Improving Performance of Adaptive Component-Based Dataflow Middleware

Timothy D. R. Hartley\textsuperscript{a,b,*}, Erik Saule\textsuperscript{b}, Ümit V. Çatalyürek\textsuperscript{a,b}

\textsuperscript{a}Department of Electrical and Computer Engineering, The Ohio State University, Columbus, OH, USA.
\textsuperscript{b}Department of Biomedical Informatics, The Ohio State University, Columbus, OH, USA.

Abstract

Making the best use of modern computational resources for distributed applications requires expert knowledge of low-level programming tools, or a productive high-level and high-performance programming framework. Unfortunately, even state-of-the-art high-level frameworks still require the developer to conduct a tedious manual tuning step to find the work partitioning which gives the best application execution performance. Here, we present a novel framework, with which developers can easily create high-performance dataflow applications, without the tedious tuning process. We compare the performance of our approach to that of three distributed programming frameworks which differ significantly in their programming paradigm, their support for multi-core CPUs and accelerators, and their load-balancing approach. These three frameworks are DataCutter, a component-based dataflow framework, KAAPI, a framework using asynchronous function calls, and MR-MPI, a MapReduce implementation. By highly optimizing the implementations of three applications on the four frameworks and comparing the execution time performance of the runtime engines, we show their strengths and weaknesses. We show that our approach achieves good performance for a wide range of applications, with a much-reduced development cost.

Keywords: distributed computing, runtime system middleware, heterogeneous computing

1. Introduction

The high-performance computing world has undergone a major paradigm shift. Supercomputers are now inherently hierarchical and heterogeneous \cite{1,2,3} due to the high degree of on-die parallelism in microprocessors, as well as the ever-increasing programming flexibility of accelerator devices such as graphics...
processing units (GPUs) and field-programmable gate arrays (FPGAs). Indeed, as of March, 2012, all of the five fastest supercomputers in the world use multi-core processors and three of the five also use Nvidia GPUs. However, while the raw performance of supercomputers has continued its steady increase, the programmability of such systems is suffering; indeed, programming such systems can be a very tedious task even for expert programmers. The heterogeneity of these systems, in terms of processing capability and network infrastructure, makes their efficient use a major technical challenge. However, even with these challenges, recent research has shown that cooperatively using CPUs and accelerators is an important goal.

Traditionally, applications have been developed with various low-level software tools for high-performance computing (HPC) systems. MPI provides a portable, high-performance message passing semantic for distributed-memory HPC applications, with certain implementations providing support for high-bandwidth networks. With the release of Nvidia’s Compute Unified Device Architecture (CUDA), and later with the standardization of OpenCL, GPUs became capable of performing general purpose computation, without requiring developers to learn to overload graphics APIs such as OpenGL. FPGAs typically have vendor-specific programming environments, but they continue to increase in flexibility and capability. Multi-core, multi-processor machines are commonly used through POSIX threads, OpenMP, and more recently shared-memory frameworks such as Intel’s Threading Building Blocks. These tools are important and useful to the expert programmer, but designing applications solely with these low-level building blocks remains a complex task.

To lower the bar to programming efficient applications for modern HPC systems, high-level programming frameworks must provide a simple but powerful programming API, must easily support processor heterogeneity, and must provide automatic load-balancing. Unfortunately, modern distributed programming frameworks also leave to the developer the task of finding the right data and task granularity for their application. Since distributed applications targeted to heterogeneous systems have many variables (such as processor speeds, node configuration, and network topology) which affect their runtime performance, the size of the data partitions has traditionally been tuned empirically, not only to find the “best” size for each processor type, but also to balance the competing demands of cache locality, network throughput, network latency, and overlap of communication with computation. Using a sub-optimal partitioning can significantly increase the application’s overall execution time. This tedious empirical tuning step must be performed independently for each new application and each new application deployment on a specific system. Therefore, to help developers extract the best performance from their applications, we should...
allow the middleware to partition the data automatically using only limited information from the developer.

Our previous research presented a solution called the Adaptive Partitioning Controller (APC) [7], which simultaneously balanced the load and tuned the data partition size for applications running on distributed, heterogeneous systems; it was implemented in the DataCutter middleware [8]. However, APC has some shortcomings. Using a single APC controller enables a global view of the system, but this centralized mechanism suffers from scalability issues for larger numbers of processors. Using multiple APC controllers achieves higher scalability, but each APC makes its decisions independently, leading to potentially high load imbalance. Finally, by design, APC solely focuses on efficiently using the processors, neglecting the network. In this paper, we address these shortcomings by developing a significantly improved system, APC+. In APC+, one controller runs on each node and is only responsible for scheduling tasks on the local, on-node processors. Thus, we enable good performance modeling and utilization of the local processing units. Global load balance is achieved by a lightweight work-stealing mechanism. Finally, the data transfers are explicitly managed through a dedicated storage layer, which enables high network performance. These improvements allow APC+ to offer enhanced scalability and utility.

We use three general-purpose distributed programming frameworks as points of reference by which to judge our approach: DataCutter [8] with Demand-Driven load-balancing, a coarse-grain dataflow framework, KAAPI [9], an asynchronous task execution framework based on the Athapascan-1 language [10], and MR-MPI [11], a MapReduce framework using an MPI communication backend. We have chosen compare against these systems for several reasons. Chiefly, they lower the bar to developing distributed applications by presenting simple, clean APIs for managing parallel computational resources. Dataflow systems enable parallelism by allowing developers to partition their data and computation along natural application-specific boundaries. Asynchronous task execution frameworks with explicit data dependencies have a simple task execution API and alleviate the difficulties of managing distributed access to shared data objects. And MapReduce systems enable parallelism by using a functional approach, leveraging a very simple distributed hash table for data storage. These frameworks were chosen because of all distributed programming frameworks, they best exemplify the goals we are targeting with our proposed approach, although all three frameworks fall short in some respect or another. These goals are: a simple programming API, automatic data partitioning and load balancing, and some degree of support for processor heterogeneity.

We will focus on three applications, whose computation occurs in a finitely-divisible workspace, namely Synthetic Aperture Radar [12], Biomedical Image Analysis [13], and Black-Scholes [14]. The regular, finitely-divisible application model is common in real-world applications and exposes the simplest model of parallelism. Each application has different characteristics: they differ in initial data distribution, task execution time profiles, and CPU/accelerator speedups. To further understand the performance of the different middleware systems, we
also designed a synthetic application where we can tune the ratio between the size of the input and the amount of computation.

The rest of this paper is organized as follows. We will first describe our proposed adaptive application programming framework in Section 2 and the three comparison programming frameworks in Section 3. Section 4 describes the three test applications used in the experiments. Then we will present an experimental study in Section 5. We discuss related research in Section 6 and finally conclude in Section 7.

2. APC+: Adaptive Component-based Dataflow

Because parallelism, hierarchy, and heterogeneity conspire to reduce the performance of applications developed for modern supercomputers, high-level programming frameworks must explicitly handle these concepts. Since parallelism implies data and computation partitioning, and since a poor choice can adversely affect the overall execution time, the number of partitions and their sizes are important parameters to choose carefully [7]. Additionally, the heterogeneity in raw computational throughput in modern accelerators versus even highly parallel multicore CPUs brings special challenges which require special attention. Moreover, all of the previously discussed distributed programming frameworks require an explicit decision to be made as to the partition size. The range of acceptable partition sizes can be large for complex parallel applications, and thus, the process to find the optimal value for the partition size can be quite tedious. Therefore, this section describes our enhanced Adaptive Partitioning Controller, APC+, a technique for using a feedback loop to automatically tune the data and computation partition sizes for efficient processor utilization, efficient network utilization, and good overall load balance.

2.1. DataCutter: Component-Based Dataflow

APC+ is built with DataCutter, which is a component-based dataflow [15] system, implementing the filter-stream programming model [8]. DataCutter leverages high-performance networks through the use of MPI. In DataCutter, applications are framed as a set of components, referred to as filters, that exchange data through logical streams. A stream denotes a uni-directional data flow from some filters (i.e., the producers) to others (i.e., the consumers). Data flows along these streams in untyped databuffers so as to minimize various system overheads. A layout is a filter ontology which describes the set of application tasks, streams, and the connections required for the computation. A filter can be replicable, if it is stateless; for instance, if a filter’s output for a given databuffer does not depend on the ones it processed previously, it is stateless and replicable. Replicability of filters allows data-parallelism. A placement is one instance of a layout where filters (and eventually copies of replicable filters) are allocated to physical processors. Figure 1 shows an example filter-stream layout and placement. Filter type B in the layout forks to filter types C and D, whose outputs are joined in filter type E. The placement for a particular configuration
uses three nodes, one of which has a GPU accelerator, two copies \(B\) and \(E\), and three copies of \(C\).

### 2.2. APC+ Framework

Built on top of DataCutter, APC+ gains all of the benefits of a high-performance dataflow runtime engine. APC+ automatically tunes the data-buffer size and application data partitioning, which helps schedule an efficient application execution. There are four parts to our method:

- **Performance Model** \([7]\)

  The performance model tracks each databuffer’s execution time for each processor in the system. By continually tracking the execution time of databuffers in the system, we can dynamically find the databuffer size with the fastest processing rate. We must track each processor separately, because processor type (CPU or accelerator), bus bandwidth, and NUMA memory effects can all conspire to alter the processing rate versus databuffer size curve.

- **Work Partitioner** \([7]\)

  By using user-supplied knowledge of the application’s workspace, and the information about the system’s characteristics from the performance model, the work partitioner streams databuffers to the processors with the aim of balancing the load.
• **Distributed Work-Stealing Layer**

With APC+, each node of the system has a controller filter running, which only provides work to the local processing units. It handles all of the performance modeling of the local processors, and manages the partitioning of its own queue of work. All work sharing between the nodes is conducted by a new work-stealing layer, which tightly integrates with the performance modeler and the work partitioner.

• **Storage Layer**

While the performance model focuses on keeping each processor in the system operating at peak efficiency, and the work partitioner and work-stealing layer ensure good load balance across all of the nodes, the storage layer’s goal is to optimize the use of the network interface. By streaming work to remote nodes, and by transferring data in large chunks, the storage layer ensures that the network is used as efficiently as possible. It is important to note that the work-stealing layer simply balances the assignment of work to nodes, while the storage layer actually moves the application’s input data to the appropriate node. So stealing even a large work assignment is actually a lightweight operation, and the actual data is streamed only as fast as the network can handle.

In our previous work [7], we proposed an adaptive databuffer tuning and work partitioning technique, which featured the first two parts, the Performance Model and the Work Partitioner. Here, we present improvements to the technique, the Distributed Work-Stealing and Storage Layers, which make our framework more network aware. Providing large-scale scalability with a single APC filter was impractical, because APC is only indirectly network-aware. Indeed, APC does not make any distinction between communication time and computation time. The main reason is that the network interface is difficult to model effectively within a high-performance dataflow runtime system. Since distributed dataflow runtime systems are not inherently synchronous, it is impossible to instantaneously determine the execution time of any one data transfer, when there are multiple pending data transfers. Therefore, APC cannot accurately predict the data transfer times in a dynamic system, preventing good load balance among the downstream processors. Further, more than one controller must be used when there are upstream data sources on multiple nodes (e.g., pre-distributed input data is read from disks on multiple nodes in parallel). Each APC must be able to send computation to all the processing filters to be able to utilize the distributed system fully. However, each APC is oblivious to its peers, and each of them will send too-small tiles to each processing filter, because they do not have a global picture of the amount of work flowing into the system. This unfortunate consequence induces high network overhead, sub-optimal processing filter efficiency, and high load imbalance.

To solve these problems, APC+ features a work-stealing layer which handles all distributed-memory work allocation and a storage layer which optimizes the use of the network interface. This twofold approach has many benefits: 1) It
allows APC+ filters to transfer input data in sizes which are efficient for the network interface, 2) it decouples the transfer of data and the processing of the data, and 3) it simplifies dataflow application executions which have distributed input data (data can first be processed locally, without forcing a remote operation). Since work is shared at the node level, instead of the filter level, work can be transferred in much larger partitions. Further, the assignment of work to a thief APC+ filter is a lightweight operation, and is separate from the transfer of the application data. This distinction allows the storage layer to stream the data and avoid overloading the network interface, while still allowing for rapid load-balancing among the nodes. To be sure, component-based dataflow with demand-driven load balancing also simplifies distributed work sharing, but the approach adopted by APC+ integrates knowledge about where work is produced and where and how fast work is consumed.

Figures 2 and 3 show simple example placements of APC and APC+. The major difference is that APC is a centralized system, using only a single filter to manage all of the processors in the entire distributed computational resource, whereas APC+ is a decentralized system, using an APC+ filter copy on each node, which is only responsible for driving the processors resident on that same node.

2.2.1. Load-balancing mechanism

APC+ targets applications whose work area is finitely divisible, subject to minimum and maximum databuffer size constraints stemming from application-specific and hardware constraints such as memory size. We assume that there is a monotonically increasing execution time for increasing databuffer sizes for all processor types. APC+ can handle any type of computation that can be expressed as an n-dimensional work area. For instance, the simplest workload is 1-dimensional, and represents a simple reduction or transformation operation on a 1-d vector of input data. Or, an image analysis operation which operates on 2-d images and produces a feature vector represents a 3-d work area. The work area definition simply needs to match the dataflow application’s requirements.

APC+ operates on WorkTiles, which are n-dimensional work area abstractions and are generic enough to allow APC+ to automatically partition the overall application’s work area and submit work to downstream processors. Applications simply send a WorkTile to an APC+ to alert it of work to be performed. Since APC+ is oblivious to application specifics, it is up to the developer to choose a work area which can be interpreted in the application domain. For instance, in a 2-d image analysis application, the work area would consist of a set of WorkTiles pairs representing several images of a certain number of rows and columns. APC+ will recursively break these WorkTiles down into smaller WorkTiles, representing contiguous subtiles of a certain number of rows and columns. The analysis application will then process these image subtiles.

Algorithm 1 shows the main APC+ pseudocode. One APC+ copy will run on each node in the application’s placement. Each APC+ relies on its own
performance model for estimated processing rate information for all of the processors for which it is responsible. To bootstrap the performance model, APC+ looks for the largest minimum tile size for which a minimum time threshold is reached. This helps to alleviate the overhead of processing tiles which are too small. Once this largest minimum tile size is found, the information is exchanged among all of the distributed APC+ copies. This introduces some synchronization, unfortunately, but this step only occurs once per application execution, and the cost is quite small when compared to the extra overhead caused by the databuffer packing/unpacking, network, and work area fragmentation associated with choosing a minimum tile size which is too small.

Following bootstrapping, APC+ enters its main processing loop (line 17 in Algorithm 1), and the overall execution of the application begins. WorkTile sizes are increased exponentially until the maximum-sized WorkTile is reached, at which point the WorkTile size which gives the fastest processing rate for a particular processor will be preferred.

In the situation where an APC+ filter has an upstream data source, the
Algorithm 1: Adaptive Partitioning Controller (APC+)

Function APC(min\_tile, max\_tile, threshold)

1. queue = GET\_MORE\_WORK(steal = queue.\EMPTY());
2. exit\_bootstrap = false;
3. while min\_tile < max\_tile and !exit\_bootstrap do
4.   for all processor p ∈ P do
5.     min\_tile = GET\_NEXT\_BUFFER\_SIZE(p);
6.     SEND\_WORK(queue, min\_tile, p);
7.     exit\_bootstrap = true;
8.   loop |P| times
9.     |P| times
10.    RECEIVE (p, tile, timing, block = true);
11.    ADD\_DATA\_POINT(p, size, timing);
12.    if timing < threshold/2 then
13.       exit\_bootstrap = false;
14.      min\_tile = parallel\_reduce(min\_tile, MAX);
15.    completed\_work = 0; replied = P; p = 0;
16.    global\_end\_time = 0; agg\_proc\_rate = 0;
17. while !SHUTDOWN(queue) do
18.      queue = GET\_MORE\_WORK(steal = queue.\EMPTY(), work\_received, agg\_proc\_rate);
19.      queue = RECEIVE\_STEAL\_REQUESTS(queue, agg\_proc\_rate);
20.      if |replied| = |P| or work\_received then
21.         alloc\_work = PARTITION(queued\_work, queue, agg\_proc\_rate);
22.         replied = ∅;
23.         global\_end\_time = END\_TIME(work\_received, completed\_work, agg\_proc\_rate);
24.         block = (∀p′ ∈ P, \text{BEST\_PROCESSING\_RATE}(p′) > threshold);
25.         if RECEIVE(p′, tile, timing, block) then
26.             ADD\_DATA\_POINT(p′, size, timing);
27.             replied = replied ∪ {p′};
28.             queued\_work[p′] += size;
29.             completed\_work += size;
30.         queued\_work[p] += size;
31.         alloc\_work[p] += size;
32.         p = (p + 1) mod |P|;
33.         p = (p + 1) mod |P|;
34.      if BEST\_PROCESSING\_RATE(p) < threshold then
35.          size = min (alloc\_work[p], GET\_NEXT\_BUFFER\_SIZE(p, global\_end\_time, queued\_work[p]));
36.          SEND\_WORK(queue, size, p);
37.          queued\_work[p] += size;
38.          alloc\_work[p] += size;
39.          p = (p + 1) mod |P|;
main loop will perform a non-blocking read on it each time through the loop, because having accurate knowledge of the global amount of work in the system is key to ensuring a good load balance. If there is no more work to be queued to downstream processors, the APC+ filter will attempt to steal some work from a randomly selected peer. If, when asked, a peer returns that it has no work, that peer is marked such that it is not asked for work again. If all the peers are marked, then a shutdown process is attempted. In shutdown, each peer is asked whether its upstream is closed, and if all of the upstreams (including its own) are closed, then the filter will complete its shutdown. Otherwise, all of the peers with active upstream ports will be marked as eligible to steal from, and the shutdown attempt will be canceled.

Our stealing algorithm is quite simple, although we do leverage the aggregate processing rate of all of the node’s processors to let APC+ filters steal only as much data as they are likely to be able to process. Therefore, the APC+ filter (the thief) making the steal request to a peer (the victim) includes its own aggregate processing rate in the steal request. Then, the victim calculates the correct amount of work to give to the thief. That is, the victim will give the thief work such that both APC+ filters will complete their work at the same time. The victim APC+ filter will send the work to the thief APC+ in as large WorkTiles as possible, since this reduces the network and processing overhead of sending many small WorkTiles.

Periodically, the APC+ filters will exchange the size of any additional work received from an upstream data source, the amount of completed work, and the aggregate fastest processing rate information for its processors. This enables the entire system to have some global knowledge of the estimated end time. When WorkTiles are queued to processors, this global estimated end time is used to provide another opportunity for reducing the size of excessively long-running WorkTiles, to prevent load imbalance.

Using a simple round robin scheme, APC+ sends WorkTiles to each processor having less than $T$ time units of work queued ($T$ defaults to 100 ms). Once the entire range of possible WorkTile sizes has been tried for a particular processor, APC+ switches to sending the WorkTiles which are of a size to gives the highest expected processing rate. APC+ allocates work to each processor such that they finishing executing their queue at the same time. When the remaining work is low, and the WorkTile size suggested by the performance model is too large, APC+ will send a smaller WorkTile (even if the WorkTile sizing function is still in the bootstrap phase). By doing so, we prioritize load balance over processing rate at the end of the application’s execution. This is especially important when we consider that accelerators often have dramatically increased processing rates with much larger tile sizes. For instance, GPUs have a relatively low bandwidth bus connection, but large computational throughput. Thus, their overall processing rates are highest when given a large amount of work, thereby amortizing the cost of transferring the input and output data to/from the device. Frequently (exactly after receiving a reply from all the processors), the amount of work each processor should be given is updated using partition(); this allows APC+ to react quickly to changing system conditions.
The shutdown function uses a master-slave method to reach global consensus that there is no more work to be completed by anyone. Once all of the upstream data sources are closed, and each APC+ has no more work to complete, rank 0 will send a shutdown message to all of the peer APC+ filters, and the application will exit.

Algorithm 2 shows pseudocode for the PARTITION() function which provides the amount of work each processor should be allocated. The function calculates the estimated amount of time each processor’s queued work will take to complete; it uses this information to allocate to each processor enough work to last max_time. It then distributes the remaining work to the processors proportionally to their processing rate. If PARTITION() runs out of work before all of the processors are given sufficient work to match the longest running processor, then that longest running processor will set the application end time anyway; by not actually queuing more than T time units of work to any processor in Algorithm[11] we ensure the load imbalance is no worse than T.

Algorithm 2: Adaptive Dataflow Work Partitioner()

```
Function PARTITION (queued_work, work)
    procrate = \sum_{p \in P} \text{BEST\_PROCESSING\_RATE}(p);
    for all processor p \in P do
        queue_time[p] = \frac{\text{queued_work}[p]}{\text{BEST\_PROCESSING\_RATE}(p)};
        max_time = \max_{p \in P} \{queue_time[p]\};
    for all processor p \in P do
        alloc[p] = \min\{ (max_time - queue_time[p]) \times \text{BEST\_PROCESSING\_RATE}(p), work \};
        work = work - alloc[p];
    for all processor p \in P do
        alloc[p] += \frac{work \times \text{BEST\_PROCESSING\_RATE}(p)}{procrate};
    return alloc;
```

2.2.2. Storage Layer

The new storage layer in APC+ allows the work-stealing mechanism to work efficiently and quickly to balance the application’s workload, even in situations where a large input data source is only resident on one node of the system. Like APC+, the storage layer is distributed over all of the nodes involved in the processing of the application. The lightweight WorkTiles used by APC+ keep track of where the application data is actually stored (in which node’s storage layer), so that remote nodes can request the data directly from the source. The actual application input data can be arbitrarily large (as it is application-specific), and the storage layer serves WorkTiles which contain the data in a demand-driven fashion. When a thief APC+ steals some work from a victim, the victim alerts its local storage which WorkTiles have been stolen.
All of the WorkTiles which have been sent to the thief are put into a transfer queue, and the first WorkTile is sent immediately to the thief’s storage filter. Upon receipt of the first WorkTile by the thief’s storage filter, two things occur: 1) an acknowledgement is sent back to the victim storage filter, and the next WorkTile is sent; and 2) the thief’s storage filter alerts the thief APC+ that work has arrived which can be queued to the processors.

Notice that an APC+ filter running on node 0 might steal work from node 1, while the actual input data is physically resident on node 2 (and is queued to be transferred to node 1). Since the WorkTiles themselves keep track of where the application-specific data is, the transfer queues can simply be canceled and rerouted to node 0 by sending the appropriate messages to node 2. Since the APC+ work-stealing mechanism prioritizes stealing large WorkTiles, and because the storage layer only streams data as quickly as the network can transfer, APC+ exhibits improved network performance, when compared to APC.

The storage layer places no limit on the number of times a WorkTile can be transferred. However, since the work-stealing operation is very lightweight, in most cases the storage layer will not have to transfer WorkTiles over the network more than once. Compute-intensive applications may have more redundant WorkTile transfers than data-intensive applications, because the data transfers will occur faster than the computation. However, the work-stealing layer ameliorates this problem by stealing WorkTiles according to the relative aggregate processing rates of the victim and the thief, such that the correct amount of WorkTiles which balances the load is stolen, and scheduled for transfer.

2.2.3. Support for heterogeneity

APC+’s support for heterogeneity inherits all of DataCutter’s built-in support for heterogeneity, specifically the ability to abstract away architectural details from the application decomposition point of view. However, unlike the other distributed programming frameworks presented here, APC+ partitions the application’s work area dynamically, allowing each processor type to work on WorkTile sizes for which it is the most efficient, providing efficient performance, and good load balance. The efficient management of accelerators, which can sometimes have computational throughput orders of magnitude higher than conventional CPUs on certain workloads, is of critical importance for maximum application performance.

3. Distributed Programming Frameworks

3.1. DataCutter: Component-Based Dataflow

In Section 2 we presented APC+, an adaptive partitioning technique to develop dataflow applications for heterogeneous supercomputers. The middleware used to develop APC+, DataCutter, also has native capabilities to develop dataflow applications, and we have chosen to compare APC+ to the state-of-the-art load-balancing technique included in it.
3.1.1. Load-balancing mechanism

The default DataCutter load-balancing technique is Demand-Driven data-buffer distribution. It balances the load between a number of data consumers, according to their processing rate. By only sending databuffers to each consumer when they request more work, consumers which work faster will get more work to process.

The Demand-Driven mechanism is initialized by each producer sending one databuffer downstream to each data consumer. Data consumers will send a request for more data to a producer when they remove a databuffer from their incoming queue. Consumers will request data from the very upstream filter which produced the consumed databuffer. One can initially send more than one databuffer in order to hide the latency of the send-demand/reply-data handshake in cases where the processing time of one databuffer is insufficient. To further allow intra-node load-balancing in DataCutter, copies of a filter allocated on the same node use a shared queue, and pop databuffers from this queue, instead of a private queue.

3.1.2. Support for heterogeneity

The filter-stream model is ideal for programming for heterogeneous processor types, because the architecture-specific details are hidden inside the filter. Provided the same data interface is used by two filters, they can be used interchangeably. Filter implementations can be specialized for dedicated architectures such as the Cell processor [16], GPUs [2], FPGAs, or SMPs.

3.2. KAAPI: Asynchronous Task Execution

KAAPI [9] is a programming framework tightly integrating the Athapascan-1 [10] language and a runtime system for the development of distributed applications. The Athapascan-1 programming language, through the use of C++ templates, allows users to describe the exact data dependencies between tasks in a dataflow graph. The use of commands to asynchronously spawn tasks at runtime and the use of explicit shared variables, which describe the data flow, allow the application’s task graph to be created at runtime and to be unfolded in a dynamic, and data-dependent fashion.

In order to develop an efficient schedule of how to map this dynamically-created dataflow task graph onto distributed resources, Asynchronous Task Execution frameworks in general, and the KAAPI runtime system in particular, construct at runtime a lexicographic ordering of the tasks to be completed, and use the data dependency information from the user program to determine which tasks are ready to execute. The middleware distributes the ready tasks onto the processors automatically using a work-stealing strategy.

Figure 4 shows a general example of the power of KAAPI’s runtime execution of asynchronously forked tasks, and the implicit ordering of task executions by shared data access. Since developers explicitly define how all shared data objects are accessed (in terms of read, write, or read/write semantics), the runtime engine can keep track of which tasks are ready to execute.
3.2.1. Load-balancing mechanism

KAAPI’s load balancing is achieved through work-stealing [17]. Each processor (e.g., a core in multi-core CPUs, or a logical core in CPUs with symmetric multithreading) involved in the computation has its own thread which manages its stack of tasks to complete. When a processor runs out of tasks to complete, it will attempt to steal tasks from a random processor. The victim processor will then yield some portion of its tasks to the thief processor. Provided that tasks and the processors’ processing speeds are homogeneous, this will achieve excellent load balance. If processing speeds are proportional (heterogeneous, but with a constant scaling factor), then this work-stealing scheme still achieves theoretically optimal load balance [18].

3.2.2. Support for heterogeneity

KAAPI’s support of GPUs is essentially left to the developer. Because KAAPI tasks can be stolen at any time before their execution, there is, by design, no affinity between tasks and any particular computational unit. Therefore, developers must turn to mechanisms to either control access to the GPU through a mutex, or create some affinity between one particular KAAPI thread and the GPU device.

We would like to remark that KAAPI is currently being completely rewritten as XKAAPI [19], which will include native support for GPUs. However, XKAAPI is still in development and currently does not support distributed memory architectures.
3.3. MR-MPI: MapReduce over MPI

MapReduce [20] is a programming framework for distributed applications emphasizing simplicity and accessibility for non-parallel programmers. Developers define sequential primitives which operate on portions of an implicitly-referenced distributed hash table which resides locally on nodes involved in the computation. All parallel communication details are hidden from the user by API calls and occur simply to reorganize the distribution of the hash table in the distributed environment. By providing a simple interface, even novice parallel programmers can leverage parallel computational resources and gain benefits in terms of speed of application execution or scale of problem size.

MR-MPI [11] is a C++ MapReduce implementation using MPI as the distributed communication library. The data in MR-MPI are stored in the internal hash table either as Key-Value pairs or Key-MultiValues tuples and are processed by five main functions. Key-Value pairs are produced by the map() function through a user-defined callback function. The aggregate() function distributes Key-Value pairs to processors and guarantees that pairs with the same key are on the same processor. Key-Value pairs with identical keys are merged into a single Key-MultiValue tuple by the convert() function. Each Key-MultiValue tuple is transformed into (zero, one, or more) Key-Value pairs by the reduce() function based on a user-defined function. Data are redistributed on less processors (typically a single one) using the gather() function. MR-MPI’s API is easy to understand and its core API fits into a single page.

![Figure 5: Example MapReduce Execution](image)

Figure 5 shows a simple example of a MapReduce application running on a distributed cluster of machines. The simplest MapReduce applications only need to define one Map stage, and one Reduce stage. Most MapReduce frameworks (including MR-MPI) have some built-in support for managing file-based input data, to ease the burden of partitioning and distributing multiple files, possibly...
of different lengths, and possibly resident in a non-uniform manner across the set of computational nodes.

Map tasks, therefore, take as input some subset of the globally defined input data space. The Map stage has two goals: to put the input data into the implicitly-held hash map data structure, and to ensure that the size of the key-set is sufficiently large so as a good hash-based load-balancing of the downstream reduce tasks. The number of reduce tasks may not match the number of map tasks; the number can be higher or lower (the latter is reflected in Figure 5). The Reduce stage's goal is to operate on the intermediate results in the distributed hash map and determine some final result, or another intermediate result to be passed to a further MapReduce cycle, for more complex applications.

3.3.1. Load-balancing mechanism

The user is not required to write any code related to partitioning, communicating, sorting, or balancing the application data. Rather, users simply write sequential functions which will operate on portions of their data, and use calls to the MR-MPI runtime system to perform all of the parallel communication. Internally, the load-balance is achieved by partitioning the key space into equal chunks. If the internal hash table’s keys are complex structures, users should supply a custom hash function, which ensures good key distribution over the processors. If applications have unpredictable task execution times, MR-MPI also has a master/slave load balancing mechanism, which can improve load balance, depending on the specific application and system configuration.

3.3.2. Support for heterogeneity

MR-MPI’s support for GPUs is entirely up to the developer to manage. Because MR-MPI is closely linked with its MPI back-end, we can simply use the MPI rank to determine which processes should use the GPU and which should use CPU threads. If cooperative CPU / GPU is desired, then a custom hash function needs to be supplied which gives the GPU ranks a larger share of the intermediate data values. For the best load balance, developers can profile their application kernels, and give the GPUs the correct proportion of the total work, such that they finish at the same time as the CPU threads.

4. Application Descriptions

To test the performance of the three distributed programming frameworks, we have chosen three applications with different characteristics. Each application has a different initial data distribution, task execution time profiles, and CPU/accelerator speedups. This helps to stress the runtime systems tested in our experiments. Each of the selected applications has processing kernels implemented for a single CPU thread and a GPU accelerator device. We reuse these basic kernels for the implementations for each of the three frameworks.

Our first application is a Synthetic Aperture Radar (SAR) imaging application [12]. Radar return signals from multiple viewing angles of the same
scene are combined together to create an image. The simple backprojection algorithm used here is a triple loop where each input vector of radar return data is applied to each pixel of the output image. The inputs to the application are small, so they are broadcasted at the beginning of the application execution. Because the application tasks are independent, and because the task execution times are more dependent on the number of output pixels created than on the number of inputs, we partition the output space into tiles. After processing, the final image is aggregated back on one node, and written to disk. Our output image sizes range from 4K x 8K on one node up to 32K x 32K on 32 nodes. Our CPU implementation is only of reference quality, but its further optimization is outside the scope of this paper. The computation style is very well-suited to the GPU, and we see large speedups (see Section 3 for all CPU and GPU execution time specifics for our platform). The lack of an efficient CPU implementation in specific libraries is common with GPU implementations of applications [21]. This is not a middleware issue and the authors believe runtime systems should cope with it.

Our second application is a **Biomedical Image Analysis (BIA)** application [13]. In this application, highly magnified digital images of specially prepared biopsy tissue samples are processed using cooccurrence matrices and operators called linear binary patterns to determine the texture of the tissue found in the biopsy sample. The texture determination affects the prognosis for the patient. This is a time-consuming, error-prone process for human slide readers to perform, and the goal of this application is to reduce error and increase analysis throughput. The digitized images can be over 100K x 100K pixels, and many slides are often analyzed as part of a patient’s study. In our experiments we have used image sizes ranging from 16K x 32K pixels for a single node to 128K x 128K pixels for 32 nodes. For our experiments, the input image begins already decompressed in the memory of a single node (or two nodes, if it would not fit into a single node’s main memory). This input image is tiled and sent to computational nodes for processing, and a feature vector for each tile of 13 single-precision floats is returned. The processing requirements are a good match for the GPU’s many-core architecture and high memory bandwidth; hence, high speedups can also be seen, as compared to a single-threaded reference CPU implementation can be obtained.

Our last application is **Black-Scholes**, from the Nvidia CUDA SDK. It is a stock market option pricing application which is designed to model the way options are sold on the market, and to predict when buying and selling these options would be advantageous. Each option consists of three 32-bit floating point values, and the output calculated by the application is two 32-bit floating point values, which represent the “call” and “put” prices of the option. To make this simple calculation data-intensive, our input dataset includes $2^{27}$ options per node to calculate. On 32 nodes, this means that there are 4 billion options to process. Options are processed in tiles. Because the computation is relatively simple, this application has a high communication to computation ratio. This application is very I/O-bound. Therefore to provide the maximum disk bandwidth, we partition the input data set over the local disks of all the
nodes involved in the experiment, as part of an untimed setup phase. Thus, when using 2 or more computational nodes, the input data may be read concurrently from one disk of each of these nodes. The output of the computation is aggregated on a single node, although not written to disk.

5. Application Experiments

5.1. Experimental Setting

Our experiments were run on Owens, the new GPU cluster in the Department of Biomedical Informatics at The Ohio State University. For our experiments, we used 32 computational nodes, each with dual Intel Xeon E5520 Quad-core CPUs (with 2-way Simultaneous Multithreading), 48 GB of memory, and 500 GB of scratch disks. The nodes are equipped with Nvidia C2050 Fermi GPUs, each with 3 GB of memory, and are interconnected with 20 Gbps InfiniBand. Our experiments were run on CentOS with the 2.6.18-194.8.1.el5 Linux kernel, and compiled with GCC 4.1.2 using the -O3 optimization flag. Our GPU code was compiled with CUDA 3.1. While DataCutter and MR-MPI can use an InfiniBand-aware MPI such as MVAPICH, KAAPI is not InfiniBand-aware. Therefore, to provide a meaningful comparison, we have run all of our DataCutter and MR-MPI experiments using MVAPICH2 1.5 over TCP (TCPoIB) as well as InfiniBand (IB). Our experimental results show the difference between the IB and TCPoIB runtimes as an explicit overhead. As a point of reference, for our cluster, using IB nets 800 MB/s point-to-point bandwidth vs 110 MB/s for TCPoIB.

In all our experiments, we performed a parameter sweep to select the tile size and thread configuration that lead to the best performance separately for each runtime system, except for APC+ (since it requires no developer tuning). We tuned each application for the three remaining runtime systems by choosing the configuration which gave the best performance for the majority of different numbers of nodes. Detailed discussions of why a specific tile size and thread configuration are chosen over a different configuration are out of the scope of this paper; rather, we will simply be satisfied that the configurations for each runtime system are as optimized as they can be, and consider the application kernels to be black boxes, except for the tunable parameters they expose. The importance of properly selecting the tile size has been shown in [7], where Figure 6 shows that APC performs within 10% of the optimal performance in 90% of the test cases. The second best is a Demand-Driven mechanism, manually tuned per application, which performs within 25% of the optimal on a comparable number of instances. All tests performed with a fixed tile size performed dramatically worse.

Table 1 shows the execution times using only one CPU thread and using only one GPU of all three applications for the tiles sizes which were found to give the fastest performance, according to our parameter sweep.

The next sections present the results of the experiments for our three applications. To best show the scalability of each system, our experiments are
weak scalability experiments: we increase the amount of work in commensurate amounts when we increase the numbers of nodes. The volume of data per node for each application is given in Section 4. To aid in the clarity of the charts, we have broken down the execution times into several categories. There are five main categories which are present in the charts of all three applications, PROC, L-IMB, OVER, BOOT, and TCP. The PROC portion of each bar represents the minimum processing time of all of the threads, while L-IMB is the load imbalance of the system (calculated by subtracting the minimum processing time of all of the threads from the maximum). The OVER category is a catch-all category, and includes communication, and other runtime engine overheads. Finally, BOOT represents the amount of time to bootstrap the runtime engine itself, including remote process invocation, etc, and TCP represents the difference in the total runtimes of the TCPoIB and IB experiments for DataCutter and MR-MPI. Note, therefore, that the breakdown of the overall TCPoIB execution time into the sub-categories may be different from that of IB. On each chart, the y-axis gives the runtime of the configuration in seconds, while the x-axis gives the runtime system used and the number of nodes it ran on.

5.2. APC and APC+ Comparison

This section briefly compares the performance of APC and APC+ while running experiments on our three applications over the IB network. Figures 6, 7, and 8 show the CPU-GPU results for APC and APC+ for SAR, BIA, and Black-Scholes, respectively. Recall that the experiments show weak scalability: linear scaling is achieved when the execution times are flat, because extra work and extra computational nodes are provided in equal measure. We have chosen to restrict this comparison to the CPU-GPU results, to avoid belaboring the point: scalability on system configurations with higher numbers of nodes is poorly affected by using a centralized work scheduler.

In SAR (Figure 6), there is a large initial work area (although with relatively small input data), and the different processor types have drastically different processing rates types. APC suffers in this instance because while it tries to balance the load, even a single too-long tile being executed on a single CPU can cause serious load imbalance. As one can see, APC+’s execution times are nearly

<table>
<thead>
<tr>
<th>Application</th>
<th>Tile Size</th>
<th>1 CPU thread</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAR</td>
<td>128</td>
<td>3.354.1</td>
<td>21.5</td>
<td>156.1</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>53.792.3</td>
<td>182.5</td>
<td>294.7</td>
</tr>
<tr>
<td>BIA</td>
<td>512</td>
<td>213.8</td>
<td>8.2</td>
<td>26.0</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>857.6</td>
<td>22.3</td>
<td>38.4</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>3,433.6</td>
<td>96.0</td>
<td>35.8</td>
</tr>
<tr>
<td>Black-Scholes</td>
<td>64K</td>
<td>19.6</td>
<td>0.8</td>
<td>24.4</td>
</tr>
<tr>
<td></td>
<td>128K</td>
<td>34.7</td>
<td>1.4</td>
<td>25.3</td>
</tr>
</tbody>
</table>

Table 1: Execution times (in milliseconds) for SAR, BIA, and Black-Scholes for various tile sizes
flat, showing near linear scalability up to 32 nodes. Interestingly, APC+ also shows some load imbalance, although this is a direct side effect of the dramatic GPU speedup; APC+ keeps the CPUs slightly starved, in order to keep one too-long-running task from setting the overall execution time. This ability to manage to give even applications with dramatically different tile execution times on heterogeneous processors good load balance is a key feature of the updated Work Partitioner component in APC+. We shall see in Section 5.3 that other distributed programming frameworks cannot match the performance of APC+, even when well-tuned.

In BIA (Figure 7), a large input image starts on one node (or two nodes for the 32 node experiment), which causes a major bottleneck in the network endpoint, which is an impediment to effectively distributing the input data to the whole cluster. Worse, the smallest tile size used in our experiments (32 x 32 pixels) leads to a very small 3 KB databuffer, which results in poor IB performance. This runtime effect of the network contention is visible both in the overhead category and as a side-effect in load imbalance. While executing with 16 nodes, a 64K x 128K image is 24 GB, meaning that it takes over 28 seconds merely to transfer an equal 1/16 portion of the image to each remote node. However, because APC+ has explicit optimization of the network interface due to the Storage Layer, its performance degrades the more gracefully than APC, and the most gracefully of the other frameworks we compare to in Section 5.4.

Black-Scholes (Figure 8) has distributed input data streaming into the system from every disk, which can lead to a situation where a centralized controller, responsible for every node in the system, can never increase the databuffer size sufficiently to reach high-performing tiles. In the case of Black-Scholes, all of
the computation time can be hidden by reading the input data off of the disk, but when the tile size is too small, the network becomes a major bottleneck, especially for the aggregation down to one node. As one can see from Figure 8, APC+ also has issues with the one node aggregation on 32 nodes, but as we will see later in this section, this is simply caused by endpoint contention on the aggregation node. However, APC+ notices the bottleneck caused by poor network performance and only lazily transfers work, such that more of the work is done by the node on which the input data initially enters the system. As you can see, the performance of APC+ is substantially better than that of APC in the case of 16 and 32 nodes.

5.3. Synthetic Aperture Radar

Figures 9 and 10 present the experimental results for the SAR imaging application for all four programming frameworks for, respectively, CPU-only and CPU-GPU configurations. For the CPU-only experiments, APC+ uses CPU 15 threads per node, because one thread is taken up by the APC+ functions. The remaining frameworks all use 16 threads. Please note that, the results shown for DC-DD, KAAPI and MR-MPI use tile size that have been manually tuned to obtain the best performance at the most scales. DC-DD performs best with tiles of size 128 x 128, while KAAPI and MR-MPI perform best with 512 x 512 tiles. For the CPU-GPU experiments, APC+ uses 14 CPU threads and the GPU, while the rest of the CPU-GPU configurations for SAR use the GPU and no CPU threads, reflecting the large GPU speedup. DC-DD, KAAPI and MR-MPI obtain the maximum performance when a tile size of 512 x 512 is selected.
This application is very computationally intensive, and the CPU kernel implementation is fairly inefficient, accounting for the excellent CPU scalability for APC+, DC-DD, and KAAPI up to 32 nodes, and for MR-MPI when using pure IB visible on Figure 9. The slowdown for MR-MPI is related to the relatively large image size formed by the application on 32 nodes. A 32K x 32K complex-valued single precision image is 8 GB, which takes some time to transfer over the inefficient TCPoIB protocol. Since this communication is not overlapped at all by MR-MPI, it is visible as overhead in the TCP category. Each of the frameworks reports some degree of load imbalance. However, it is always less than the execution time of a single tile.

Our GPU filter implementation for SAR, on the other hand, is quite efficient, leading to much faster execution times, and regrettably, poor scalability for all of the frameworks with TCPoIB (which can be seen on Figure 10). Notice that APC+ and DC-DD are able to keep up with the increasing computational demand and scheduling complexity when using IB, however. The large image size on higher node configurations causes large execution time penalties, in the TCP category for APC+ and DC-DD, and in the OVER category for KAAPI and MR-MPI.

KAAPI is the only framework which shows significant load imbalance in the GPU case when using more than 4 nodes. This is caused by the requirement to gather the output image onto a single node before exiting the application. (Recall that the image is not written to disk, but simply gathered on one node.) The gathering of the output is implemented using remote iterators initialized on the node where the data need to be gathered. The data transfers involving...
remote iterators are not overlapped with communication. Therefore, the node which owns the data executes significantly more computations than the other ones, leading to a large load imbalance. Unfortunately, this load imbalance also causes most of the time accounted for in OVER. When a thread on the node that owns the data no longer has work in its queue, it tries to steal work from a remote thread. However, the network stack and hardware are congested by the remote iterator transfer, making the total time of each stealing operation more than one second on 16 processors. This idles the node that owns the data for a significant amount of time at each stealing operation. Further, it is important to notice that KAAPI’s remote iterators can be read-only or read-write but not write-only. Therefore, to write the output data for each tile, the uninitialized tile data must be read first, doubling the total amount of data transfer and worsening the network congestion in Figure 10. However, while this is an unfortunate consequence of the remote iterator approach, the alternative is to include the input data in the instantiation of the task, and to repeatedly copy it and split it into parts, as the recursive task generation model dictates. Using this slower method can dramatically damage application performance, worsening by up to 20x in our experiments.

Most of the time spent in MR-MPI is accounted for in OVER. This can be decomposed into two parts, a constant part and a part proportional to the number of processors. The constant overhead consists of the requirement to simply distribute the small input data, and the requests to each node for what portion of the output image they have to calculate. Indeed, these two steps take over 24 seconds to complete. DC-DD and KAAPI allow asynchronous
task execution and the processing can start immediately. However, MR-MPI is synchronous and must wait for all the data transfer to finish. The variable part comes from the aggregation of the data to one node, which must occur for all three middleware systems. However, the synchronized behavior of MR-MPI forces the communications to be serialized after the computation, worsening the network contention.

Thanks to DataCutter’s filter-stream programming model, both DC-DD and DC-APC+ properly overlap communication and computation leading to significantly faster execution in the CPU-GPU case. Additionally, the ability of DC-APC+ to accurately model the overall application end time allows better dynamic choice of tile sizes, leading to better load-balance, as is particularly noticeable in 32 node values of Figure 9.

5.4. Biomedical Image Analysis

Figures 11 and 12 present the experimental results for the biomedical image analysis (BIA) application for, respectively, CPU-only and CPU-GPU configurations. Again, APC+ uses 15 CPU threads during the CPU-only experiment, and 14 CPU threads and the GPU during heterogeneous configurations. All the other three frameworks’ CPU-only experiments are fastest when using 16 threads per node, when running without the GPU. DC-DD and KAAPI use a 1024 x 1024 tile size, while MR-MPI use a 512 x 512 tile size. In the mixed CPU-GPU experiments, DC-DD and MR-MPI are fastest when only using the GPU, while KAAPI is able to make use of 7 CPU threads as well as the GPU, with a 512 x 512 tile size. DC-DD uses a 2048 x 2048 tile size, while KAAPI and MR-MPI use a 1024 x 1024 tile size.
Unfortunately, unlike the SAR application, none of the runtime systems scale linearly beyond 4 nodes when using TCPoIB, even on the CPU implementations. Further, APC+, DC-DD and KAAMI all suffer from an increasing degree of load imbalance on high numbers of nodes. A bottleneck must account for the loss of scalability. In this case, it is the network interface contention on the “storage” node (or storage nodes in the case of 32 nodes), where the input data is resident. Since there are 16 threads per CPU processing tiles, and 31 nodes in the system making requests of the network endpoint, at 857 milliseconds per tile, this would require 1820 MB/s network endpoint bandwidth. The cluster has 20 Gbps InfiniBand, which gives an effective bandwidth around 800 MB/s. MR-MPI does not show any load imbalance: since the work is statically partitioned, all the processors performs the same number of operations.

However, we see that DC-APC+ is able to scale near linearly using the IB interface while running with CPU threads only. Unfortunately, the load imbalance gets the better of it on high numbers of nodes when using a heterogeneous system. Since the CPU runtimes for even large tiles are relatively small for BIA, a well-tuned Demand-Driven execution will be hard to beat outright, but APC+ is able to reach nearly the same performance level, without the laborious tuning step.

The major slowdown of KAAMI is again due to KAAMI’s inability to overlap communication with computation. All of the latency is visible in the L-IMB execution time category, and is visible in the OVER time category, while DC-DD does a good job of overlapping communication with computation thanks to the dataflow paradigm.

The MR-MPI runtime system’s performance is not linearly scaling for a
variety of reasons. First, since MR-MPI cannot overlap communication with computation, all of the data transfer time is seen directly in the OVER category. When using 32 nodes, our image size is 128K x 128K pixels, with three color channels, leading to a 48 GB image. Second, to stay within the memory constraints of the node, we need to use working sets which fit inside the memory of a single MPI process. We can asymmetrically allocate memory to the MPI processes which share a node, such that the process responsible for loading and distributing the input data can get the lion’s share of the data, but MR-MPI allocates 7+ pages of memory (according to the documentation) for parallel communications, requiring the working set of the application to be artificially small. Thus any overheads to set up and tear down these parallel communications are paid more than necessary.

5.5. Black-Scholes

Figures 13 and 14 show the CPU-only and the CPU-GPU results for the Black-Scholes application, respectively. All of the CPU-only implementations use 8 CPU threads, and all of the CPU-GPU implementations use 1 GPU thread only, except for DC-APC+, which uses 7 CPU threads and the GPU. DC-DD’s CPU-only and KAAPI’s GPU implementations are fastest at most scales with tiles of 128K options, while the rest of the implementations are fastest at most scales with 64K option tiles.

Like many real-world applications, the Black-Scholes application is bound entirely by disk read bandwidth, because the amount of computation per option is relatively small. Thus, even though our implementation streams data in tiles
to the processing tasks from the scratch disk of the nodes, there will be some amount of disk access time which is not overlapped with computation. The INIT time presents this value: it represents the maximum of all of the time each read task takes to read the options from disk and store them in memory minus the maximum processing time of a processing thread. The OUT portion of each bar refers to the non-overlapped time to aggregate the output data on one node (but not actually written to a file - the application is already disk-bound, we do not need to belabor the point).

There are several notable trends in Figures 13 and 14, which we will discuss in turn. The most obvious fact is that all of the runtime systems perform terribly when using TCPoIB. The INIT time for DC-APC+, DC-DD and MR-MPI stays constant when the number of nodes is increased, meaning that while the size of the disk-based input data is increased, the disk read bandwidth is also increased, and the INIT execution time component is not worsened. However, perfect scalability is not achievable in Black-Scholes, since all the results are gathered on one node, and this operation is limited by that node’s network bandwidth.

DC-APC+ using IB scales the best, followed by DC-DD. When using IB, DC-APC+ slows down by a factor of 2.6 when increasing to 32 nodes. DC-DD slows down by 2.7 times, KAAPI is 11.4 times slower on 32 nodes, and MR-MPI only slows down by 4.9 times, although it is over twice as slow on one node as DC-APC+. When using TCPoIB, however, KAAPI scales better, beating DC-DD’s performance on 8, 16, and 32 nodes. While not surprising, efficient use of a high bandwidth network is important. DC-DD’s Demand-Driven policy induces communication from each producing filter to each processing filter. Since the
DC-APC+ removes the all-to-all communication by decoupling the management of the distribution of the work and the distribution of the data. Thus, the Distributed Work-Stealing Layer is able to quickly load-balance, because the assignment of future work is a lightweight operation. Databuffers are exchanged in large blocks, maximizing network performance, thanks to its Storage Layer. Both components contribute to providing and efficient load balance and data distribution that take into account the hierarchical nature of clusters of heterogeneous, multicore nodes.

The most noticeable difference between KAAPI and the other runtime systems is the growing INIT time. Indeed, KAAPI is not able to place data input read tasks on the same node where the data resides; we therefore incur a large latency and disk and network contention penalty by reading the data across the network through a parallel file system of any type. The decrease in non-overlapped INIT time beyond 2 nodes is due to there being less disk contention and more aggregate disk bandwidth than on 2 nodes. KAAPI’s load imbalance is once again due to the lack of communication and computation overlapping.

MR-MPI’s poor scalability performance is caused by not overlapping communication and computation, and by MapReduce’s load-balancing technique. On 32 nodes, while using IB, MR-MPI spends 57.8 seconds communicating input data to the processing tasks, and 92.8 seconds gathering the output data to a single node. When using TCPoIB, MR-MPI spends 302.3 seconds distribu-
ing the input data, and 607.0 seconds gathering the output data to a single node. INIT takes more time in MR-MPI than in DC-DD because of the synchronization of MR-MPI that prevents the overlapping of reading the files and of their processing. An overhead similar to the one shown on BIA is present in Black-Scholes as well.

5.6. Tunable CCR

Finally, Figure 15 shows a strong scalability experiment where we choose a static system configuration, and vary the relative amounts of communication and computation. Our tunable CCR application is based on Black-Scholes: the amount of computation is increased by executing several iterations of the Black-Scholes kernels for each options. If the number of iterations is set to 1, the tunable CCR application and Black-Scholes are exactly the same. The chart shows the 16-node CPU speedups (calculated by dividing each framework’s 16-node time at each iteration point, by the fastest 1-node execution time for that iteration point) as a function of the number iterations on the Black-Scholes kernel per option. As such, for low values of number of iterations, the application is heavily data-intensive, and as the number of iterations increases, the application becomes compute-intensive.

While it appears that there is some super-linear speedup for DC-APC+ and DC-DD for 75 iterations and above, this is merely a consequence of the additional disk bandwidth provided the application when running on 16 nodes. The execution time on 16 nodes is never more than 3% faster than the numerically derived best-case execution time extrapolated from the single-node execution time for the fastest framework.

When the number of iterations is 1, DC-APC+, DC-DD and KAAPI have a low speedup of 2 due to the long time spent aggregating the results on one node. When the number of iterations increases, the time spent to aggregate becomes smaller (in relative terms) and the speedup increases. DC-DD and DC-APC+ achieve linear speedup when the number of iteration is greater than 75. KAAPI scalability does not increase as well, since it achieves a speedup less than 14 for all of the numbers of iterations less than 250. We believe KAAPI’s speedup would likely reach 16 if we had run larger experiments.

MR-MPI does not show great speedup. Its speedup increases when the number of iterations increases, although even when the kernel of Black-Scholes is executed 250 times per option, the speedup of MR-MPI stays less than 2. This is caused by several factors. First, by examining Figure 13 we see that MR-MPI begins considerably slower on 16 nodes than the fastest framework. When running on TCPoIB, the MR-MPI 16-node execution time is 470.0 seconds, while DC-APC+ runs in 152.5 seconds. Thus, MR-MPI is at an immediate 3x deficit. Additionally, MR-MPI serializes the reads from disk and the processing of the options. Thus, any increases in the processing time of the options are immediately visible. The other frameworks are able to hide the increase in processing time until it exceeds the disk read time, at which point the overall execution time begins to increase.
6. Related Work

There are many research projects investigating automatically optimizing the performance of software which are similar to our work. FFTW [22] is well-known as a library for portably generating high-performance FFT implementations on a variety of platforms. ATLAS [23] uses compile-time performance tests to tune its implementations of linear algebra operations. In [24], the authors discuss adaptive techniques for improving the performance of MPI collective communication operations. Thus, automatic tuning for a variety of fine-grained kernels and for general communication patterns is not new; however, our approach begins to develop a runtime framework which can automatically tune and balance the load of general dataflow applications, provided they fit the application model.

The mapping and scheduling of static or dynamic programs onto heterogeneous resources also informs this work [25]. However, our work differs from this field because the focus in that field is on the constrained scheduling of tasks with heterogeneous execution times or hardware capabilities. These algorithms do not attempt to subdivide tasks or data in any way; the applications are considered immutable input.

The closest related theoretical work is the Divisible Load Scheduling (DLS) model [26]. The goals in DLS are to determine what fraction of a large input dataset should be processed by each processor in a system, and how to schedule the communications, under various models of the capabilities of the endpoints and the network. Models in this area usually consider large time intervals so that any discrete effect in the computation can be omitted. There-
fore, the models often consider the platform to be static and the buffer of the processors to be infinite. The beginning and ending of the computation are also frequently neglected. However, newer work such as [27] introduces heuristic solutions for scheduling communications and computation, including the collection of the results of the computation. Our work differs from this work in two main respects. DLS requires accurate knowledge of the system’s computation and communication rates for scheduling. Our intent is to provide a real system for dataflow application development, without requiring developers to perform any tedious parameter discovery themselves. DLS also focuses on the steady-state, whereas real-world applications can perform all of their work without reaching any steady-state condition.

Software runtime systems which adapt the work partitioning between heterogeneous processors on a single node are also closely related to our work. Qilin [4] conducts a static work partitioning between a multicore CPU and a GPU by initially executing training runs with various work sizes for each processor type, while Harmony [28] adaptively subdivides the application tasks to the CPU and a Field Programmable Gate Array at runtime. Both of these systems use the CPU and an accelerator in concert to achieve faster processing than by using either alone. GPUs [29] is an OpenMP-like programming framework which automatically creates parallel code for multiple GPUs from annotated C code. By using knowledge about the data locations, tasks can be scheduled to run on the correct GPU without excessive data movement. StarPU [30] is a single-node dataflow programming framework and runtime engine which uses work-stealing to balance the workload. All of these systems only target single computational nodes, whereas our target is to handle distributed resources.

The field of middleware runtime systems has a long history of developing techniques for balancing the load of distributed applications at runtime. River [31] uses a distributed queue and a simple load balancing technique for producer-consumer application task relationships. Anthill [5] features algorithms for scheduling dataflow applications where the amount of work is not known prior to runtime. Coign [32] focuses on partitioning the dataflow graph to perform a static runtime data decomposition, whereas Cilk [17], Charm++ [33], ACDS [34], and TelegraphCQ [35] perform automatic load balancing as well as application partitioning. Capsules [36] allows developers to tune their application’s task and data granularity with a simple parameterization scheme. More recently, MapReduce frameworks for heterogeneous systems such as Merge [37] have been designed. In these systems, a manual data-granularity tuning step must be performed to achieve the best application performance. Our framework, APC+, automatically performs this tuning step at runtime. Thus, our work is complementary to the research in this field, and indeed, could be directly implemented in many of these dataflow runtime systems.

The family of Partitioned Global Address Space (PGAS) languages such as Titanium [38], X10 [39], and Chapel [40] are powerful languages for programming high-performance distributed applications. However, all of these languages leave the important load-balancing problem up to developers to solve, albeit with a concise programming methodology.
There are many other instances of runtime engines we could have chosen to include in our comparison. Cilk \cite{cilk} is an obvious choice, due to its powerful work-stealing runtime engine, and although a distributed version of Cilk, called Cilk-NOW \cite{cilknow} (Network Of Workstations) is referenced in the literature, it is not available for use. The most famous publicly available MapReduce implementation is Hadoop\cite{hadoop} but its focus is incorrect for our work; Hadoop is an enterprise-grade framework designed for out-of-core operation and fault-tolerance, and relies heavily on a parallel file system which cannot be bypassed. Charm++ \cite{charm++} is an object-oriented distributed programming framework which associates computation to specific object instances, and migrates these objects to balance the computational load. However, adding Charm++ to our discussion would not increase our coverage of the distributed runtime engine space significantly beyond what we already present. While Charm++ provides a number of load-balancing techniques, including a measurement-based technique, a manual tuning step to determine the optimal data domain decomposition is required. Additionally, providing load balancing for GPU-based computations is a non-trivial problem, since object state must be check-pointed in order to allow migration from node to node.

7. Conclusion and Future Work

Writing distributed applications for high performance computing systems is becoming more complex. The tremendous number of parameters involved in distributed application design makes their programming and tuning a tedious task. Numerous programming models and middleware frameworks have been proposed to lower the programming complexity, while still achieving good parallel performance, but none of them have attempted to lessen the burden on the developer to find the optimal data granularity to achieve the best performance on a target system. Thus, we have developed APC+ to be simple, highly efficient, and to ease the developer’s task in creating high-performance applications. APC+ achieves good load balance through the use of a work-stealing engine, while still providing efficient network usage by using a dedicated storage layer. It adaptively partitions the application workarea to optimize the performance of the processors, by modeling each task’s execution profile. We experimentally showed that APC+ is able to meet or beat the performance of well-tuned implementations on a dataflow framework with Demand-Driven load balancing, an asynchronous task execution framework with work-stealing, and a MapReduce framework.

It may seem biased to compare APC+ to frameworks without native support for heterogeneous processors or communication and computation overlap. Indeed, these are important facets for modern distributed programming frameworks to exhibit. However, as discussed in Section \ref{related_work} there are no modern

\footnote{http://hadoop.apache.org/}
distributed runtime systems with a simple programming API, built-in communication and computation overlapping, good support for heterogeneous processing rates, and automatic load balancing.

Further, these frameworks represent a good sample of the state-of-the-art in high-level distributed programming frameworks in recent literature. Dataflow is a stalwart alternative to message-passing systems, asynchronous task execution has been chosen by DARPA as the next-generation in high-level programming languages for developer productivity, and MapReduce has become wildly popular because of its ease of use for analyzing large data sets simply. Indeed, for control-heavy parallel programming, such as the implementation of branch-and-bound algorithms, an asynchronous task execution framework is likely a better fit than a dataflow system. And MapReduce is a good choice as an entry-level parallel programming framework, due to its simplicity and focus on sequential processing. Unfortunately, while these frameworks have their strengths, even their best representatives suffer performance degradation on simple applications.

APC+ gracefully handled many of the issues that appeared on the other test cases without introducing new corner cases. Some of the main problems we encountered were the lack of overlapping communication and computation and the lack of asynchronicity of the computations, which are both natively provided by the filter-stream programming model used in DataCutter. The dynamic tile sizing and the estimation of the computational requirements, which were introduced in APC, allows the estimation of the overall computation time, allowing close to perfect load balance even using highly heterogeneous processing units. All to all communication is avoided by APC+, thanks to its hierarchical work distribution. Network performance in general is harmed by small and frequent messages; hence, the storage layer of APC+ is designed to perform communications in large chunks, thereby ensuring efficient use of the network resource.

As future work, we intend to bring our adaptive dataflow solution to bear on other application types, such as iterative solvers, data-mining, or applications with dependent tasks. We also intend to investigate supporting other programming models in our framework, such as MapReduce, to extend the technique’s applicability into other application domains.

Acknowledgment

This work was partially supported by the U.S. Department of Energy SciDAC Grant DE-FC02-06ER2775, NSF grants CNS-0643969, OCI-0904809 and OCI-0904802 and AFRL/DAGSI Ohio Student-Faculty Research Fellowship RY6-OSU-08-3.

The authors would like to thank the reviewers for their valuable comments, which helped us improve the presentation of this paper. The authors would also like to thank Jean-Noël Quintin for his help using KAAPI, as well as Steve Plimpton and Karen Devine, for their help using MR-MPI.
References


35


Object-oriented programming systems, languages, and applications (OOPSLA 1993), pp. 91–108.


