Sequential kernels are assembled into applications that can execute in parallel. Figure 1 is an example of a simple streaming `sum` application, which takes in two streams of numbers, adds each pair, and then writes the result to an outbound data stream.

A salient feature of streaming processing is the compartmentalization of state within each compute kernel [1], which simplifies parallelization logic for the runtime [19] as well as the programming API (compared to standard parallelization methods [2]). Stream processing has two immediate advantages: 1) it enables a programmer to think sequentially about individual pieces of a program while composing a larger program that can be executed in parallel, 2) a streaming runtime can reason about each kernel individually while optimizing globally [38]. Moreover, stream processing has the fortunate side effect of encouraging developers to compartmentalize and separate programs into logical partitions. Logical partitioning is also beneficial for the optimization and tuning process.

Despite the promising features of stream processing, there are hurdles that affect programmers' decision to use the paradigm. First and foremost, before any technical issues are thought of, ease of integration becomes a bottleneck. To make stream processing, and RaftLib successful, a path must be cleared to use streaming within legacy code. Most streaming frameworks require that applications be re-written or substantially modified to conform [34]. RaftLib skirts this hurdle by

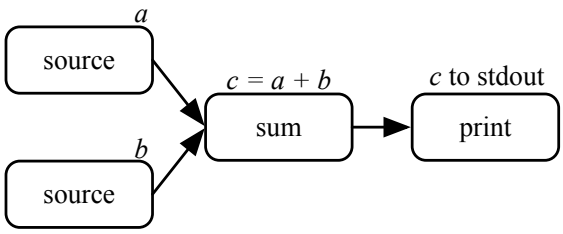


Figure 1: Simple streaming application example with four compute kernels of three distinct types. From left to right: the two source kernels each provide a number stream ($a$ and $b$), the `sum` kernel adds pairs of numbers ($c = a + b$) and the last kernel prints the result ($c$). Each kernel acts independently, sharing data via communications streams depicted as arrows.

existing within one of the most popular languages, `C++` (according to the TIOBE index [49], future versions will include other language bindings to the library). A second hurdle, is the perceived increase in communications cost. The issues leading to thread to thread communication (real or false) are huge, and endemic to all types of thread parallel processing (the issues themselves are too lengthy to discuss in detail, see relevant texts). Stream processing itself, offers several substantive solutions given the directed graph (often acyclic) nature of the communications pattern, whereas in a standard threaded application the ability to reason about these patterns is hampered by the randomness of the access pattern itself. The FIFO pattern of most streaming systems can be optimized using `pre-fetch` instructions found in many modern multi-core processors since the next access is quite regular. Optimizing the communications pattern further involves minimizing the path latency between compute kernels and maximizing the overall throughput through the application. In general, this requires solving the graph partitioning problem which is NP-hard [22], however several good heuristics exist.

This work introduces RaftLib [11, 41], a `C++` template library, which enables safe and fast stream processing. By leveraging the power of `C++` templates, RaftLib can be incorporated with a few function calls and the linking of one additional library. RaftLib aims to transparently parallelize an application in a way that is automatic to the programmer. RaftLib is an auto-tuned streaming system. As such it must mitigate communications cost, adaptively schedule compute kernels, and provide low overhead instrumentation to the runtime so that informed decisions can be made. RaftLib minimizes the communications overhead in a multitude of ways, drawing on research from past works [7, 8, 28, 29]. Machine learning techniques [6, 10] are used to model buffers within the streaming graph and select progressively more appropriate buffer sizes while the application is executing. The framework incorporates low overhead instrumentation, which can be turned on and off dynamically to monitor such metrics as queue occupancy, non-blocking service rate, and utilization. All of these pieces put together make a highly usable and adaptive stream parallel system, which is integrable with legacy code. In addition to being a performant and robust stream processing platform for general or commercial usage, RaftLib is also intended to contribute as a research platform. RaftLib's modular construction enables empirical evaluation a myriad of issues related to data-flow execution without extensive runtime modifications.

## Related work

There are many streaming languages and runtime systems (both academic and commercial). StreamIt [48] is a streaming language and compiler based on the synchronous dataflow model. Storm [46] and Samza [43] are open-source streaming platforms that are focused on message-based data processing. Google Cloud Dataflow [40] is a proprietary stream processing system for Google's cloud. Many of the full languages and frameworks have until recently been suited only to "niche" applications, often with steep learning curves. RaftLib's advantage over them is that a `C++` template library is compiled as optimized machine code from the outset, easily integrable with legacy code, and has more general usage. ScalaPipe [51] and StreamJIT [12] are two similar language extensions for streaming processing with a Scala frontend and a Java frontend, respectively. A similar (but non-streaming) `C++` parallelization template library is the open source Threading Building Blocks (TBB) [42] library originally from Intel. RaftLib differs from the last framework in several ways, first and foremost, is that it supports distributed and heterogeneous stream processing in addition to single node (multi-core) parallelism.

There has been considerable work investigating the efficient execution of streaming applications, both on traditional multi-cores and on heterogeneous compute platforms, from the early work on dedicated data-flow engines [19], to the synchronous data-flow programming model [30]. Lancaster et al. [29] have built low-impact performance monitors for streaming computations across FPGA accelerators and multi-core devices. Padmanabhan et al. [38] have shown how to efficiently search the design space given a model that connects tuning parameters to application performance. Beard and Chamberlain [7] showed how to approximate throughput through a streaming application efficiently using network flow models. RaftLib leverages and expands upon the above work, as it seeks to efficiently execute streaming/data-flow applications.

In order to dynamically "tune" RaftLib, online instrumentation is required. Much previous work has been done in this area as well, although not all specifically targeted towards streaming systems. Tools such as DTrace [14], Pin [33], and even analysis tools such as Valgrind [37] can provide certain levels of dynamic information on executing threads. Other, more modern performance monitoring tools of note are Paradyn [35] and Scalasca [23]. These toolkits provide a multitude of information for parallel systems, however not quite the same type of information that our instrumentation provides. Many of these past works pioneered things like trace compression for instrumentation, however we are interested in eliminating traces entirely. Using the data in real time, then throwing it away reduces the communications overhead dramatically while mirroring the streaming paradigm that it espouses. RaftLib's instrumentation is geared specifically towards stream processing, while leveraging non-streaming instrumentation methods developed by others where possible.

Scheduling a streaming application is akin to partitioning a directed graph. Communication between kernels via streams is accounted for by edge weights (weights potentially calculated using information from libraries like `hwloc` [13]). More advanced partitioning algorithms can add additional degrees of freedom in the form of matching kernels to specific processing resources. Early work by Kernighan et al. [27] gave a heuristic to efficiently partition the graph into two highly communicating partitions. Later work by Sanchis [44] extended partitioning to multiple parts. Partitioning an application is but one part of scheduling. Once an application is set into motion, classic scheduling algorithms like Round Robin, FIFO, and work-stealing can be used to load-balance the application. RaftLib's specific approaches are discussed in the following sections.

## Design considerations

To be successful, stream processing systems must provide efficient ways of accessing data as the program needs it, while minimizing communications cost, and maximizing the use of given compute resources. The stream access pattern is often that of a sliding window [48], which is accommodated efficiently in RaftLib through a `peek_range` function. Streaming systems, both long running and otherwise, often must deal with behavior that differs from the steady state [8]. Non-steady state behavior is often also observed with data-dependent behavior, resulting in very dynamic I/O rates (behavior also observed in [48]). This dynamic behavior, either at startup or elsewhere during execution, makes the analysis and optimization of streaming systems difficult, however not impossible. RaftLib's handling of dynamic behavior is demonstrated empirically through a text search application. Many text search algorithms have the property, that while the input volume is often fixed, the downstream data volume varies dramatically as the algorithm heuristically skips over non-matching patterns. Compute kernel developers should focus on producing the most efficient algorithm for an application, and not the burden of handling data movement or resource allocation. RaftLib dynamically monitors the system to eliminate data movement and resource allocation bottlenecks where possible, freeing the programmer to focus on application logic.

At one time it was thought that programmers were probably best at resource assignment [17], whereas automated algorithms were often better at partitioning an application into compute kernels (synonymous to the hardware-software co-design problem discussed in [4]). Anecdotal evidence suggests that the opposite is often true. Programmers are typically very good at choosing algorithms to implement within kernels, however they have either too little or too much information to consider when deciding how to parallelize or where to place a computation. Understanding this information is critical to understanding the secondary effects that each decision has for the performance of an application. Within a streaming data-flow graph, it is often possible to replicate kernels (executing them in parallel) to enhance performance without altering the application semantics [32]. RaftLib exploits this ability to extract more pipeline and task parallelism at runtime (dynamically) without further input from the programmer. The next few sections discuss how these considerations are embodied within the programmer interface.

## RaftLib description

The complexity of traditional parallel code (e.g., pthreads) decreases productivity, which can increase development costs [24]. This complexity also limits the access to the performance benefits of modern chip multi-processors to more experienced programmers. RaftLib aims to bring simplicity to parallel programming so that everyone can experience the performance gains promised by our multi-core future to novice programmers who would otherwise only write sequential code. The streaming compute paradigm generally, and RaftLib specifically, enables the programmer to compose sequential code (compute kernels) and execute not only in parallel but distributed parallel (networked nodes) using the same source code.

RaftLib has a number of useful innovations as both a research platform and a programmer productivity tool. As a research platform, it is first and foremost easily extensible; modularized so that individual aspects can be explored without a full system re-write. It enables multiple modes of exploration: 1) how to effectively integrate pipeline parallelism with standard threaded and/or sequential code, 2) how to reduce monitoring overhead, 3) how best to algorithmically map compute kernels to resources, 4) how to model streaming applications quickly so that results are relevant

during execution. It is also fully open source and publicly accessible [41]. As a productivity tool it is easily integrable with legacy `C++` code. It allows a programmer to parallelize code in both task and pipelined fashions.

Before diving into RaftLib as a research platform, we introduce a bit more of streaming through a concrete RaftLib example. The `sum` kernel from Figure 1 is an example of a kernel written in a sequential manner (code shown in Figure 2). It is authored by extending a base class: `raft::kernel`. Each kernel communicates with the outside world through communications "ports." The base kernel object defines `input` and `output` port class accessible objects. These are inherited by sub-classes of `raft::kernel`. Port container objects can contain any type of port. Each port itself is presented as a FIFO interface. The constructor function of the `sum` kernel adds the ports. In this example, two input ports are added of type `T` as well as an output port of the same type. Each port gets a unique name which is used by the runtime and the programmer to address specific ports. The real work of the kernel is performed in the `run()` function, which is called by the scheduler. The code within this section can be thought of as a "main" function of the kernel. Input and output ports can access data via a multitude of methods from within the `run()` function. Accessing a port is safe, free from data race, and other issues that often plague traditional parallel code [5].

Figure 3 shows the full application topology from Figure 1 assembled in code. Assembling the topology can be thought of as connecting a series of actors. Each actor is sequential on its own, but when combined in a graph can be executed in parallel. Once the kernel "actors" are assembled into an application, the runtime starts to work parallelizing the application. Barrier operations are also provided so that sequential operations can be performed within the main function that interact with the parallel kernels, such as those described in Figure 4.

```
template < typname T > class sum : public raft::kernel
{
public:
   sum() : raft::kernel()
   {
      input. template addPort< T >( "input_a", "input_b" );
      output.template addPort< T >( "sum" );
   }

   virtual raft::kstatus run()
   {
      T a,b;
      input[ "input_a" ].pop( a );
      input[ "input_b" ].pop( b );
      auto c( output[ "sum" ].template allocate_s< T >() );
      (*c) = a + b;
      return( raft::proceed );
   }
};
```

Figure 2: A simple example of a `sum` kernel which takes two numbers in via ports `input_a` and `input_b`, adds them, and outputs them via the `sum` stream. The `allocate_s` call returns an object which releases the allocated memory to the downstream kernel with the call of its destructor as it exits the stack frame.

```
const std::size_t count( 100000 );
using ex_t = std::int64_t;
using source  = raft::random_variate< ex_t, raft::sequential >;
sum< ex_t > sum_kernel();
raft::map m;
m += source( 1, count ) >> sum_kernel[ "input_a" ];
m += source( 1, count ) >> sum_kernel[ "input_b" ];
m += sum_kernel[ "sum" ] >> print< ex_t, '\n' >();
m.exe();
```

Figure 3: Example of a streaming application map for a "sum" application (topology given in Figure 1). Two random number generators are instantiated inline with the mapping (labeled as `source`), each of which sends a stream of numbers to the `sum` kernel, which then streams the sum to a `print` kernel. The `+=` operator overload adds kernels from the current line to the map. The `>>` overload indicates a stream or link from one kernel to another. The kernel objects are created inline above for conciseness, however, the `raft::kernel::make<type>` syntax is preferred as it avoids additional copy overhead.

There are many factors that have led to the design of RaftLib. Chief amongst them is the desire to have a fully open source framework to explore how best to integrate stream processing with legacy code (in this case `C/C++`). In addition to being a productivity enhancing platform, it also serves as a research platform for investigating optimized deployment and optimization of stream processing systems. Scheduling, mapping, and queueing behavior are each important to efficient, high-performance execution. RaftLib is intended to facilitate empirical investigation within each of these areas. The following sections will discuss RaftLib's programmer interface for authoring applications, its usage as a research platform, followed by a concrete benchmark compared to other parallelization frameworks.

## *Authoring streaming applications*

RaftLib views each compute kernel as a black-box at the level of a port interface. Once ports are defined, the only observability that the runtime has is the interaction of the algorithm implementation inside the compute kernel with those ports, and the kernel's interactions with the hardware. A new compute kernel is defined by extending `raft::kernel` as in Figure 2. Kernels have access to add named ports, with which, the kernel can access data from incoming or write to outgoing data "streams." Once defined, programmers have multiple methods to access data from each stream. The example in Figure 2 shows the simplest method (`pop`) to get data from the input stream, which as the name suggests, pops an element from the head of the port's stream and returns it to the programmer by copy to the variables `a` and `b`. A reference to memory on the output stream is returned by the `allocate_s` function (equivalently for fundamental types it is just as efficient to incur a copy using the push operator). If the object is not plain old data, RaftLib constructs the object in place on the output port. The return object from the `allocate_s` call has associated signals accessible through the `sig` variable. There are multiple calls to perform push, pop, and range style operations, each embodies some type of copy semantic (either zero copy or single copy). All operators provide a means to send or receive synchronous signals that can be used by the programmer, kernels will receive the signal at the same time the corresponding

data element is received (useful for things like end of file signals). Asynchronous signaling (i.e., immediately available to downstream kernels) is also available. Future implementations will utilize the asynchronous signaling pathway for global exception handling.

Arranging compute kernels into an application is one of the core functionalities of a stream processing system. RaftLib links compute kernels via an operator overload of the right shift operator `>>` to mimic the pattern of the `C++` stream operator. The `>>` operator has the effect of assigning the output port of the compute kernel on the left hand side of the operator to the input port of the compute kernel on the right hand side of the operator. Once kernels are linked, they are added to a map object of type `raft::map` to be executed via an overload of the `+=` operator. The return is an object containing iterators to the source and destination kernels added in the last add increment operation. Figure 3 shows our simple example application which takes two random number generating kernels, adds pairs of the random numbers from the `source` kernels using the `sum` kernel and prints them.

The graph itself is executed as the `raft::map exe()` function is called, or if a barrier issued by the programmer as shown in Figure 4. Before executing, all ports are checked to ensure that they are connected, if not an exception is thrown. While type checking is performed at the time of port linking, allocation is performed lazily, right before actual execution. The runtime itself selects the type of allocation depending on where each compute kernel is mapped, currently the choices are one of (POSIX shared memory, heap allocated memory, or TCP link). Since mapping can place kernels at any resource for which an implementation is available, the allocation types themselves must follow. RaftLib supports type conversion through compatible types, as a consequence, the runtime can select the narrowest convertible type. Compression is also possible as well, and future work will investigate how best to incorporate link compression. Each stream is monitored via the runtime and dynamically re-allocated as needed (this is beneficial for both performance, and device alignment requirements).

Streaming applications are often ideally suited for long running, data intensive applications such as big data processing or real-time data analytics. The conditions for these applications often change during the execution of a single run. Algorithms frequently use different optimizations based on differing inputs (e.g., sparse matrix vs. dense matrix multiply). The application can often benefit from additional resources or differing algorithms within the application, to eliminate bottlenecks as they emerge. RaftLib gives the user the ability to specify synonymous kernel groupings called submaps, that the runtime can swap out to optimize the computation. These can be kernels that are implemented for multiple hardware types, or can be differing algorithms. For instance, a RaftLib version of the UNIX utility `grep` could be implemented with multiple search algorithms, swapped out dynamically at runtime.

Integration with legacy `C++` code is one of our goals. As such, it is imperative that RaftLib work seamlessly with the `C++` standard library functions. Figure 4 shows how a `C++` container can be used directly as an input queue to a streaming graph. It can be accessed in parallel if the out of order processing hint is given by the user. Just as easily, a single value could be read in. Output integration is equally simple. Kernels are available to assign data streams to standard library containers, or a reduction to a single value is also possible.

Copying of data is often an issue as well within stream processing systems. RaftLib provides a `for_each` kernel (Figure 5), which has behavior distinct from the `write_each` and `read_each` kernels. The `for_each` takes a pointer value and uses its memory space directly as a queue for downstream compute kernels. This is essentially a zero copy and enabling behavior from a "streaming" application similar to that of an OpenMP [15] parallelized loop. Unlike the `C++` standard library

```
using ex_t = std::uint32_t;
/** data source & receiver container **/
std::vector< ex_t > v,o;
ex_t i( 0 );
/** fill container **/
auto func( [&](){ return( i++ ); } );
while( i < 1000 ){ v.emplace_back( func() ); }
/** read from one kernel and write to another **/
auto readeach( read_each< ex_t >( v.begin(), v.end() ) );
auto writeeach( write_each< ex_t >( std::back_inserter( o ) ) );
raft::map m;
m += readeach >> writeeach;
m.barrier( writeeach );
/** data is now copied to 'o' **/
```

Figure 4: Syntax for reading and writing to C++ standard library containers from raft::kernel objects. The read_each and write_each kernels are reading and writing on independent threads.

```
int *arr = { 0, ..., N };
int val = 0;
raft::map m;
m += for_each< int >( arr, arr_length ) >> some_kernel< int >()
        >> reduce< int, func /* reduct function */ >( val );
/** wait for map to finish executing **/
m.exe();
/** val now has the result **/
```

Figure 5: Example of the for_each kernel, which is similar to the C++ standard library for_each function. The data from the given array is divided amongst the output queues using zero copy, minimizing data extraneous data movement.

for_each, the RaftLib version provides an index to indicate position within the array for the start position. This enables the compute kernel reading the array to calculate the position within it. When this kernel is executed, it appears as a kernel only momentarily, essentially providing a data source for the downstream compute kernels to read. Data from arrays and C++ containers can be divided up dynamically to facilitate work stealing as a means of load balancing. Further reducing copying is a size specific allocation mechanism that passes via reference versus copy when it is more efficient to do so.

Code verbosity is often an issue. Readily available in C++ are examples of full class and template decarations, when what is wanted is the ability to create a simple function without a full class declaration. C++11 has met the demand for this functionality with lambda functions. RaftLib brings lambda compute kernels, which give the user the ability to declare a fully functional, independent kernel, while freeing her from the cruft that would normally accompany such a declaration. Figure 6 demonstrates the syntax for a single output random number generator. The closure type of the lambda operator also allows for usage of the static keyword to maintain state within the function [16]. These kernels can be duplicated and distributed, however they do induce one com-

plication if the user decides to capture external values by reference instead of by value. Undefined behavior may result if the kernel is duplicated; especially across a network link (an issue to be resolved in subsequent versions of RaftLib).

```
using ex_t = std::uint32_t;
/** instantiate lambda kernel as source **/
auto lambda_kernel(
   lambdak< ex_t >( 0, 1, []( Port &input, Port &output )
      {
         auto out( output[ "0" ].allocate_s< ex_t >() );
         (*out) = rand();
      } /** end lambda kernel **/ )

);
raft::map m;
m += lambda_kernel >> print< ex_t, '\n' >();
m.exe();
```

Figure 6: Syntax for lambda kernel. The user specifies port types as template parameters to the kernel, in this example `std::uint32_t`. If a single type is provided as a template parameter, then all ports for this lambda kernel are assumed to have this type. If more than one template parameter is used, then the number of types must match the number of ports given by the first and second function parameters (input and output port count, respectively). The number of input ports is zero and the number of output ports is one for this example. Ports are named sequentially starting with zero. The third parameter is a `C++11` lambda function, which is executed by the runtime.

### *RaftLib as a research platform*

As a research platform, RaftLib is designed to enable the investigation of a number of questions that impact the performance of streaming applications. In addition to the open question of how best to blend parallel and sequential execution, RaftLib intends to be a platform for facilitating scheduling, resource mapping, and buffer allocation (queueing) within streaming/data-flow systems. Other research avenues abound, however, most of them stem from these core questions. Our focus here, is not on solving each question but in facilitating further research.

Blending parallel code with sequential code, often results in a "spaghetti code" that is hard to debug [20]. Streaming requires that each kernel maintain user accessible state within the compute kernel, simplifying the reasoning process for the programmer. When building an application all that is left is to string compute kernels together. The best way to manage the interface between code executing in parallel via streams and procedural code remains an open question. Likewise, what information can the programmer give the runtime to aid optimization decisions? Some applications require data to be processed in order, others are okay with data that is processed out of order, yet others can process the data out of order and re-order at some later time. RaftLib accommodates all of the above paradigms. Currently RaftLib supports insertion of ordering information while linking streams (see Figure 7), but more hints can easily be incorporated in future versions (especially if user studies hint that they are useful).

Automatic parallelization of candidate kernels is currently accomplished by analyzing the graph for segments that can be replicated preserving the application's semantics. As part of the graph

```
raft::map m;
m += S[ "0" ] >> A >> T[ "0" ];
m += S[ "1" ] >> raft::order::out >> B;
m += B[ "0" ] >> C;
m += B[ "1" ] >> raft::order::out >> D;
m += C >> E[ "0" ];
m += D >> raft::order::out >> E[ "1" ];
m += E >> raft::order::out >> T[ "1" ];
```

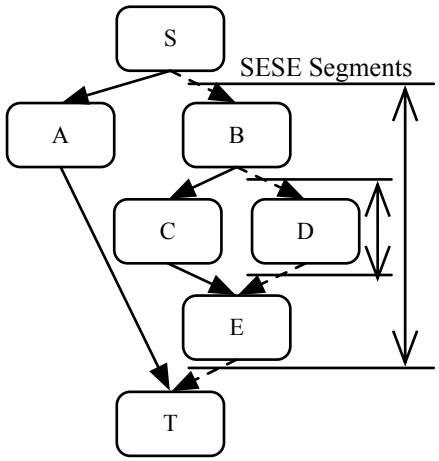


Figure 7: Example of a compute kernel mapping, with multiple single entry, single exit (SESE) sections identified by the user through the `raft::order::out` enumerated value. These SESE out of order sections can be parallelized as the runtime sees fit (e.g., "D" or "$B \rightarrow E$").

analysis process, single entry single exit (SESE) [26] segments are identified (with respect to user indicated out of order links) and indexed as potential parallelization points (see Figure 7). Split and reduce adapters are inserted where needed. Custom split/reduce objects can be created by the user by extending the default split/reduce objects. Split data distribution can be done in many ways, and the runtime attempts to select the best amongst round-robin and least-utilized strategies (queue utilization used to direct data flow to less utilized servers). As with all of the specific mechanisms that we will discuss, each of these approaches is designed to be easily swapped out for alternatives, enabling empirical comparative study between approaches.

Given an application topology to execute, the kernels need to be assigned to specific compute resources, and scheduled for execution. Scheduling of compute kernels within a streaming application has been the subject of much research. Conceptually it has two parts, initial resource assignment or "mapping" of kernels to compute resources and then scheduling the kernels to actually execute temporally. RaftLib currently supports multiple schedulers, including OS level threads, and user space "fibers" or "threadlets" within each heavy-weight kernel thread using the Qthreads lightweight thread library [50]. Threadlets give the runtime yet another degree of freedom in scheduling since within each kernel thread, the scheduler can partition the time quanta to its threadlets in an application dependent manner. Different architecture and operating system combinations prefer different types of threading models, so an open question is how best to switch between these models on the same architecture. Even more complicated, some virtual memory systems perform better with combinations of heavy-weight processes and threads. Asking for the "best" combination is a very loaded question at best. For applications that require it, RaftLib supports forcing a specific source for a particular compute kernel.

Fast partitioning itself is a vibrant area of research. RaftLib enables empirical evaluation of the partitioning problem in isolation from the scheduling problem. The act of partitioning kernels and threads of a streaming application to compute resources is nearly identical to the decades old problem of partitioning and mapping a circuit. Partitioning for RaftLib means finding the best

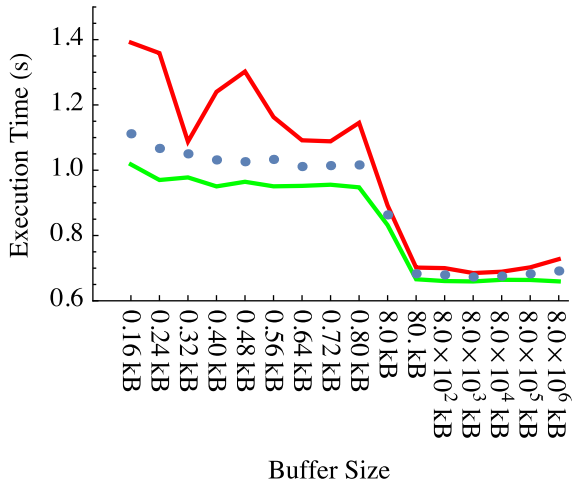


Figure 8: Buffer sizes for a matrix multiply application [6], shown for an individual queue (all queues sized equally). The dots (middle data point) indicate the simple mean of each observation (each observation is a summary of 1k executions). The red (top) and green (bottom) lines indicate the $95^{th}$ and $5^{th}$ percentiles respectively. The execution time increases slowly with buffer sizes $\geq$ 8 MB, as well as becoming far more varied.

layout of compute kernels that minimizes the cost of communications between compute operations while attempting to maximize the match of the hardware to the operations being performed within each kernel. As mentioned before, the partitioning problem in general is NP-hard [22]. The default partitioner uses a variant of $k$-way partitioning similar to the work by Sanchis [44]. Separating the scheduling and partitioning enables researchers, and programmers to consider one problem (e.g., data locality in mapping) without necessarily having to dive into scheduling each kernel temporally (although tuning both knobs could lead to better overall performance).

As illustrated in Figure 8, the allocated size of each queue of a streaming application can have a significant impact on performance (the data from the figure is drawn from a matrix multiply application, as in [6], performance based on overall execution time). One would assume perhaps that simply selecting a very large buffer might be the best choice, however as shown the upper confidence interval begins to increase after about eight megabytes. Instrumentation using the PAPI [36] toolkit, shows that as the queue increases in size the L1,L2 miss rates increase dramatically, as do "soft" page-faults, and finally "hard" page-faults begin cropping up towards the extreme right side of the queue sizing for the platform utilized (Note: it should be apparent that these trends hold in general, the exact shape and limits are architecture dependent). RaftLib currently uses a variety of approaches to optimize buffer allocation size ranging from branch and bound search to queueing network models guided by machine learning (described in detail by [6]). The best solution to optimizing buffer allocation and placement is still an open question. RaftLib modularizes the interface to dynamically resize buffers, and buffer placement, so that new methods may be incorporated as they are developed.

Considering the application as a whole for optimization is also possible for RaftLib (i.e., tuning more than one knob across an entire application). Prior works by Beard and Chamberlain [7] demonstrated the use of flow models to estimate the overall throughput of an application. The flow-

model approximation procedure can be combined with well known optimization techniques such as simulated annealing, analytic decomposition [38], or other heuristic techniques to continually optimize long-running high throughput streaming applications. In practice, local (to each compute kernel) search often works better due to reduced overall communications cost during dynamic optimization [6]. Currently RaftLib uses a combination of flow model based optimization followed by localized heuristic search. Modularity enables easy expansion as more efficient methods are developed.

Performance monitoring is essential to the optimization and tuning of systems. In addition to performance data pertinent to the tuning of standard applications (e.g., performance counters), RaftLib provides instrumentation that is specifically useful to the tuning of an application structured as a streaming directed graph (abstract arrangement depicted in Figure 10). Specifically RaftLib can monitor statistics such as queue occupancy (mean, and full histogram), non-blocking service rate (see Figure 9 for example, online approximated rate and variance, as well as time averaged), and overall throughput. The data collection process, and instrumentation itself is optimized to reduce overhead and has been the subject of much research [6, 9, 29]. As new instrumentation methods are developed, they can be easily added to the RaftLib platform, improving the statistics to be optimized over.

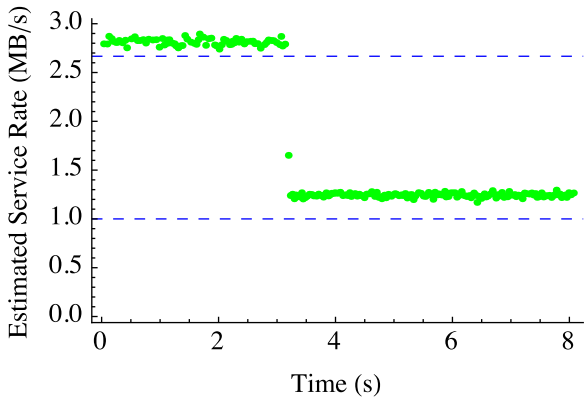


Figure 9: Depiction of the ideal (drawn from empirical data) of the instrumentation's ability to estimate the service rate while the application is executing. Each dot represents the converged service rate estimate ($y$-axis). The top and bottom dashed lines represent the first and second phases as verified by manual measurement in isolation.

Non-blocking service rate, and distribution of that service rate are of particular interest when using stochastic queueing models to optimize a streaming system. Stochastic models are desirable because they are much faster than the alternatives, e.g., branch-and-bound search, which require many memory reallocations. Both service rate and process distribution can be extremely difficult to determine online without effecting the behavior of the application (i.e., degrading application performance). In previous works, Beard and Chamberlain [9] show that a heuristic approach can approximate the non-blocking service rate with relatively high accuracy and very low overhead. RaftLib incorporates this approach. Figure 9 shows the instantaneous approximations of service

rate using this method for a microbenchmark implemented with RaftLib. This method is critical for dynamic optimization using stochastic models, and is available within RaftLib.

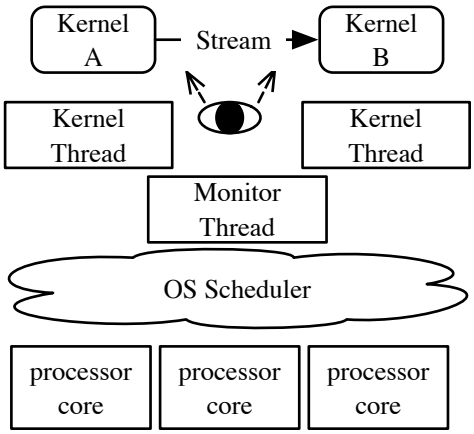


Figure 10: High level depiction of the abstraction layers coalesced around a simple streaming application with two compute kernels. An independent monitor thread serves to instrument the queue. Both the kernel threads and monitor threads are subject to the runtime and operating system (OS) scheduler.

One often overlooked benefit of stream processing from the programmer perspective is that data "streams" can be contiguous in memory. Within RaftLib, fundamental types are by default contiguous, the exact memory alignment is selected by the runtime. Vectorized mathematical operations are a stalwart feature of high performance computation. For machine architectures that support SIMD instructions, RaftLib has specialized kernels for basic operations (more to be added in the future) which support vectorized addition, subtraction, and multiplication on input ports. This is important as the `C++` compiler often cannot determine when a particular vector operation could be safely inserted. The contiguous alignment of data on input ports, and indeed the regular access pattern provided via a FIFO communications paradigm are perfect for cache positioning hints provided by some computer architectures. FIFO patterns are also quite useful for determining where to optimally place memory within NUMA systems as the reader and writer are in defined locations.

The "share-nothing" mantra of stream processing might introduce extra overhead compared to looser parallelization paradigms, however this overhead is paid for by ease of parallelization. Each compute kernel can be easily duplicated on the same system, on different hardware across network links or really any compute resource for which an implementation is available (or a translator exists). As a research vehicle, RaftLib enables studies that explore how the communication and resource placement can be optimized. As a productivity tool, we are more interested in how few lines of code it takes to produce a result. Mentioned but not described has been the distributed nature of RaftLib. The capability to use network connections for many distributed systems is clunky at best. With RaftLib there is no difference between a distributed and a non-distributed program from the perspective of the developer. A separate system called "oar" is a mesh of network clients that continually feed system information to each other in order to facilitate distributed RaftLib computation. This information is provided to RaftLib in order to continuously optimize

and monitor Raft kernels executing on multiple systems. Future work will see its full integration, as well as container integration facilitated through "oar."

# Benchmarking

Text search is used in a variety of applications. We will focus on the exact string matching problem which has been studied extensively. The stalwart of string matching applications (both exact and inexact) is the GNU version of the `grep` utility. It has been revised and optimized for 20+ years, resulting in excellent single threaded exact string matching performance ($\sim$ 1.2 GB/s) on our test machine (see Table 1). To parallelize GNU `grep`, the GNU Parallel [47] utility is used to spread computation across one through 16 cores. Two differing text search algorithms will be tested and parallelized with RaftLib. One will utilize the Aho-Corasick [3] string matching algorithm, which is quite good for multiple string patterns. The other will use the Boyer-Moore-Horspool algorithm [25], which is often much faster for single pattern matching. The realized application topology for both string matching algorithms implemented with RaftLib, are conceptually similar to Figure 11, however the file read exists as an independent kernel only momentarily as a notional data source since the runtime utilizes zero copy, and the file is directly read into the in-bound queues of each `match` kernel.

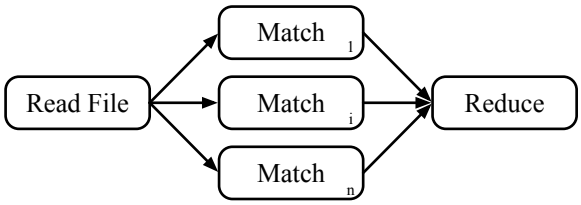


Figure 11: String matching stream topology for both Boyer-Moore-Horspool and Aho-Corasick algorithms. The first compute kernel (at left) reads the file and distributes the data. The second kernel labeled `Match` uses one of the aforementioned algorithms to find string matches within the streaming corpus. The matches are then streamed to the last kernel (at right) which combines them into a single data structure.

Figure 12 shows code necessary to generate the application topology used to express both string matching algorithms using RaftLib. Not shown is the code to handle arguments, setup, etc. Note that there is no special action required to parallelize the algorithm. The `filereader` kernel takes the file name, it distributes the data from the file to each string matching kernel. The programmer can express the algorithm without having to worry about parallelizing it. The programmer simply focuses on the sequential algorithm. Traditional approaches to parallelization require the programmer to have knowledge of locks, synchronization, and often cache protocols to safely express a parallel algorithm. Even more exciting is that when using RaftLib, the same code can be run on multi-cores in a distributed network without the programmer having to do anything differently. The partitioner decides where to run which piece of the application and the online scheduler can make decisions to tune performance dynamically.

For comparison we contrast the performance of our implementations of Aho-Corasick and Boyer-Moore-Horspool against the GNU `grep` utility and a text matching application implemented using the Boyer-Moore algorithm implemented in Scala running on the popular Apache Spark framework.

```
using strsearch = raft::search< raft::ahocorasick >;
std::vector< hit_t > total_hits;
raft::map m;
/** capture an object with source and destination iterators **/
auto kernels( m += filereader( file, offset ) >> strsearch( search_term ) );
/** get begin and end iterator to destination **/
std::tie( BEGIN, END ) = kernels.getDst();
m += (*BEGIN).get() >> write_each< match_t >( std::back_inserter( total_hits ) );
/** wait for dst kernel to complete **/
m.exe();
```

Figure 12: Implementation of the string matching application topology using RaftLib. The actual search kernel is instantiated by making a `search` kernel. The exact algorithm is chosen by specifying the desired algorithm as a template parameter to select the correct template specialization.

We'll use a single hardware platform with multiple cores and a Linux operating system (see Table 1).

We use version 2.20 of the GNU `grep` utility. In order to parallelize GNU `grep`, the GNU Parallel [47] application is used (version 2014.10.22), with the default settings. RaftLib (and all other applications/benchmarks used) is compiled using GNU GCC 4.8 with compiler flags "`-Ofast`." For this set of experiments, the maximum parallelism is capped to the number of cores available on the target machine. A RAM disk is used to store the text corpus to ensure that disk IO is not a limiting factor. The corpus to search is sourced from the post history of a popular programming site [45] which is $\sim$ 40 GB in size. The file is cut to 30 GB before searching. This cut is simply to afford the string matching algorithms the luxury of having physical memory equal to the entire corpus if required (although in practice none of the applications required near this amount). All timing is performed using the GNU `time` utility (version 1.7) except the Spark application, which uses its own timing utility.

Table 1: Summary of Benchmarking Hardware.

| Processor | Cores | RAM | OS Version |
|---|---|---|---|
| Intel Xeon E5-2650 | 16 | 64 GB | Linux 2.6.32 |

Figure 13 shows the throughput (in GB/s) for all of the tested string matching applications, varying the utilized cores from one through 16. A data point is shown for each repetition (10x) for each benchmark, for each thread count. The performance of the GNU `grep` utility when single threaded is quite impressive. It handily beats all the other algorithms for single core performance (when not using GNU Parallel, as shown in the figure). Perfectly parallelized (assuming linear speedup) the GNU `grep` application could be capable of $\sim$ 16 GB/s. When parallelized with GNU Parallel however, that is not the case.

The performance of Apache Spark when given multiple cores is quite good. The speed-up is almost linear from a single core though 16 cores. The Aho-Corasick string matching algorithm using RaftLib performs almost as well, topping out at $\sim$ 1.5 GB/s to Apache Spark's $\sim$ 2.8 GB/s. RaftLib has the ability to quickly swap out algorithms during execution, this was disabled for this benchmark so we could more easily compare specific algorithms. Manually changing the algorithm RaftLib used to Boyer-Moore-Horspool, the performance improved drastically. The speed-up from one through 10 cores is now linear, with the 30 GB file searched in $\sim$ 4.1 s which gives it close to

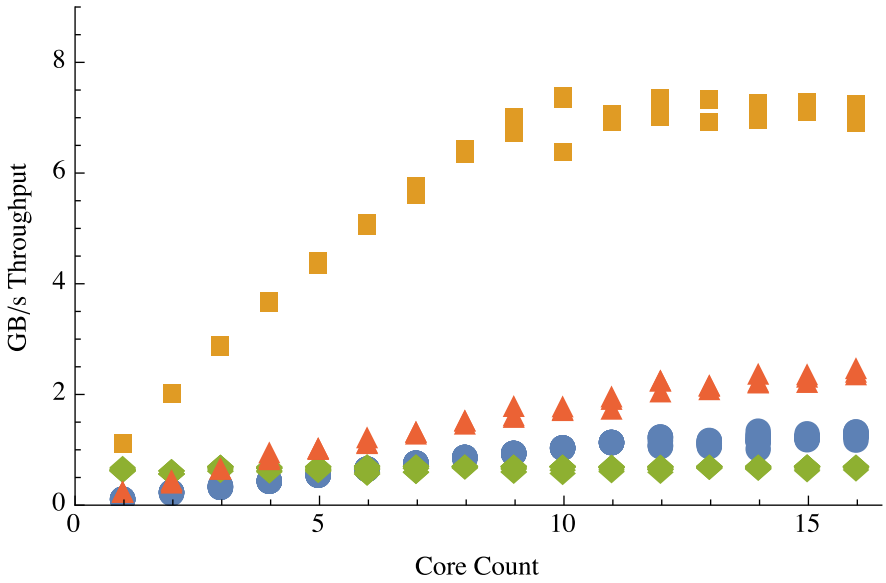


Figure 13: This figure shows the performance of each string matching application in GB/s by utilized cores. This is calculated using a 30 GB corpus searched on the hardware from Table 1. The green diamonds represent the GNU Parallel parallelized GNU `grep`. The red triangles represent Apache Spark. The blue circles and gold squares represent the Aho-Corasick and Boyer-Moore-Horspool text search algorithms, respectively, parallelized using RaftLib.

8 GB/s throughput.

Overall the performance of the RaftLib Aho-Corasick string matching algorithm is quite comparable to the one implemented using the popular Apache Spark framework. The Boyer-Moore-Horspool however outperforms all the other algorithms tested. The change in performance when swapping algorithms indicates that the algorithm itself (Aho-Corasick) was the bottleneck. Once that bottleneck is removed we found that the memory system itself becomes the bottleneck. All in all, the performance of RaftLib is quite good, comparable with (arguably better than) one of the best current distributed processing frameworks (Apache Spark) and far better than the popular command line parallelizing utility GNU Parallel for this application.

## Conclusions & Future Work

RaftLib has many features that enable a user to integrate fast and safe streaming execution within legacy `C++` code. It provides interfaces similar to those found in the `C++` standard library, which we hope will enable users to quickly pick up how to use the library. New ways were also described to specify compute kernels, such as the "lambda" kernels which eliminates much of the "boiler-plate" code necessary to describe a full `C++` class or template. The RaftLib framework enables massively parallel execution, in a simple to use form. The same code that executes locally can execute distributively with the integration of the "oar" framework. No programming changes are necessary. This differs greatly from many current open source distributed programming frameworks.

RaftLib is put forward as a tool to enable programmers to safely exploit parallelism, from within a familiar environment. While doing that, it also lays a foundation for future research. How best to integrate stream processing with sequential computation is still an open question. Pragma methods such as OpenMP *for loop* parallelization work well for fork-join parallelism, however they are far from ideal (in complexity, and extracting as much parallelism as is possible from an application). RaftLib promises similar (or greater) levels of parallelism that are automatically optimized by the runtime. The RaftLib framework provides a platform for safe and fast parallel streaming execution within the `C++` language. It serves as a productivity tool and a research vehicle for exploring integration and optimization issues. Stream processing, and data-flow computing in general, has been around as a well-known concept for over four decades [18]. Despite this long history, not a single streaming language has broken into the top ten programming languages (as kept by TIOBE [49]). We hope that RaftLib serves as a catalyst to gain more than a niche user base to the stream processing paradigm.

# Funding

This work was supported by Exegy, Inc., and VelociData, Inc. Washington University in St. Louis and R. Chamberlain receive income based on a license of technology by the university to Exegy, Inc., and VelociData, Inc.

# References

[1] William B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, 1982.

[2] Vikram Adve, Alan Carle, Elana Granston, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, John Mellor-Crummey, Scott Warren, and Chau-Wen Tseng. Requirements for data-parallel programming environments. Technical report, DTIC Document, 1994.

[3] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] Péter Arató, Sándor Juhász, Zoltán Ádám Mann, András Orbán, and Dávid Papp. Hardware-software partitioning in embedded system design. In *IEEE International Symposium on Intelligent Signal Processing*, pages 197–202. IEEE, 2003.

[5] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[6] Jonathan C. Beard. *Online Modeling and Tuning of Parallel Stream Processing Systems*. PhD thesis, Department of Computer Science and Engineering, Washington University in St. Louis, August 2015.

[7] Jonathan C. Beard and Roger D. Chamberlain. Analysis of a Simple Approach to Modeling Performance for Streaming Data Applications. In *Proc. of IEEE Int'l Symp. on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 345–349, August 2013.

[8] Jonathan C. Beard and Roger D. Chamberlain. Use of a Levy Distribution for Modeling Best Case Execution Time Variation. In András Horváth and Katinka Wolter, editors, *Computer Performance Engineering*, volume 8721 of *Lecture Notes in Computer Science*, pages 74–88. Springer International, 2014.

[9] Jonathan C. Beard and Roger D. Chamberlain. Run Time Approximation of Non-blocking Service Rates for Streaming Systems. In *Proceedings of the 17th IEEE International Conference on High Performance and Communications*, pages 792–797. IEEE, August 2015.

[10] Jonathan C. Beard, Cooper Epstein, and Roger D. Chamberlain. Online Automated Reliability Classification of Queueing Models for Streaming Processing using Support Vector Machines. In *Proceedings of Euro-Par 2015 Parallel Processing*, pages 82–93. Springer, August 2015.

[11] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM 2015, pages 96–105, New York, NY, USA, February 2015. ACM.

[12] Jeffrey Bosboom, Sumanaruban Rajadurai, Weng-Fai Wong, and Saman Amarasinghe. StreamJIT: A commensal compiler for high-performance stream programming. In *Proc. of ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 177–195. ACM, 2014.

[13] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[14] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[15] Rohit Chandra. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[16] Working Draft, Standard for Programming Language C++. `http://goo.gl/JIOjsU`. Accessed Ocbober 2014.

[17] G De Michell and Rajesh K Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

[18] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[19] Jack B Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[20] Marc Eisenstadt. Tales of debugging from the front lines. In *Empirical Studies of Programmers: Fifth Workshop*, pages 86–112. Palo Alto, CA: Ablex Publishing Corporation, 1993.

[21] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *38th International Symposium on Computer Architecture (ISCA)*, pages 365–376. IEEE, 2011.

[22] Michael R Garey and David S Johnson. *Computers and Intractability*, volume 29. WH Freeman, 2002.

[23] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[24] Lorin Hochstein, Jeffrey Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K Hollingsworth, and Marvin V Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *Proc. of ACM/IEEE Supercomputing Conference*, pages 35–35. IEEE, 2005.

[25] R Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[26] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *ACM SigPlan Notices*, volume 29, pages 171–185. ACM, 1994.

[27] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[28] Joseph M. Lancaster, E. F. Berkley Shands, Jeremy D. Buhler, and Roger D. Chamberlain. TimeTrial: A low-impact performance profiler for streaming data applications. In *Proc. IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, September 2011.

[29] Joseph M. Lancaster, Joseph G. Wingbermuehle, Jonathan C. Beard, and Roger D. Chamberlain. Crossing Boundaries in TimeTrial: Monitoring Communications Across Architecturally Diverse Computing Platforms. In *Proc. 9th IEEE/IFIP Int'l Conf. Embedded and Ubiquitous Computing*, October 2011.

[30] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9), 1987.

[31] Charles E Leiserson. The Cilk++ Concurrency Platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[32] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. Adding data parallelism to streaming pipelines for throughput optimization. In *Proc. of IEEE Int'l Conf. on High Performance Computing*, 2013.

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[34] Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proc. of ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 1–18. ACM, 2013.

[35] Barton P Miller, Mark D Callaghan, Jonathan M Cargille, Jeffrey K Hollingsworth, R Bruce Irvin, Karen L Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *Computer*, 28(11):37–46, 1995.

[36] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[37] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[38] Shobana Padmanabhan, Yixin Chen, and Roger D. Chamberlain. Unchaining in design-space optimization of streaming applications. In *Proc. of Workshop on Data-Flow Execution Models for Extreme Scale Computing*, September 2013.

[39] Oliver Pell and Oskar Mencer. Surviving the end of frequency scaling with reconfigurable dataflow computing. *ACM SIGARCH Computer Architecture News*, 39(4):60–65, 2011.

[40] Frances Perry. Sneak peek: Google Cloud Dataflow, a Cloud-native data processing service. `http://goo.gl/WnP27r`, June 2014. Accessed November 2015.

[41] RaftLib. `http://www.raftlib.io`. Accessed November 2015.

[42] James Reinders. *Intel Threading Building Blocks: Outfitting C++ For Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[43] Samza. `http://samza.incubator.apache.org`. Accessed November 2015.

[44] Laura A Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[45] Stack Exchange Data Dump. `http://goo.gl/PBgYvwz`. Accessed November 2014.

[46] Storm: Distributed and fault-tolerant realtime computation. `https://storm.apache.org`. Accessed November 2015.

[47] O. Tange. Gnu parallel - the command-line power tool. *The USENIX Magazine*, 36(1):42–47, Feb 2011.

[48] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In R. Horspool, editor, *Proc. of Int'l Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 49–84. 2002.

[49] TIOBE Programming Community index. `http://goo.gl/S8LD27`. Accessed November 2015.

[50] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[51] Joseph G. Wingbermuehle, Roger D. Chamberlain, and Ron K. Cytron. ScalaPipe: A streaming application generator. In *Proc. Symp. on Application Accelerators in High-Performance Computing*, July 2012.