A Component-Based Framework for the Cell Broadband Engine*

Timothy D. R. Hartley, Umit V. Catalyurek
Department of Biomedical Informatics,
Department of Electrical and Computer Engineering,
The Ohio State University, Columbus, OH, USA.
{hartleyt,umit}@bmi.osu.edu

Abstract

With the increasing trend of microprocessor manufacturers to rely on parallelism to increase their products' performance, there is an associated increasing need for simple techniques to leverage this hardware parallelism for good application performance. Unfortunately, many application developers do not have the benefit of long experience in programming parallel and distributed systems. While the filter-stream programming paradigm helps bridge the gap between developers of scientific applications and the performance they need, current and future high-performance multicore processor designs do not have a filter-stream programming library available. This work aims to fill that gap in the software world. This initial DataCutter-Lite implementation defines a powerful, but simple abstraction for carrying out complex computations in a filter-stream model. Additionally, the initial implementation shows that complex architectures such as the Cell Broadband Engine Architecture can make use of the filter-stream model, and give good application performance when doing so.

1 Introduction

Designing parallel and distributed programs for efficient execution on large, complex supercomputers is a challenging task. Experts in fields such as physics, image and signal analysis, and biology are ill-equipped to design applications to take full advantage of the hierarchical, heterogeneous, distributed cluster supercomputers which are quickly becoming the norm for high-performance computing resources. Microprocessor architectures like the Cell Broadband Engine Architecture and Graphics Processing Units have many unique characteristics constraining their use, not to mention their own, difficult-to-learn and harder-to-master application programming interfaces. This paper presents DataCutter-Lite, a fine-grained filter-stream programming library and runtime system which enables the simple design of filter-stream applications for modern multicore processors. We show that by using DataCutter-Lite, developers can leverage modern, heterogeneous multicore processors with good efficiency and productivity.

In order to fully use the computational power of modern multicore processors, developers must be well-versed in:

- Parallel programming techniques, such as data decomposition, master-slave and pipelined computations.
- Parallel algorithms, if basic operations such as search and sort comprise a large part of the computation the developer wishes to perform.
- A threading library, such as POSIX threads.
- Architecture-specific constraints, such as memory size, or the myriad of concerns associated with the Cell Broadband Engine:
  - The Memory Flow Controller, used to transfer data in the system.
  - Techniques to deal with the small memories of the Synergistic Processor Elements, such as double-buffering.
  - Distributed memory programming models.
- The developer's own domain of expertise.

Conversely, the filter-stream programming paradigm is simple to learn, since the application implementation's components match natural divisions in the application task structure. Developers gain a large amount of flexibility and power by explicitly describing these application tasks. By defining an application programming interface and runtime engine for filter-stream programming on multicore

*This work was supported in part by the U.S. Department of Energy SciDAC Grant DE-FC02-06ER2775; by the U.S. National Science Foundation under Grants CNS-0643969, CCF-0342615, and CNS-0426241, CNS-0403342; by Ohio Supercomputing Center Grant PAS0052; by AFRL/DAGSI Ohio Student-Faculty Research Fellowship RY6-OSU-08-3.

Table 1. High Performance Computing (HPC) techniques and DataCutter-Lite's approach

HPC Technique	DataCutter-Lite's Method
General HPC techniques (data blocking, communication overlap)	Non-blocking buffer writes and appropriate buffer sizing
Parallel programming and distributed address-space programming	Application-specific task decomposition and componentization
Specific architectural idiosyncrasies (messaging libraries, etc.)	Efficient lower-level libraries enabling consistent higher-level interface

processors, we can bridge the gap between the developers who have need of their high computational power and the breadth of knowledge and experience required to write efficient applications for them. Table 1 presents the high-performance computing techniques necessary for efficient use of multicore processors and our approach to incorporating them into the DataCutter-Lite runtime system and software library. Filter-stream programming meets the needs of performance-seeking, non-computer-scientist programmers for a number of reasons:

- A filter-stream programming paradigm is appropriate for many diverse types of applications, and at many data granularities. The use of this paradigm eliminates the need to use multiple complex parallel and distributed programming techniques. Further, since filter-stream programming inherently allows applications to run on parallel and distributed systems, it is appropriate for the large-scale, complex requirements of many data-intensive applications.
- Filter-stream programming is inherently component-based, and as such, the development of standard algorithms is a natural method to achieve high productivity.
- No knowledge of threading or message-passing libraries is required for filter-stream programming; all of the lower-level threading and message-passing functions are handled by the runtime system.
- All architecture-specific constraints are abstracted away, leaving the application developer with a clean interface and programming semantic. However, should some later developer with good knowledge of the internal workings of the architecture wish to make some modifications, the system is compatible with all of the standard optimizations that High Performance Computing gurus make.

As a case study, we have implemented DataCutter-Lite on the Cell Broadband Engine. The Cell Broadband Engine is an excellent example of modern multicore processors, with its heterogeneous nature, its high performance communication bus, and high throughput processing capabilities. Other examples of modern multicore processors are the current top-end products from AMD and Intel with deep cache hierarchies and the forthcoming Larrabee processor from Intel. Larrabee will have 16 - 32 vector processors and will initially be marketed strictly as a graphics processor. These types of microprocessors are not considered in this paper. The Cell Broadband Engine is an excellent microcosm of the future of multiprocessors, and as such is an excellent test case for our techniques. As future work, we will extend these results to more traditional architectures.

Due to the large potential benefit of modern multicore processors, many research projects have developed techniques to ease the burden associated with writing applications for them. BlockLib [3], IBM's Accelerated Library Framework (ALF) [12], Charm++ [14], CellSs [6], and Sequoia [10] are examples of programming frameworks for the Cell Broadband Engine which use various block-based methods for handling the memory hierarchy of the Cell in order to get good application performance. The Cell Messaging Layer [17] and MPI microtasks [15] are examples of frameworks designed to provide an MPI-like set of semantics for SPE communication. We offer an alternative here, the CBE Intercore Messaging Library, which provides the underlying basis for our streaming programming framework. GLIMPSES [19] provides a method to improve the efficiency of SPE programs by providing profiling information for code optimization. Some streaming frameworks even exist for the Cell Broadband Engine [20], although they offer an interface more appropriate for use with a stream-language compiler and not for application development. Hence, while other programming frameworks exist for the Cell Broadband Engine, DataCutter-Lite is the only active research project which aims to develop a comprehensive, hierarchical middleware system for the development of large-scale, efficient filter-stream programs.

The rest of this paper is organized as follows. Section 2 discusses the overall DataCutter-Lite architecture, while Section 3 discusses the Cell Broadband Engine Intercore Messaging Library, which enables DataCutter-Lite to abstract some of the specific architecture's details. Sec-

tion 4 presents some of the optimizations involved with the design of a filter-stream runtime-engine for the CBE, and Section 5 presents the results of some of the test-case applications. Section 6 concludes the paper and presents our future research directions.

2 Filter-Stream Programming for Heterogeneous, Hierarchical Clusters

2.1 DataCutter and the Filter-Stream Programming Model

DataCutter [7, 8] is a component-based middleware framework [1, 2, 4, 9, 13, 16, 18] designed to support coarse-grain dataflow [5] execution on heterogeneous computational resources.

In DataCutter, the application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through *logical streams*. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). Data flows along these *streams* in *buffers* 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 *placement* is one instance of a *layout* with actual filter copy to physical processor mappings.

The DataCutter runtime system supports data- and task-parallelism. Processing, network, and data copying overheads are minimized by the ability to place filters on different platforms. The runtime engine performs all steps necessary to instantiate filters on the desired machines and cores, to connect all logical endpoints, and to call the filter's interface functions for processing work.

2.2 DataCutter-Lite Architecture and Programming Model

While DataCutter is capable of handling all levels of granularity for an application, from multiple, distributed-address space clusters to SMP nodes, the introduction of heterogeneous and massively parallel microprocessor architectures necessitated a new runtime engine. DataCutter-Lite (DCL) operates only within a single node, but is optimized for modern multicore processors. By using DCL, application developers will able to efficiently make use of modern multicore processors without being expert computer scientists fluent in the newest cutting-edge parallel programming techniques. Additionally, since DCL is a component-based framework, developers can realize a higher degree of productivity over programming the application directly for the architecture in question.

Figure 1. DataCutter-Lite Library Application Interface

```
// Library Initialization Functions
void setup_app(Placement *)
void init_dcl()
// Communication Functions
DCLBuffer * create_buffer(stream,
                          size)
int stream_write(stream,
                 DCLBuffer *)
int stream_close(stream)
```

DCL's architecture has two separate pieces, the runtime engine and the application programming interface (API). The API is comprised of a small number of easy-to-use functions to support application set up and execution. Figure 1 gives an overview of the functions in the main API. While the concepts of layout and placement are distinct, since our work is still in the development stage, we have chosen to combine them into a single initialization function.

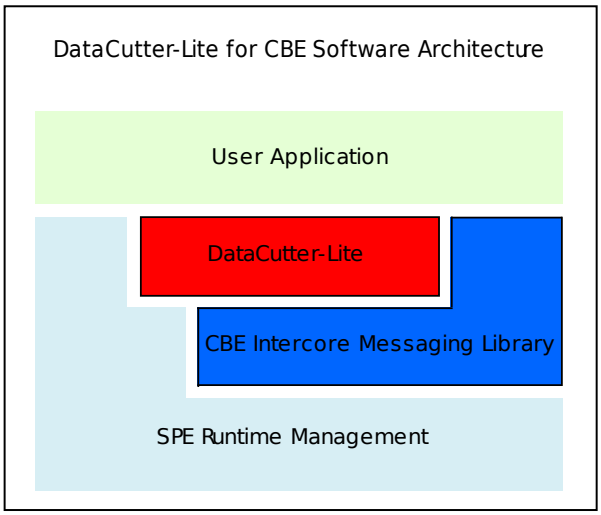

Figure 2. Overall DataCutter-Lite System on the CBE

The first implementation of DCL is designed for the Cell Broadband Engine (CBE). The overall architecture is comprised of three software layers. Figure 2 shows the runtime system and its connections to the various system components. The stepped connections to the various system components are meant to express the notion that while applications can be designed solely to use the DCL API, there

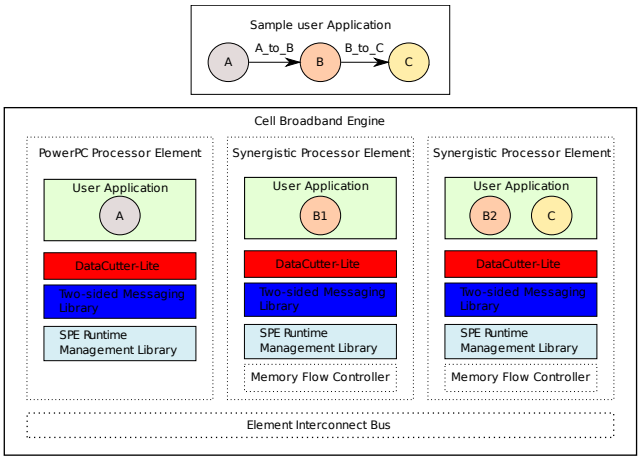

Figure 3. Sample DataCutter-Lite application and example mapping onto the Cell.

is nothing stopping expert developers from mixing function calls from all levels of the software hierarchy if they so choose. The two portions of the system architecture figure surrounded in bold lines, the DCL implementation and the CBE Intercore Messaging Library represent our contributions.

The lowest layer of software is IBM's SPE Runtime Management Library Version 2.2 (libspe2), which is part of IBM's Software Development Kit (SDK) for Multicore Acceleration. The libspe2 library is the interface the developer uses to gain access to the Synergistic Processing Elements (SPE), which represent the majority of the computing power of the CBE.

The next layer of software is the CBE Intercore Messaging Library (CIML). This library is a two-sided communications library whose interface and communication model mimic that of MPI, including blocking and non-blocking send and receive primitives. CIML is detailed in Section 3.

The last layer of software is the DCL runtime system, which is initialized by a function call from the user's main PowerPC Processor Element (PPE) program, as in Figure 1. The initialization function creates threads to manage the SPEs, begins the execution of the SPE programs, and initializes the CIML, to allow two-sided communication between the system processors. At this point, each of the PPE and SPE processors enters a runtime daemon thread which listens for data buffers. Upon receiving a buffer, the daemon consults the filter placement and calls the correct filter's processing function. This event-driven model is well-suited to the CBE, since the SPEs are single-threaded processors and have no capability to switch thread contexts like more traditional processors. All of the underlying details involved with determining where to send the data, the actual transfer

Figure 4. DataCutter-Lite Example PPE Code

```
// PPE main()
// Set up Matrices A, B, pointers
// a_ptr, b_ptr, constants
int
main(int argc, char ** argv) {
 init_dcl();

 for (i = 0; i < NUM_ROWS; i++) {
  DCLBuffer * buffer =
   create_buffer("raw_data",
   BUF_SIZE);

  append_array(buffer, a_ptr,
   NUM_COLS * sizeof(float));
  append_array(buffer, b_ptr,
   NUM_COLS * sizeof(float));

  stream_write(buffer);
  // increment pointers a_ptr, b_ptr
 }
 finish_dcl();
 return 0;
}
```

of the data and the function call is all handled by the runtime system. Also, the developer need not concern themselves with the cleanup of data buffers, since the runtime engine keeps track of all of the buffers that are created, sent, and received. When a filter returns from the processing function, the buffer passed as input is freed. Similarly, when a buffer is written to a stream it is assumed that it is handed off to runtime system. All of the buffers are created in a configurable heap area, so as to alleviate the burden of explicit memory management.

Figure 3 shows a small example application composed of three pipelined filters, A, B, and C and one possible filter placement. To help explain some of the details of the runtime engine, we have decided to create a placement with one copy each of the A and C filters and two copies of the B filter. Buffers written to the 'A_to_B' stream are handed off to the DCL daemon on the PPE. The daemon then determines where to send the buffer, either copy B1 or B2 of the B filter type. This determination is made by a configurable stream sink protocol. Common protocols are round-robin, random, or broadcast. Future versions of DCL will include a demand-driven protocol where filters which sink data faster from a stream will receive more buffers. Buffers written to the 'B_to_C' stream are likewise handed off to the DCL daemon, but since in the placement only one copy

Figure 5. DataCutter-Lite Example PPE Code

```
// PPE setup and filter code
// Called by init_dcl()
void
setup_application(Placement * p) {
 Filter * console =
  get_console(p);
 Filter * fadded =
  place_ppu_filter(p, "added_data");
 Filter * fadder =
  place_filter(p, 0, "add_values");

 Stream * sraw = add_stream(p,
  "raw_data");
 add_source(p, sraw, console);
 add_sink(p, sraw, fadder);

 Stream * sadded = add_stream(p,
  "added_matrix");
 add_source(p, sadded, fadder);
 add_sink(p, sadded, fadded);
}

// When receving a buffer from SPE
void
added_data(DCLBuffer * buffer) {
  // Deal with added matrix data
}
```

Figure 6. DataCutter-Lite Example SPE Code

```
// SPE code: Set up constants
void
add_values(DCLBuffer * buffer) {
 DCLBuffer * out_buffer =
  create_buffer("added_matrix",
  BUF_SIZE);

 float * a = (float *)
  get_extract_pointer(buffer);
 float * b = (float *)
  get_extract_pointer(buffer);
 float * c = (float *)
  get_data_pointer(out_buffer);

 for (i = 0; i < NUM_COLS; i++)
  c[i] = a[i] + b[i];

 stream_write(out_buffer);
}
```

of the C Filter type exists, there is only one destination for these buffers. While buffers written by B1 have to travel through DCL, CIML and libspe2 libraries and through the Memory Flow Controller (MFC) and Element Interconnect Bus (EIB) CBE processor elements, buffers written by B2 will be directly handed off to filter C by DCL by simply passing the pointer of it.

Figures 4, 5 and 6 show some example code for an application which uses one SPE to add together two matrices. The `setup_application()` function in Figure 5 is defined by the developer to tell the runtime engine where to place filters and how to connect the data streams. After the `init_dcl()` initialization function in Figure 4 returns, the `main()` function can do work to begin the computation. Note how simple the SPE code is (see Figure 6); it has none of the complicated operations normally associated with programming the CBE. However, the code is fully multi-buffered, as CIML (see Section 3) allows for the overlap of computation with communication.

2.3 DataCutter for Distributed Multicore Programming

DataCutter and DCL are parts of a burgeoning component-based middleware framework designed to provide efficient dataflow execution on modern hierarchical, heterogeneous cluster supercomputers. Our ultimate goal is to develop a comprehensive, flexible API such that DataCutter will handle coarse-grain dataflow over the LAN/WAN network and DCL will handle fine-grain dataflow within a single node. While we draw a distinction between these two projects since DCL is presented in this paper, the end goal is for that distinction between DataCutter implementations to disappear. DCL instances within a node will act like filters to DataCutter to achieve seamless coarse-to-fine grain integration and interoperability.

Figure 7 shows what a mixed DataCutter and DCL application might look like. At the largest application granularity, whole sets of data are considered. Raw datasets from large-scale simulations, whole patient files, or whole experiments to be run might be examples of this coarse-grained data. This data will be partitioned using DataCutter to run on whole clusters. At the smallest granularity, DCL can be used to leverage multicore processors to analyze the fine-grained data, such as individual simulation timesteps, single data point analysis, or single pixel operations. In Section 5 we discuss a real-world biomedical image analysis application implementation with a mixed DataCutter and DCL paradigm.

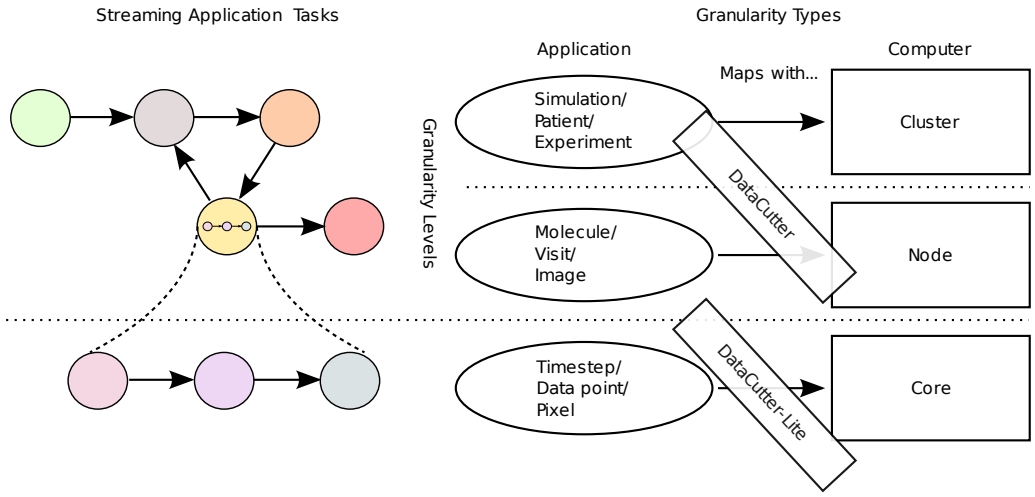

Figure 7. Software and Hardware Granularities

3 CBE Intercore Messaging Library

We have designed and implemented a two-sided communication library for the CBE processor: The CBE Intercore Messaging Library (CIML). CIML makes use of the libspe2 interface, and begins to abstract away some of the architecture-specific details of the communication channel. That is, a developer using function calls in CIML will not need any knowledge of the MFC, which is the functional unit associated with a Synergistic Processor Unit (SPU) enabling it to access main memory through Direct Memory Access (DMA) commands. Each SPU has an associated MFC, and each MFC can queue 16 DMA commands before blocking the SPU's execution of instructions. It is through this method the SPEs can overlap communication with useful computation.

Table 2. CBE Intercore Messaging Library Application Programming Interface

Send Functions	Receive Functions
sendB(dest, src_ptr, size)	recB(src, dest_ptr)
sendNB(dest, src_ptr, size)	recNB(src, dest_ptr)
send_completeNB(dest, src_ptr)	rec_completeNB(src, dest_ptr)
send_complete_allNB(dest)	rec_complete_allNB(src)
	int probeNB(src)

CIML mimics the spirit and usage patterns of MPI with both blocking and non-blocking send and receive function calls. Additionally, CIML allows direct SPE-SPE data transfers, without passing through the main memory cache hierarchy or involving the PPE. IBM's libspe2 allows the developer to map the SPEs' local store space into main memory. DMAs involving addresses in these memory-mapped locations do not go through main memory, and are therefore not subject to the 25 GB/s main-memory bandwidth limit. This being the case, multiple pairwise SPE-SPE communications can occur simultaneously, and these transfers are only subject to SPEs' end-point throughput of 25 GB/s. (Each SPE has simultaneous send and receive bandwidths of 25 GB/s.)

Table 2 shows the API for SPE-SPE communication in CIML. The B and NB suffices on the function names are intended to convey the fact that the send and receive calls deal with blocking and non-blocking communications, respectively. As such, the function calls sendB and recB will block until the communication is complete. The function calls suffixed with NB will return as quickly as possible to allow for more computation to continue while the communication completes. The final PPE-SPE communication API is similar - but not identical - to the SPE-SPE API. The reasoning behind a slightly different API for PPE-SPE communication is discussed in Section 4.

Figure 8 shows the communication bandwidth for CIML, while Figure 9 shows the latencies involved in ping-pong communication. (Incidentally, Figures 8 through 12 show the results of the final, optimized version of CIML; discussions about optimizations and their effects are included in Section 4.) In this experiment we have a single source SPE and we have varied the number of destination SPEs. The results are an average over 100 iterations for each message

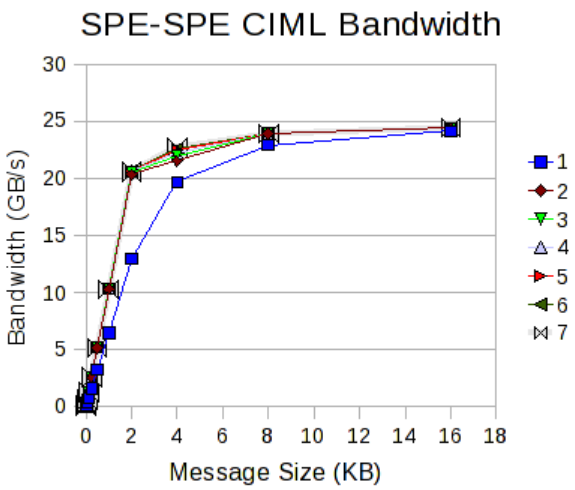

Figure 8. SPE-SPE Communication Bandwidth Results

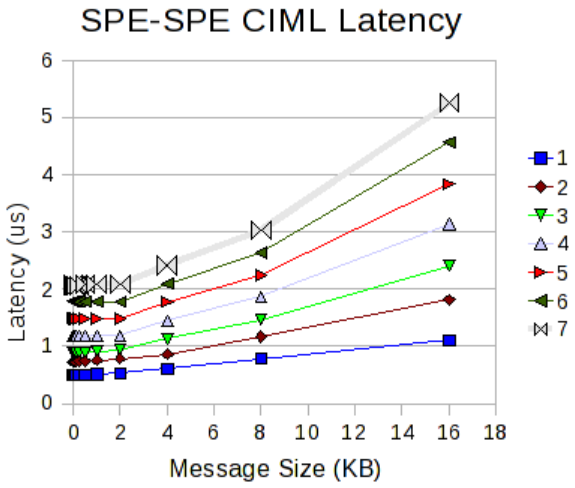

Figure 9. SPE-SPE Communication Latency Results

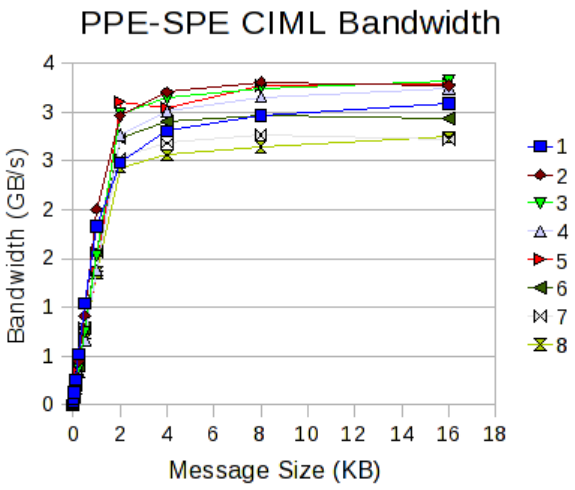

Figure 10. PPE-SPE Communication Bandwidth Results

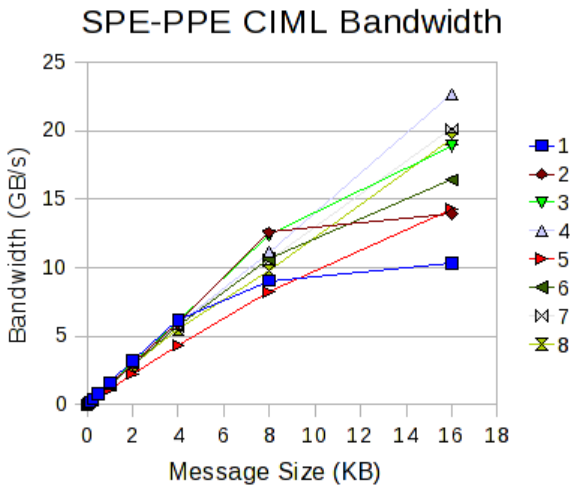

Figure 11. SPE-PPE Communication Performance Results

size from 1 byte to 16 KB; also, the results are aggregate and do not use any hardware-based broadcast mechanism. Since CIML is a two-sided library, there is some extra overhead associated with communications. However, at the larger message sizes, a single SPE-SPE communication channel gets over 80% of the possible bandwidth with 4 KB messages, and over 90% with 8 KB messages. By using more than one SPE, maximum bandwidth can be achieved even with shorter messages.

Figures 10, 11 and 12 show the bandwidth and latency measurements for PPE-SPE communication. Again, these results are an aggregate value, and averaged over 100 iterations. The most striking thing about these charts is the large decrease in communication bandwidth versus the SPE-SPE communication. In order to provide a two-sided communication interface, some information must be transferred from the main memory to the local store of the SPE, and the CBE is constrained in its PPE-SPE communications by a number of architectural idiosyncrasies. These results are actually the result of several rounds of optimizations, some of which are discussed in the next section.

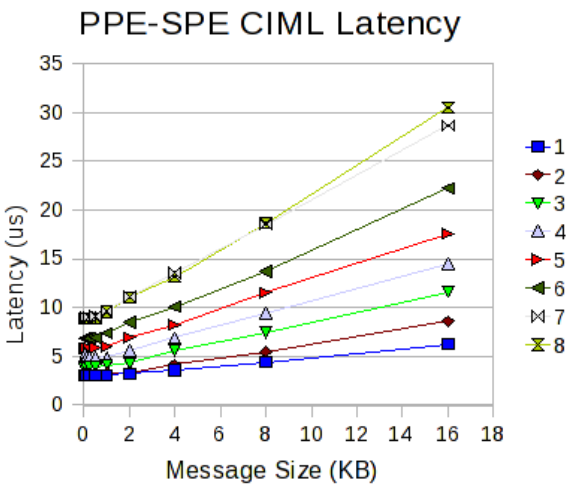

Figure 12. SPE-PPE Communication Latency Results

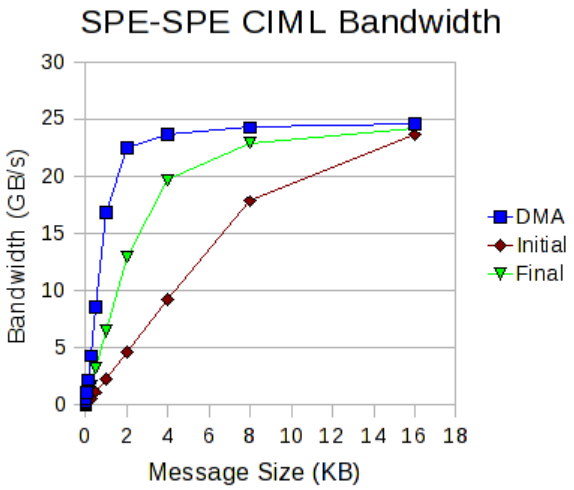

Figure 13. SPE-SPE Communication Bandwidth Results

4 DataCutter-Lite for CBE Optimizations

This section presents some of the optimizations made on the DCL for CBE runtime engine and on CIML. The first two optimizations are applied due to constraints or capabilities of the CBE, while the last two are made for more traditional reasons like allowing communication and computation overlap, or avoiding deadlock.

4.1 High-Bandwidth SPE-SPE Two-Sided Communication

In any two-sided communication, some information must be transferred from the sender to the receiver, and vice versa. Since SPEs in the CBE only have 256 KB of local store memory with which to store the executable and all data, our communication protocol must take this into account. In order to allow high-bandwidth SPE-SPE communication, we have chosen to implement a sender-initiated, pull-based protocol. While the authors of [17] achieve good results with a receiver-initiated protocol, in the streaming paradigm no write is ever blocking, meaning that the cost of having a sender wait for the receiver to transfer destination data is too high.

Therefore, in our protocol, with a 'put,' the sender transfers to the receiver a header packet containing the source address and size of the message. The receiver polls its local header queue, waiting for a message header. When the header is received, the message transfer can be initiated with a 'get', and the data can be used once the transfer is complete. A 'message-received' header is also sent back to the sender. The sender is not involved in the operation, except for sending the header to the receiver. Since DMAs are non-blocking (as long as the DMA command queue is not full), the sender can then proceed to other computation. Figure 13 shows the effect of using a sender-initiated, pull-based protocol versus a sender-initiated, push-based protocol. The results were taken with a single sender and a single receiver.

4.2 Pure Pull-based PPE-SPE Communication

Unfortunately, while a sender-initiated, pull-based scheme works well on the SPEs, when the sender of the message is the PPE, the smaller DMA command queue size available to the PPE harms the communication bandwidth. The MFC in the SPE has a DMA command queue of size 16 for DMA commands initiated by the SPU. The PPE only has a queue of size 8 for DMA commands dealing with that SPE. This asymmetry means that a carefully designed pull-based PPE-SPE communication library is more appropriate than one in which the sender transfers the message header to a known location in the receiver's memory space. That is, when the SPE is attempting to read from the PPE, it must first transfer the message header from the PPE; the PPE's responsibility is simply to write to its own message header area. To increase the bandwidth, we transfer the entire set of message headers. When the PPE is writing multiple messages to the same SPE without expecting a response, the SPE can successfully use this local cache of the message headers to initiate plenty of DMA commands. As such, CIML implements this type of PPE-SPE communication, in favor of mimicking the 'more' two-sided approach used in SPE-SPE communication. Figures 14 and 15 show the

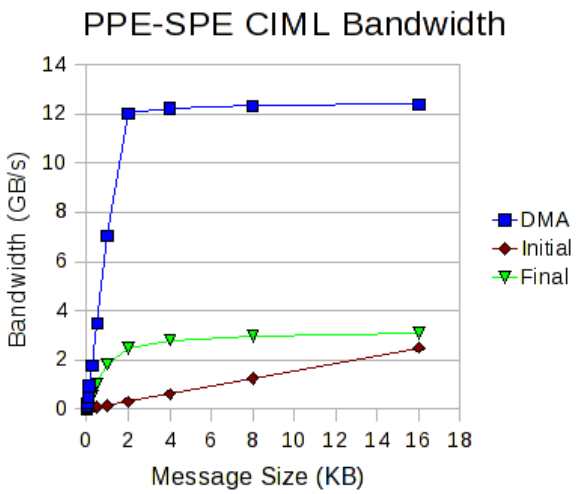

Figure 14. PPE-SPE Communication Bandwidth Results

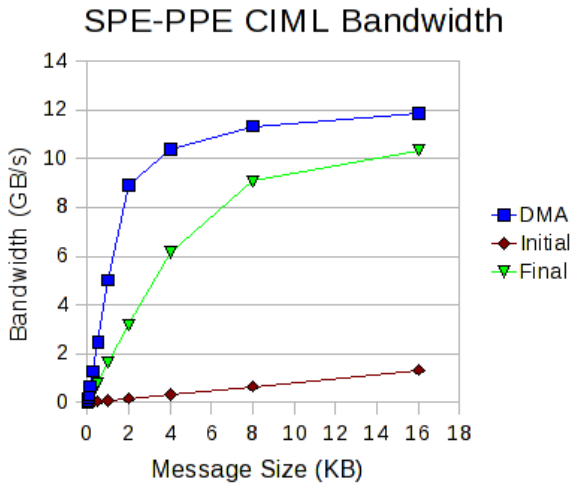

Figure 15. SPE-PPE Communication Bandwidth Results

results of changing the PPE-SPE communication method from that mimicking the SPE-SPE method to a pure pull-based method. As above, the figures show the results of a single sender and a single receiver. Therefore, the final values match the single-threaded results from Section 3. The PPE-SPE transfer from main memory is still hindered by architectural characteristics of the CBE, and as such suffers from a more anemic transfer rate than SPE-PPE data transfers to main memory. Also, even DMAs reach only half of the maximum main memory bandwidth of 25 GB/s during the SPE-PPE transfer.

4.3 Buffer Prefetching

A standard high-performance computing technique is to overlap communication with computation. This is eminently possible with the CBE, since instructions to place DMA commands into the SPE DMA command queue are issued quickly, and are non-blocking when the DMA command queue is not full. Therefore, DCL uses prefetches buffers when calling filters' processing functions. In the simplest case, this allows automatic double-buffering of data for use in streaming operations.

4.4 Fine-grained Buffer Arrival Blocking

When DMA commands are issued for message transfers in the CBE, each command can be assigned a 5-bit tag id. While each SPE's filters are intended to be independent of one another, the buffer heap and the DMA command queue tags are shared among all of the filters running on one SPE. By keeping track of which buffer matches which DMA tag, CIML is able to provide fine-grained buffer receipt or send completion blocking for each filter. The practical effect is that multiple filters running on the same SPE do not step on each other's toes when sending and receiving complex patterns of messages. When a filter needs to create a buffer which is too large for the total remaining heap, the runtime engine can wait for the oldest remaining message transfer to complete. Once that message transfer is complete, the heap space can be freed, giving room to the new buffer. If CIML did not keep track of these DMA tag/buffer associations, the entire DMA command queue would have to be flushed in order to create enough heap space for new buffers. This would cause unwanted latency and bandwidth degradation.

5 Application Experiments

In order to evaluate the performance of the CBE DataCutter-Lite implementation and to stress the programming paradigm, we have developed three applications with a range of characteristics. The experiments were performed on the Ohio Supercomputer Center's Glenn e1350 Blade Center.

The first application we developed to stress the DCL implementation and the programming API is a simple matrix addition application. Since the computation involved with this application is extremely small, this code shows a large Communication-to-Computation-Ratio (CCR). We have used IBM's Accelerated Library Framework (ALF) matrix addition example [12], in order to obtain a good baseline comparison for our DCL-based implementation.

The second of our example applications is a simple color-space transformation to be performed on an image. The application simply transforms each pixel in an image

from the RGB color space to the LAB color space. These types of computations, while simple, are fairly time consuming. As such, this example application will feature a small CCR value. As a baseline comparison, we have used a custom-implemented color-space transformation application which uses only IBM's SDK for the CBE.

The last of these applications is a real biomedical image analysis application [11]. The input to the overall application is a tissue slide image digitized at high resolution. Each RGB pixel is converted to the LAB color space and some statistics are calculated on a per-tile basis. (We reused the color-space transformation code presented earlier.) The luminance channel is then taken from the LAB image and a local binary pattern (LBP) feature is calculated. The four statistics per image channel and the LBP feature comprise a feature vector which is used in a classification stage in order to determine the properties of the image tile. To compare with our DCL-based implementation, we have developed a custom CBE implementation of the image analysis application. This implementation uses one main loop which reads the RGB image tiles from a socket and performs the functions on each tile. The operations are slightly decoupled, meaning that the RGB-to-LAB color space transformation and the LBP feature calculation must be scheduled separately on the SPEs. The main loop in the application acts as this task scheduler.

The first two applications, the matrix addition and the color-space transformation, are simple kernels that represent the widest range of CCR values which real applications might exhibit. Most real applications are built from components like these and hence their performance can be easily predicted by looking at how the individual components behave.

The DCL versions of all three applications and their baseline counterparts are inherently composed of the same independent tasks. As such, there is no algorithmic method used to reduce the amount of work for one of the implementations versus the other. All of the execution times shown in the charts for the baseline versions of the applications are the best times obtained by hand-tuning the application performance. Similarly, optimizations were made to the DCL runtime engine in order to allow the DCL versions of the three applications to achieve the best performance results. Further, optimizations were made to the DCL versions of the applications themselves. On the PPE, the optimizations were mainly made to relieve the main memory bandwidth requirements. For the SPE code, the most important optimization to be made is choosing the size of the data buffers; incorrect choices limit the amount of communication and computation overlap which is achievable. These optimizations were made in an ad-hoc manner, and their in-depth discussion is beyond the scope of this paper. In our future work, we intend to develop automatic techniques such that

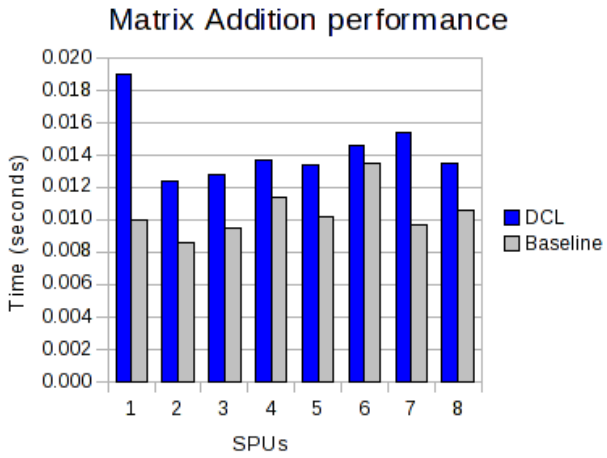

Figure 16. Execution times for Matrix Addition

these performance optimizations are made without the developer's intervention.

Figure 16 shows the execution times for the two matrix addition implementations. The input matrices are 1024 x 512 in size. DCL's execution times are greater than those of IBM's ALF implementation, ranging from 8% higher on 6 SPEs to 91% higher execution time on one SPE. The higher execution times are due to a couple of reasons. First, the construction of serialized data buffers is an operation which the ALF implementation does not need. Further, since the DCL matrix addition program is very simple, the highest throughput DMA buffer size of 16 KB is not used (DCL application simply transfers one row at a time), whereas the ALF implementation uses this DMA buffer size. Both of these issues can be solved with some extra effort, but the simple implementation is meant to serve as a baseline number, and is the worst the DCL method is likely to give. Further, the ALF implementation uses many cryptic function calls to set up the task graph. The DCL implementation merely requires the use of a handful of functions.

Figure 17 shows the execution times and parallel speedups for the color-space transformation performed on 32 image tiles. Since the overheads of reading the images from the disk are disregarded here, the speed up is nearly linear, reaching a value of 7.9 for the baseline version and 7.7 for the DCL version. Without intimate knowledge of how the IBM libspe2 library schedules the logical SPE contexts onto physical SPE resources, it is hard to postulate a reason why a knee exists in the speedup curves after 4 SPEs. However, we expect that up to 4 SPEs, a degree of regularity is maintained in the placement on the physical resources - and of the communication pattern, there being 4 independent message channels in the ring bus. From 4-7 SPEs, this

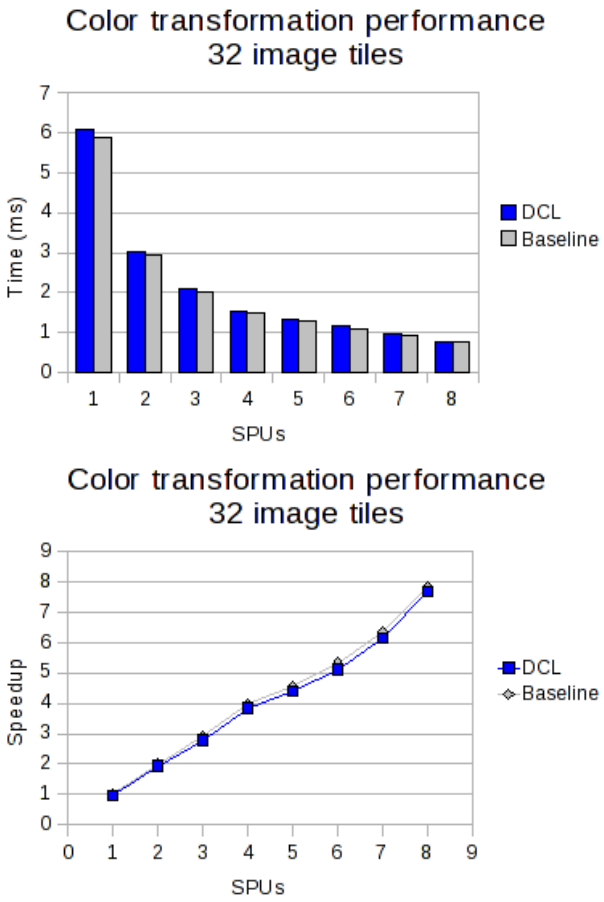

Figure 17. Execution times and speedups for color transformation for 32 image tiles

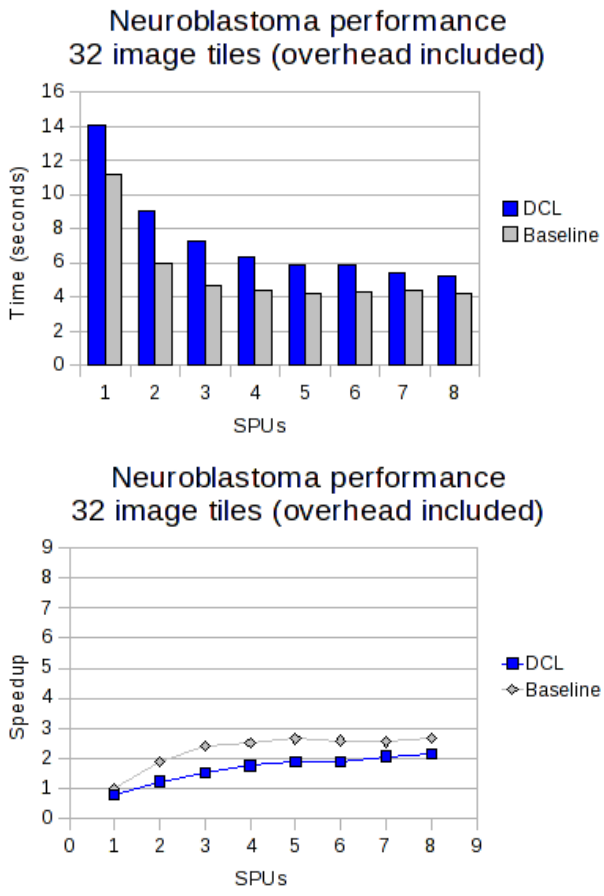

Figure 18. Execution times and speedups for biomedical image analysis application for 32 image tiles - overheads included

regularity is necessarily disturbed. When 8 SPEs are used, this regularity returns to a reasonable degree, even though the communication bus is most highly loaded at this configuration.

Figures 18, 19 and 21 show the results of the experiments on the full biomedical image analysis application with various overheads included and excluded, respectively. When end-to-end applications are considered, even the most efficient algorithm implementation is subject to such concerns as disk latency, and upstream data overheads. In this case, the upstream data overhead is the decompression of the TIFF images to be calculated. When excluding these overheads, we see a similar pattern of near-linear speedup for the DCL implementation. Unfortunately, the baseline version of the application actually begins to suffer when the number of SPEs used rises above 5. This is likely due to the extra scheduling overhead and memory bandwidth used in the baseline's implementation, since it decouples the two major stages of the operation and schedules them separately. The DCL implementation simply writes one buffer as output from the first stage to the input of the second stage. Since this buffer stays in the SPE, it saves main memory bandwidth, and since the second stage is triggered by the runtime system resident on the SPE, the PPE is not involved in the scheduling operation, saving time.

When the TIFF decompression overheads are considered, the application performance decreases, particularly when more SPEs are involved in the computation. The best speedups achieved are 2.2 for the DCL version and 2.7 for the baseline version. The DCL version does not read the decompressed TIFFs from an incoming socket, since this operation would require the use of mutexes in order to share the socket, and we have avoided this type of programming, since it is incompatible with our goal of designing a runtime system devoid of these types of details. As such, each TIFF is decompressed in the same thread which calls the DCL routines to analyze the image.

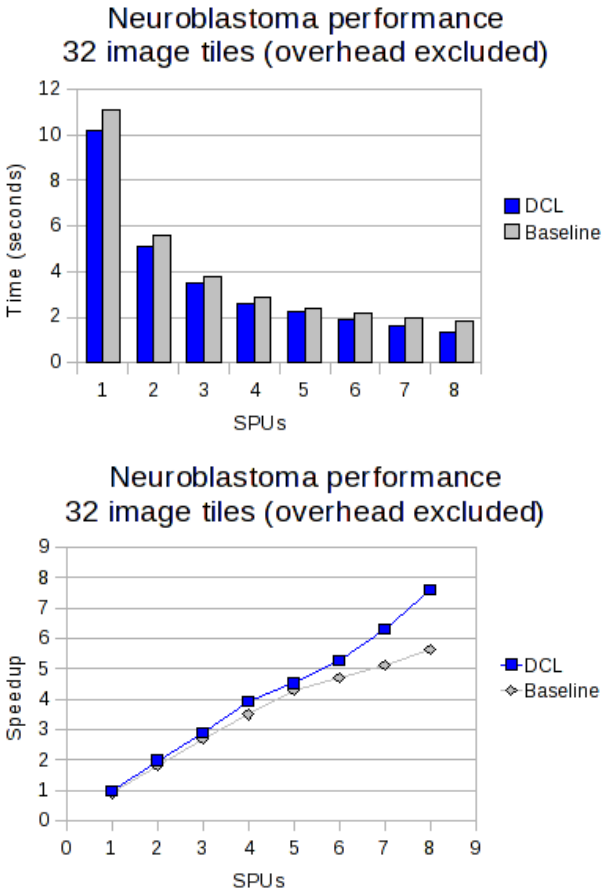

Figure 19. Execution times and speedups for biomedical image analysis application for 32 image tiles - overheads excluded

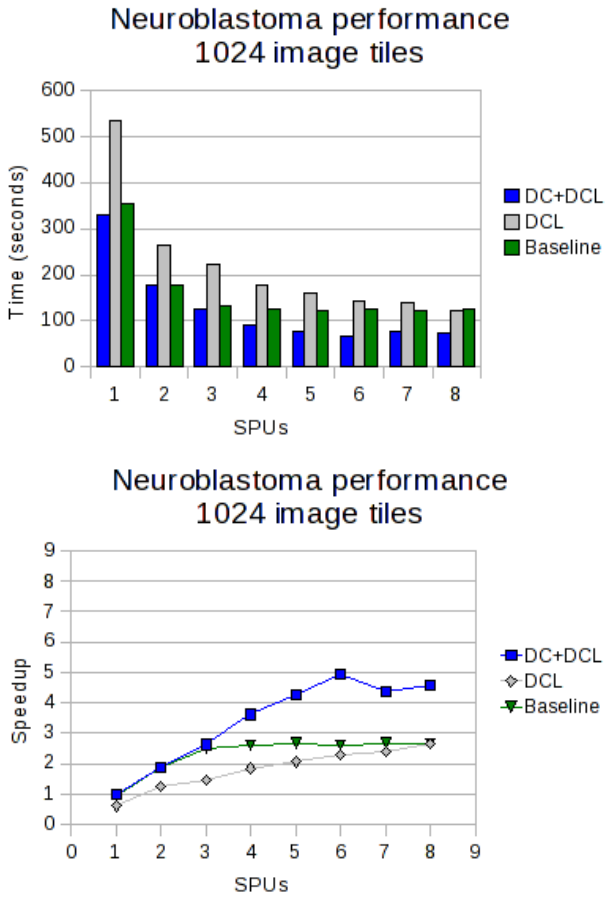

Figure 21. Execution times and speedups for biomedical image analysis application for 1024 image tiles

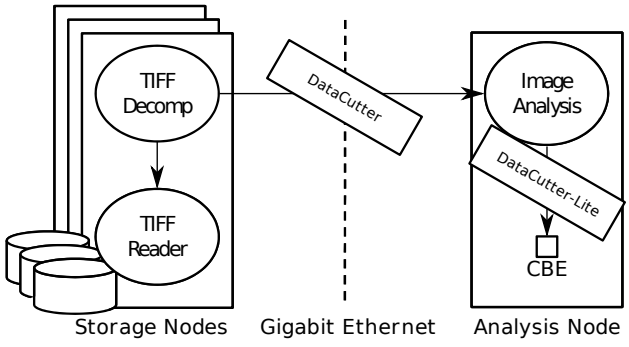

Figure 20. DataCutter and DataCutter-Lite mixed implementation

To solve the problem of insufficient TIFF decompression bandwidth, we have implemented a mixed DataCutter+DCL version of the image analysis application. Figure 20 shows the layout of the integration of DataCutter for internode communications and DCL for intranode executions. As such, Figure 21 shows the results when DataCutter is used to distribute the TIFF tile decompression stage among several computational nodes, and one CBE processor is used to analyze the tiles with its 8 SPEs. Unfortunately, OSC's CBE blades are currently only configured to run with Gigabit Ethernet, which limits the amount of help distributed nodes can give.

6 Conclusions

This work presented DataCutter-Lite (DCL), a fine-grained, component-based, filter-stream programming library and runtime engine. DCL is meant to allow application developers access to the new high-performance, multicore microprocessors available in the marketplace. We showed that the runtime engine is able to support high-bandwidth communications among the processor's cores without burdening the developer with low-level, architecture-specific instruction syntax. We showed that for applications with high communication to computation ratios, DCL does not incur an additional overhead. For applications with low CCR values, we showed that DCL scales as well as custom application implementations.

Clearly there is promise to the idea of filter-stream programming on modern multicore processors. This work represents a solid first step towards the future goals of programming libraries meant to provide robust APIs for programming large supercomputers comprised of duplicated nodes with multiple multicore processors. Future work in this area will be in multiple directions. First, the CBE-specific implementation presented in this paper will be used as a test-bed for further optimizations. We plan to release the software as open-source along with the software to help link DCL to the legacy LAN/WAN-specific DataCutter middleware.

The next set of goals will be implementing DCL for more traditional multicore microprocessors. This will involve work into minimizing the overheads involved at every level of the software stack. However, by paying careful attention to these overheads - as well as processor traits such as cache hierarchy behavior and size - we intend to create a runtime engine and programming library rivaling other options.

In parallel with both the CBE-specific DCL work and the more traditional multicore microprocessor version, work will continue to integrate DCL into DataCutter proper. This will allow a seamless application development experience, from coarse-grained dataflow at the grid or cluster level to fine-grained dataflow at the node level. Further end-to-end application optimizations would then be possible, since the layout of the entire application's filters will be known a priori. The filter stages can be appropriately sized and optimally placed for maximum application bandwidth at all granularities of dataflow.

The most long-term goals involve the development of algorithms to automate the creation of transparent filters for increased bandwidth, along with the placement of the instantiated filters onto physical resources. These decisions, along with the scheduling of the transmission of data buffers and the tasks to compute can be automated, alleviating the stress placed on the developer to optimize these application characteristics through trial and error.

References

[1] The ABACUS project. *http://www.cs.cmu.edu/∼amiri/abacus.html*.

[2] M. Aeschlimann, P. Dinda, J. Lopez, B. Lowekamp, L. Kallivokas, and D. O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, Las Vegas, NV, June 1999.

[3] M. Älind, M. V. Eriksson, and C. W. Kessler. BlockLib: a skeleton library for cell broadband engine. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM.

[4] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[5] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster i/o with river: making the fast case common. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 10–22, New York, NY, USA, 1999. ACM Press.

[6] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *SC*, page 86, 2006.

[7] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133. National Aeronautics and Space Administration, March 2000. NASA/CP 2000-209888.

[8] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[9] Common Component Architecture Forum. *http://www.cca-forum.org*.

[10] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[11] T. D. R. Hartley, Ü. V. Çatalyürek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. Biomedical image analysis on a cooperative cluster of GPUs and multicores. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008*, pages 15–25, 2008.

[12] IBM. Accelerated Library Framework. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/41838EDB5A15CCCD002573530063D465.

[13] C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, Cancun, Mexico, May 2000.

[14] D. M. Kunzman, G. Zheng, E. J. Bohm, J. C. Phillips, and L. V. Kalé. Poster reception - Charm++ simplifies coding for the cell processor. In *SC*, 2006.

[15] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Syst. J.*, 45(1):85–102, 2006.

[16] R. Oldfield and D. Kotz. Armada: A parallel file system for computational grids. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[17] S. Pakin. Receiver-initiated message passing over RDMA networks. In *IPDPS*, pages 1–12, 2008.

[18] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

[19] J. Sreeram and S. Pande. GLIMPSES: A profiling tool for rapid spe code prototyping. In *Workshop on New Horizons in Compilers (Held in Conjunction with IEEE Intl. Conference on High Performance Computing (HiPC 2007))*, 2007.

[20] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36(2):18–27, 2008.

Biographies

Timothy D. R. Hartley is a Ph.D. student in the Department of Electrical and Computer Engineering at The Ohio State University. His research interests involve high-performance scientific computing and emerging architectures. He received his M.S. from The Ohio State University in 2006 and his B.S. from New Mexico State University in 2002. He is currently a Fellow of the Dayton Area Graduate Studies Institute's AFRL/DAGSI Ohio Student-Faculty Research Fellowship Program.

Umit V. Catalyurek is an Associate Professor in the Department of Biomedical Informatics at The Ohio State University, and has a joint faculty appointment in the Department of Electrical and Computer Engineering. His research interests include combinatorial scientific computing, runtime systems for data-intensive computing, and high-performance computing in biomedicine. He received his Ph.D., M.S. and B.S. in Computer Engineering and Information Science from Bilkent University, Turkey, in 2000, 1994 and 1992, respectively.