Intra- and Inter-chip Communication Support for Asymmetric Multicore Processors with Explicitly Managed Memory Hierarchies

Benjamin A. Rose

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Science

May 12, 2009
Blacksburg, Virginia

Keywords: Cell BE, Multi-buffering schemes, Workload distribution

Intra- and Inter-chip Communication Support for Asymmetric Multicore Processors with Explicitly Managed Memory Hierarchies

Benjamin A. Rose

(ABSTRACT)

The use of asymmetric multi-core processors with on-chip computational accelerators is becoming common in a variety of environments ranging from scientific computing to enterprise applications. The focus of current research has been on making efficient use of individual systems, and porting applications to asymmetric processors. The use of these asymmetric processors, like the Cell processor, in a cluster setting is the inspiration for the Cell Connector framework presented in this thesis. Cell Connector adopts a streaming approach for providing data to compute nodes with high computing potential but limited memory resources. Instead of dividing very large data sets once among computation resources, Cell Connector slices, distributes, and collects work units off of a master data held by a single large memory machine. Using this methodology, Cell Connector is able to maximize the use of limited resources and produces results that are up to 63.3% better compared to standard non-streaming approaches.

This research is supported by NSF (grants CCF-0346867, CCF-0715051, CNS-0521381, CNS-0720673, CNS-0709025, CNS-0720750), DOE (grants DE-FG02-06ER25751, DE-FG02-05ER25689) and IBM through an IBM Faculty Award (VTF-874197).

Contents

List of Figures

Chapter 1

Introduction

Asymmetric parallel architectures are rapidly being established in emerging systems as the *sine qua non* for achieving high performance without compromising reliability. The model has been realized in asymmetric multi-core processors, where a fixed transistor budget is heavily invested on many simple, tightly coupled, accelerator-type cores. These cores provide custom features that enable acceleration of computational kernels operating on vector data. Accelerator cores are controlled by relatively few conventional processor cores, which also run system services and manage off-chip communication. Researchers have collected mounting evidence on the superiority of asymmetric multi-core processors in terms of performance, scalability, and power-efficiency [6, 27, 29]. The recent advent of the Cell Broadband Engine (Cell) processor and GPUs as High-Performance Computing (HPC) and data processing engines [5, 9, 10, 15, 18, 28, 7, 30], further attests to the potential of asymmetric architectures.

The accelerator-based approach is not only limited to the processor level, it also lends itself to be extended for designing distributed asymmetric clusters, such as LANL's RoadRunner [7]. In the distributed setting, the accelerators are packaged compute nodes with customized components, such as GPGPU, FPGAs, Clearspeed CSX600 [11] coprocessors, IBM Cell/BE processors, and NVIDIA G8800 GPUs [20], connected over a high-speed network to a powerful front-end component that manages the whole cluster. The rapid growth in high-speed networks and the use of low cost commodity off-the-shelf hardware, makes such distributed asymmetric clusters natural substitutes for expensive high-end supercomputers.

In this thesis I focus on Cell Connector, a framework designed to support a cluster of accelerator based nodes connected to a general purpose "host" node. Cell Connector will support the use of very large data sets while using accelerator nodes that have very little memory through the use of a single (or few) host nodes which have much larger memories.

I will explain the design, development, and evaluation of Cell Connector using various codes already ported to this framework. Some of these codes are hand optimized for the accelerator nodes and others use a MapReduce implementation on the individual nodes themselves. This will demonstrate the behavior of both communication and computational bound applications using Cell Connector.

Also found in this thesis is CellStream, a framework supporting communication intensive applications on a single Cell Broadband Engine chip. CellStream's greatest features support near peak communication speeds between accelerators and main memory, and it ensure data remains coherent and usable. These features help support a drop in computational kernel for data communication bound applications. The design and evaluation of CellStream can be found in Chapter 6.

Chapter 2

Related Work

I discuss some works that are related to the material found in this thesis.

2.1 DaCS

The Data Communication and Synchronization (DaCS) framework provided by IBM is their communication framework for communicating between the host Opteron blades and the PowerXCell8i blades in the latest RoadRunner supercomputer[12, 7]. It provides process management and data communication APIs for managing Cell-based systems from a host system that can be of a x86 or PowerPC architecture. The underlying hardware connecting these systems can be a traditional network, or it can be using a special PCI-Express bridge, as in the case of the Roadrunner. Provided are communication mechanisms similar to message passing communication models (asynchronous sends and receives) along with remote DMA (rDMA) transfers.

The DaCS framework is actually complementary to the work presented in this thesis. While Cell Connector was implemented using MPI for all communication, a layer of abstraction was built in to ensure it no way depends on MPI for communication. Using the message passing communication model (and perhaps with a bit more work the rDMA model) Cell Connector should be able to take advantage of any extra benefits provided by DaCS with relative ease.

2.2 MPI

The MPI Specification[25] has been one of the most popular parallel programming tools since its introduction in 1994. MPI was designed to provide a tool for effective distributed memory multi-processor hardware, however it is also useful when programming a shared memory multi-processor machine in a distributed memory fashion. While it provides a time tested tool for utilizing different parallel processing systems, it can be harder for programmers to use than other parallel processing tools, such as OpenMP[3].

MPI, specifically the LAM-MPI implementation, was used as the communication mechanism during the development of Cell Connector. It was chose as such due to familiarity, stability, and popularity. All experiments ran in this thesis were using LAM-MPI 7.1.2.

2.3 OpenMP

OpenMP[3] is a very popular method when programming shared memory parallel machines. It adds compiler directives which describe how to parallelize certain sections of code. Usually these directives exploit loop level parallelism, where different loop iterations execute on different processing elements. It is a much less daunting introduction into parallel programming than MPI[25] because the programmer doesn't have to worry about communication.

2.4 SDK Bandwidth Benchmark

There currently exists a benchmark for measuring the peak system bandwidth for the Cell BE. This benchmark is bundled with the Cell SDK and has shown to generate results that are consistent with the theoretical on-chip bottlenecks[21]. They measure the bandwidth of not only SPE to Main Memory, but also between SPEs and from the PPE to main memory using either DMA lists or regular sequential DMA calls. Although the results in their paper were generated with a slower clocked Cell Processor than the commonly found 3.2GHz version, the results scale accordingly.

They list some of the methods for achieving the most efficient use of the on chip bus. These methods include using DMA lists for small transfers, loop unrolling, use SPE to SPE communication when possible to avoid the bottleneck at the memory interface, and so on.

The major difference between this benchmark and my CellStream benchmark is data coherency. While this benchmark was designed to get the absolute fastest speed possible inside of the Cell BE, it does not maintain data integrity and instead uses the same buffers for multiple simultaneous read/write operations. CellStream aimed to get close to these speeds while still maintaining coherent work units that computation could be performed on and return them back to main memory in the same state the SPE left them in.

2.5 STREAM

STREAM[24] is a benchmark for conventional shared memory multi-processors. It measures sustained data transfer speeds and performs vector operations on the data transferred. Using STREAM, one can determine the performance of a processor when all the available data does not fit into the local cache and must be transferred from main memory. The growing gap of processor speed versus memory speed is clearly illustrated with this benchmark. This benchmark is the equivalent of the memory benchmark bundled with the Cell SDK for conventional shared memory multi-processors (such as x86, PowerPC, etc.).

2.6 MapReduce

MapReduce[14] is a parallel programming model for very large-scale data processing application. It is based on two operations: a map operation and a reduce operation. It uses the map operation to process key/value pairs from input data and then a reduce operation to combine these processed key/value pairs into one or more results. It is the programming model used by Google for many of its data mining operations on its commodity component clusters[14].

CellMR[31] is an extension of the Cell Connector framework on which a more high-level parallel programming model, such as MapReduce, can be based. While similar to Cell Connector, it used MapReduce on every compute node for every application, supported one to many global reduction operations, and even included an automatic work unit size optimizer.

Chapter 3

Enabling Technologies

In this section, we discuss the technologies that serve as the motivator for Cell Connector and enable its design.

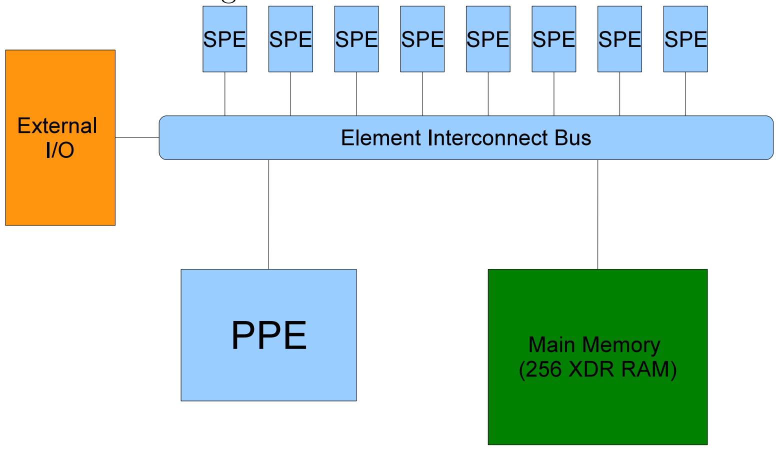
3.1 Using commodity components

The use of cheap off-the-shelf components in large-scale clusters is well established. Setups from academia, e.g., Condor [34], etc., to commercial data centers, e.g., Google [8], Amazon's EC2 [4], etc., routinely employ such components to meet their high-performance computing needs. Commoditization is now becoming true for asymmetric accelerator-type processors such as Cell/BE-based Sony Play Station 3 (PS3) [1, 26] and NVIDIA GPGPU-based graphics engines [17, 16]. Consequently, there is a downward cost trend for such components, which facilitates the building of asymmetric accelerator-based clusters, similar to the framework that we consider in this study.

Accelerator engines provide much higher performance to cost ratio compared to conventional processors, and have also been shown to support better thermal and energy efficiency [17]. Thus, a properly designed asymmetric cluster comprising accelerators has the potential to provide very high performance at a fraction of the cost and operating budget of a traditional symmetric cluster.

Unfortunately, accelerators also pose challenges to programming and resource management. Programming accelerators requires working with multiple ISAs and multiple compilation targets. Accelerators typically have much higher compute density and raw performance than conventional processors, therefore coupling accelerators with conventional processors may introduce imbalance between the two. Accelerators also typically have limited on-board storage and limited – if any – support for system services such as I/O. To ensure overall high efficiency, resource management on accelerator-based systems needs to orchestrate carefully

Figure 3.1: Cell architecture.



data transfers and work distribution between heterogeneous components.

3.2 IBM's Cell Broadband Engine

The Cell processor [33] provides a suitable resource that can serve as an accelerator component in Cell Connector. The availability of the Cell processor in the commodity Sony PS3 setup further makes it economically attractive for deployment on clusters.

The Cell [22, 19], shown in Figure 3.1, is composed of a single PowerPC Processing Element (PPE), which acts as a manager of cores, and eight Synergistic Processing Elements (SPE), which are specialized for high-performance data-parallel computation. A fast Element Interconnect Bus (EIB) connects all the cores with memory and an external I/O channel to access other devices (such as the disk and network controller). The SPEs have private address spaces and the programmer is responsible for moving data between the main memory and each SPE's local storage using the Cell's coherent DMA mechanism. The programmer can overlap data transfer latency with computation, by issuing multiple asynchronous DMA requests on the SPE or PPE side. In current installations, the PPE runs Linux with Cell-specific extensions that provide user-space libraries access to the accelerator-type cores of the processor.

We use Sony PS3's as compute nodes in this work. A shortcoming of using PS3 for high performance computing is that it has only 256 MB of XDR RAM out of which only about 180 MB is available to user applications. In a cluster setting, this shortcoming may be addressed with proper data streaming and staging. The Cell gives the programmer the ability to explicitly manage the flow of data between the main memory and each individual SPE's local store.

Chapter 4

Design

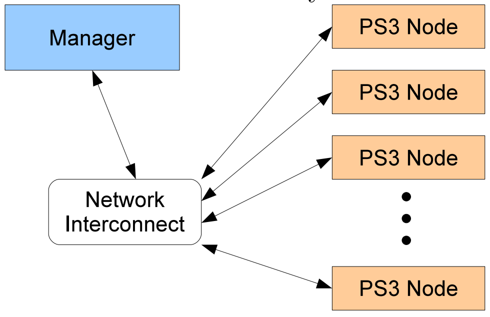
In this chapter I present the detailed design of the Cell Connector framework. The features provided to the application developer and implementation details are also covered in sections 4.4 and 4.5 respectively.

4.1 High-Level System Architecture

The two main components in the high level architecture are the host machine and the accelerator machines. The host machine is a general purpose multi-core server with a large amount of DRAM, which acts as a cluster *manager*. The back-end compute nodes are asymmetric cell-based accelerators, PS3s. The manager distributes and schedules the workload to the compute nodes. Each compute node then uses code optimized for that accelerator node to utilize all of its processing resources. The accelerator nodes then return the computed results back to the manager node where all the results are merged into one global result.

A high-level view of the Cell Connector architecture is illustrated by Figure 4.1. The manager and all the compute nodes are connected via a high-speed network, e.g., Gigabit Ethernet.

Figure 4.1: Cell Connector system architecture.



The manager is primarily responsible for invoking the jobs at the compute nodes, distributing data and allocating work between compute nodes, and providing other support services as the front-end of the cluster. The brunt of the processing load is carried by the PS3 nodes. This setup mimics, in a distributed setting, the architecture adopted in emerging asymmetric multi-core processors and asymmetric hybrid clusters, such as the PS3 and LANL's RoadRunner, respectively. Thus, it is expected to yield a high performance to cost ratio.

The novelty of Cell Connector lies in its adoption of a streaming approach for distributing workloads. Allocating huge workloads to compute nodes in a single operation would choke the limited non-computation resources on the asymmetric compute nodes and negate any performance benefits. Instead, Cell Connector slices the input into small work units and streams them to the compute nodes, which can then be processed efficiently. Cell Connector employs a hoard of techniques such as double-buffering to avoid stalls due to I/O operations and asynchronous data mapping and collection to ensure peak performance of all system components.

4.2 Design Alternatives

A crucial task of Cell Connector is to efficiently allocate chunks of large application data to compute nodes. This poses several alternatives. A straw man approach is to simply divide the total input data into as many chunks as the number of available processing nodes, and copy the chunk to the compute node's local disk. The application on the compute node can then retrieve the data from the local disk as needed, as well as write the results back to the local disk. When the task completes, the result-data can be read from the disk and returned to the manager. This approach is easy to implement, and potentially lightweight for the manager node as it reduces the allocation task to a single data distribution. However, there are several drawbacks to this approach: (i) it requires creation of additional copies of the input data from the manager's storage to the local disk, and vice versa for the result data, which can quickly become a bottleneck, especially if the compute node disks are much slower than those available to the manager; (ii) it entails changing the workload to account for explicit copying, which is undesirable as it burdens the application programmer with system-level details, thus making the application non-portable across different setups; (iii) it entails extra communication between the manager and the compute nodes, which can slow the nodes and affect overall performance. Hence, we do not adopt such an approach in Cell Connector.

Another alternative is to still divide the input data as before, but instead of copying a chunk to the compute node's disk as in the previous case, map the chunk directly into the virtual memory of the compute node. The goal here is to leverage the high-speed disks available to the manager and avoid unnecessary data copying. However, this approach can create chunks that are very large compared to the (small) physical memory available at the compute nodes, thus leading to memory thrashing and reduced performance. Hence, single division of input data is not a viable approach for Cell Connector.

The third alternative that we consider is dividing the input data into small size chunks that can be efficiently processed at the compute nodes without thrashing and without requiring explicit data copying to local disk. Instead of a single division of data, the approach streams chunks to the compute nodes until all the data has been processed. This approach can improve the performance from the compute nodes, at the cost of increasing the manager's load. However, with careful design an efficient balance between the manager's load and compute node performance can be achieved. We adopt this data streaming approach in Cell Connector.

4.3 Cell Connector Operations

In this section, we describe the runtime interactions between the various components of Cell Connector at the manager and each of the compute nodes. These interactions are depicted in Figure 4.2.

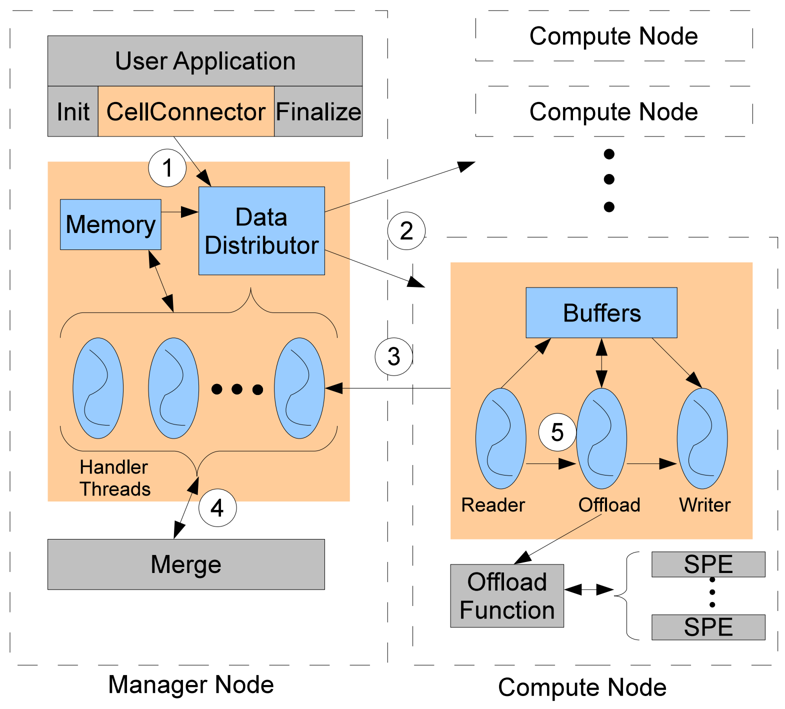
4.3.1 Manager Operation

The manager is responsible for a number of tasks such as job queues, scheduling, data hosting, and managing compute nodes. The Cell Connector library on the manager assumes the user application has taken care of all the initialization that is required of that application when it is invoked (Step 1 in Figure 4.2). This includes declaring custom data types, allocating memory and reading in all the required data, and determining the best work unit size.

Next, the client tasks are started on the available compute nodes (Step 2). These tasks essentially self-schedule their work by requesting input data from the manager, processing it, and returning the results back to the manager in a continuous loop (Steps 2&3). Once the manager receives the results, it merges them (Step 4) to produce the final result set for the application.

This model has a possibility for stalling when data is sent from the manager to the compute nodes for processing. In order to avoid stalling on the manager, it uses asynchronous sends so the manager can quickly move on to handling requests from other compute nodes. When results are being received from compute nodes, a simple and asynchronous method is needed to again make sure the manager doesn't stall on any operation. Since the manager has to always be available for incoming requests, handler threads are used for receiving results from each compute node and then merge them into the global result using the application specific merge function. On the compute node, multiple buffers are utilized so data is always being read in, worked on, and sent back to the manager simultaneously (Step 5).

Figure 4.2: Interactions between Cell Connector components.



4.3.2 Compute Node Operation

Application tasks are invoked on the compute nodes (Step 2), and begin to execute the request, process, and reply (Step 5) loop as stated earlier. We refer to the amount of application data processed in a single iteration on a compute node as a *work unit*. With the exception of an application-specific *Offload function*[1] to perform the computational work on the incoming data, the Cell Connector framework on the compute nodes provides all other functionality, including communication with the manager and preparing data buffers for input and output.

Each compute node has three main threads that operate on multiple buffers for working on and transferring data to/from the manager. The Reader thread is responsible for requesting and receiving new data to work on from the manager. The data is placed in a receiving buffer. When the thread has received the data, it hands off the receiving buffer to an Offload thread, and then requests more data until all free receiving buffers have been utilized. The Offload thread invokes the Offload function on the accelerator cores with a pointer to the receiving buffer, the data type of the work unit (specified by the User Application on the manager node), and size of the work unit. Since the input buffer passed to the Offload

[1]The function that processes each work unit on the accelerator-type cores of the compute node. The result from the Offload function is merged by the generic core to produce the output data that is returned to the manager.

Table 4.1: Cell Connector provided data types

Cell Connector Type	MPI Type	C/C++ Type
CELL_CON_BYTE	MPI_BYTE	N/A
CELL_CON_CHAR	MPI_CHAR	char
CELL_CON_SHORT	MPI_SHORT	short
CELL_CON_INT	MPI_INT	int
CELL_CON_LONG	MPI_LONG	long
CELL_CON_LLONG	MPI_LLONG	long long
CELL_CON_FLOAT	MPI_FLOAT	float
CELL_CON_DOUBLE	MPI_DOUBLE	double
CELL_CON_UNSIGNED	N/A	unsigned

function is also its output buffer, all of these parameters are read-write parameters. This is to allow the Offload function with the abilities to resize the buffer, change the data type, and change the data size depending on the demands of the output. When the Offload function completes, the recent output buffer is handed off to a Writer thread, which returns the results back to the manager and also frees the buffer for reuse by the Reader thread. Note that the compute node supports variable size work units, and can dynamically adjust the size of buffers at runtime.

4.4 Using Cell Connector

From an application programmer's point of view Cell Connector is used as follows. The application is divided into three parts as shown earlier in Figure 4.2.

4.4.1 Data Types

In order to abstract Cell Connector from any specific communication mechanism, it uses its own data abstracts when describing data in case the communication mechanism it is currently using requires it. This data type mechanism is very similar to data types used in MPI. The provided data types are listed in Table 4.1. Note that `CELL_CON_UNSIGNED` is actually used in combination with the other data types in a bit-wise OR fashion. So if one has an array of unsigned integers, the appropriate Cell Connector data type would be `CELL_CON_INT | CELL_CON_UNSIGNED`.

Of course there are always instances where user applications require their own custom data

Figure 4.3: Prototype of the `createType` function.

```
Cell_Con_Type createType(
        int count,
        unsigned int *blocklens,
        unsigned int *indices,
        Cell_Con_Type *oldTypes,
        unsigned int size
);
```

types. Cell Connector supports these custom data types by providing a construction function, `createType`, which behaves similarly to the MPI custom data struct function. The prototype of the `createType` function can be found in figure 4.3.

The `count` parameter specifies how many elements are in this custom data type. The `blocklens` parameter specifies the byte size of each element in the custom data type. Note that it accepts a pointer to an array of integers, which should have a value for each element. `indices` specifices the offset, in bytes, from the beginning of the data type where the current element is located. `oldTypes` is a list of the data types for each element listed and is also an array. These are Cell Connector data types that have already been established. Note that once you have created a custom data type, it can be used to construct future data types. `size` is the total size of this custom data type in bytes.

The block of code in Figure 4.4 demonstrates its use. It creates a three element custom data type that has two unsigned ints for the first two elements, and an unsigned long long for the last element.

Node that data types have to be defined both on the manager node and on the compute node. That is, the data types have to be defined in the same order and prior to the invoking of the Cell Connector libraries on both the manager and compute nodes. Cell Connector does not yet have a way of transferring custom data types from the manager node to the compute nodes so for the moment this must fall on the application developer.

4.4.2 Manager Initialization

This section includes the code to initialize and utilize the Cell Connector framework. This is time spent reading data into the manager's memory, adding application parameters, setting up custom data types, etc. This is all done in preparation for the initial execution of Cell Connector. This code is located in the main function on the manager and is considered everything up to the point where Cell Connector is invoked.

A few things that can be done here is the inclusion of parameters and the creation of custom Cell Connector data types. The creation of custom data types was covered in the previous

Figure 4.4: Example of the `createType` function.

```
// Create the average data type (which is an average id, current coordinate,
// and number of points that coordinate is based on)

#define DIMENSION 3
#define NUM_MEANS 100

struct average_t {
        unsigned int id;
        unsigned int count;
        unsigned long long coord[DIMENSION];
};

struct average_t means[NUM_MEANS];
unsigned int blocklens[3], indices[3];
Cell_Con_Type oldtypes[3], average;

blocklens[0] = 1;
indices[0] = 0;
oldtypes[0] = CELL_CON_UNSIGNED | CELL_CON_INT;

blocklens[1] = 1;
indices[1] = sizeof(unsigned int);
oldtypes[1] = CELL_CON_UNSIGNED | CELL_CON_INT;

blocklens[2] = DIMENSION;
indices[2] = 2*sizeof(unsigned int);
oldtypes[2] = CELL_CON_UNSIGNED | CELL_CON_LLONG;

average = createType(
        3,                          \\ Total elements
        blocklens,          \\ Size of each element
        indices,            \\ Byte location of each element
        oldtypes,           \\ List of element data types
        2*sizeof(unsigned int)+DIMENSION*sizeof(unsigned long long) \\ Total Size
);
```

section, however the use of parameters remains untouched this far. The need for parameters arise when an application requires some global information to be sent to all compute nodes prior to any computation. Cell Connector provides this via the `addParameter` function.

In Figure 4.5 we can see the prototype of the `addParameter` function. The first argument of this function is the `parameter` pointer. This points to any data that you would like to send to all compute nodes, where they'll be able to access it in their offload function. The next argument is `type` and this simply tells Cell Connector what is the data type of this parameter. The last argument is the size of this parameter. This is the byte size of the data type pointed to by `void *parameter`. A user program can call this function as many times as it requires to add multiple parameters that will be broadcast to all available compute nodes. Discussion on how to access these parameters on the compute node's offload function follows in the next section. An example of `addParameter` in use can be found in Figure 4.6. It uses a custom data type that was actually constructed in the code example found in Figure 4.4.

Figure 4.5: Prototype of the `addParameter` function.

```
void addParameter(
        void *parameter ,
        Cell_Con_Type type ,
        unsigned int size
);
```

Figure 4.6: Example of the `addParameter` function.

```
for(i = 0; i < NUM_MEANS; i++) {
        means[i].id = i;
        for(j = 0; j < DIMENSION; j++) {
                means[i].coord[j] = rand() % GRID_SIZE;
        }
        means[i].count = 0;
}
addParameter(means, average, NUM_MEANS);
```

4.4.3 Compute Node Offload Function

The code that runs on the compute node and does the actual work of the application is in the Offload Function. This function needs to be implemented in the user's application code. The function must match the prototype shown in figure 4.7.

The parameters passed into the function are all descriptive of the workload received and should be considered in-out parameters. The *buffer* parameter is actually a pointer to a pointer where the workload is located. This is done incase the memory has to be modified for the output, i.e. if the output requires more space, the buffer can be reallocated and the new memory address location (if it changes) will be passed out of the function and back into the Cell Connector libraries when the function exits. The *type* parameter describes the type of data that is contained in the buffer. The *size* parameter describes how many data types of *type* are in the buffer. Like noted for the *buffer* parameter, *type* and *size* are pointers so they can be used as out variables as well as in variables incase the computational work done to the data requires changing these parameters of the work unit.

As mentioned in section 4.4.2, sometimes applications require certain pieces of data to be distributed among all compute nodes that can't be configured at compile-time. The answer to this in Cell Connector is the use of parameters implemented via the `addParameter` function on the manager node. These parameters can be accessed on the compute nodes by directly accessing provided parameter variables. Each parameter that is added by the manager can be found in the `params` array of pointers. So the first parameter added can be found at `params[0]`, while the second can be found at `params[1]` and so on. The information about the types of data found in these parameters can be found in the `paramTypes` array, while

Figure 4.7: Prototype of the offload function on the accelerator node.

```
void offloadBuffer(void** buffer, Cell_Con_Type *type, int *size) {

}
```

Figure 4.8: Use of parameters in the compute node's offload function.

```
void offloadBuffer(void** buffer, Cell_Con_Type *type, int *size) {
        // Creating pointers to access the workload buffer
        unsigned char *points = *((unsigned char **)buffer);
        struct average_t *outputMeans = *((struct average_t **)buffer);

        // Accessing the parameters
        unsigned int numAvgs = (unsigned int)paramSizes[0];
        struct average_t *newAvgs = (struct average_t *)params[0];
}
```

the sizes can be found in the `paramSizes` array. An example of the use of these parameters can be found in figure 4.8.

4.4.4 Manager Node Merge Function

The code to merge partial results from each compute node into a complete result set. This is called every time a result is received from a compute node. This function is exclusive to only one thread at a time in order to avoid any race conditions on the manager when merging results into the main data set. An example implementation of this function can be found in figure 4.9.

While it is somewhat similar to the Offload function on the compute node found in figure 4.8, the parameters for the merge function are all in parameters and it is up to the programmer to merge these results into some global solution set. In the case of figure 4.9 we are adding all the averages received into a global data structure, *sums*.

4.5 Implementation

Cell Connector has been implemented as lightweight static libraries for each of the platforms, i.e., x86 on the manager and PowerPC on the compute nodes, using only about 1400 lines of C code. The libraries provide the application programmers with necessary constructs for using Cell Connector.

A goal of this implementation is to maintain a constant memory footprint and keep the

Figure 4.9: An example merge function on the manager node.

```
void merge(void *buffer, Cell_Con_Type type, int size) {
        struct average_t *avg = (struct average_t *)buffer;
        int i = 0, j = 0;
        unsigned int curID;

        for(i = 0; i < size; i++) {
                curID = avg[i].id;
                if(avg[i].count != 0) {
                        for(j = 0; j < DIMENSION; j++) {
                                sums[curID].coord[j] += avg[i].coord[j];
                        }
                        sums[curID].count += avg[i].count;
                }
        }
}
```

memory pressure in check on the compute nodes even with large input data. This required some experimenting to determine the number of buffers to use on the compute nodes: too few buffers and the nodes will have to stall for data to be transferred from the manager, too many and the application has less memory to use for computation. In either case, performance is reduced. Initially we started by giving each of the three threads on a compute node a dedicated buffer. However, during the course of our development we observed that for the majority of applications, the computation time far outweighs the communication time and the output data was considerably less than the input data. Thus, the Receiving and Writing threads spend only a fraction of the total time working with their associated buffers. So, we decided to eliminate one of the buffers. Having two buffers gives the compute node more memory to use for computation but still allows overlapping the communication with computation. Moreover, to accommodate for applications that do not exhibit such behavior, we do allow the users to modify the number of buffers as necessary.

Another implementation decision is determining how best to transfer data between the compute nodes and the manager. In an initial design, the manager provided a compute node with information regarding input file location, starting offset, and size of the chunk to process. Each compute node would access the same NFS server in order to ensure they were all accessing the same input files. The compute node would then use this information and read the file chunk into its memory. However, the large number of compute nodes created contention at the NFS server and increased the I/O times for all nodes. Moreover, applications typically reuse data, and this approach required rereading of data from disk whenever it was needed. We addressed this by letting the manager read the input data in its memory, which is then distributed to compute nodes using any standard communication scheme. We used MPI [25] in our implementation due to its proven performance and our familiarity with it.

While implementing Cell Connector, instead of creating a hard dependency on one specific communication mechanism (such as LAMMPI, which is what was used during development), the communication functions used were of a generic nature. This way various files can be written defining these communication functions using different communication mechanisms. For example, we can have one file define these functions with MPI functions, and another file using DaCS functions [12].

Chapter 5

Evaluation

In this section, we detail our evaluation of Cell Connector using the implementation of Section 4.5. We describe our experimental testbed, the benchmarks that we have used, and present the results.

5.1 Experimental Setup

Our testbed has eight Sony PS3s as compute nodes connected via 1 Gbps Ethernet to a manager node. The manager has four quad-core AMD Opteron 8350 processors, 8 GB main memory, 280 GB hard disk, and runs Ubuntu Linux 8.04. The PS3 is a hypervisor-controlled platform, and has 256 MB of main memory (of which only about 200 MB is available for applications), and a 60 GB hard disk. Of the 8 SPEs of the Cell, only 6 SPEs are visible to the programmer [30, 23] in the PS3. Moreover, each PS3 node has a swap space of 512 MB, and runs Linux Fedora Core 7.

We used three different configurations of our resources for the experiments. (1) *Single* configuration that runs the benchmarks on a stand alone PS3, with data provided from an NFS server to factor out any effects of the PS3's slower local disk. *Single* provides a measure of one PS3's performance in running the benchmarks. (2) *Basic* configuration that uses all nodes. In this case, the manager equally divides the input at the beginning of the job and assigns it to the PS3s in one go. The manager then waits for the PS3s to process the data, before merging their output to produce the final results. *Basic* serves as the baseline for a more traditional cluster configuration with no concept of work units. (3) *Cell Connector* configuration that also uses all nodes but employs Cell Connector for work unit scaling and scheduling. These configurations allow us to study various aspects of Cell Connector in detail.

5.2 Methodology

For the evaluation conducted, applications with various implementations we used to demonstrate the performance of Cell Connector with both compute-bound and communication-bound workloads. The first set of applications are a nice variety of applications in terms of behavior and memory usage, and are implemented using a MapReduce implementation for the Cell processor [13]. Another application is used that was hand optimized for the Cell processor, a Jacobi stencil code. This code was added to demonstrate the point where a very fast and highly optimized code would see benefits from using the Cell Connector framework despite any extra overhead added.

- *Linear Regression:* This application takes as input a large set of 2D points, and determines a line of best fit for them. The work done on the compute nodes involves creating 5 different sums from each set of points. These sums are: the sum of all X coordinates, the sum of all Y coordinates, the sum of each X coordinate squared, the sum of each Y coordinate squared, and finally the sum of each X coordinate multiplied by its respective Y coordinate. Thus the input data of each compute node is a large array of coordinates, and the output is 5 sums. The merge function on the manager simply adds the 5 sums from each workload into 5 global sums in the manager node's memory. When all execution has been finished, the manager node can do approximately 20 operations on these numbers to generate all the numbers required of a linear regression.

  Thus the notable features of this application are:

  - Large amount of input data on the manager node
  - Large amount of input data per work unit
  - Average amount of computation on each compute node
  - Very little amount of output data per work unit
  - Very little computation in the manager node's merge function

- *Word Count:* This application counts the frequency of each unique word in a given input file. The input of each compute node is a large amount of data, a section of the input text file read on the manager node. The work done on the compute nodes involve summing together the number of times each word appeared in that compute node's current work unit. The output is a list of unique words found in the input along with their corresponding occurrence counts and can vary in side from medium to large depending on how many words repeat in that current work unit. The manager maintains a hash table of all words and their occurrences. The merge function also varies in work, as it can take an average amount of time to a large amount of time depending on how many new words have to be inserted into the hash table.

- Large amount of input data on the manager node
  - Large amount of input data per work unit
  - Average amount of computation on each compute node
  - Average to large amount of output data per work unit
  - Average to large amount of computation in the manager node's merge function

- *Histogram:* This application takes as input a bitmap image, and produces the frequency count of each color composition in the image. The format of any 24-bit bitmap image includes a few bytes of header information, followed by 3 bytes of information per pixel[2]. Each byte represents the strength of its corresponding color (red, green, or blue) which ranges from 0-255. The manager reads in the bitmap image passes it to the Cell Connector library along with a maximum work unit size. Each compute node generates 768 sums. We're looking to see how often each color saturation appears in the image (3 colors * 256 different levels of saturation). The manager node's merge function adds these 768 totals from each work unit into 768 global sums in the manager node's memory.

  - Large amount of input data on the manager node
  - Large amount of input data per work unit
  - Average amount of computation on each compute node
  - Small amount of output data per work unit
  - Small amount of computation in the manager node's merge function

- *K-Means:* This application takes a set of points in an N-dimensional space and groups them into a set number of clusters with approximately an equal number of points in each cluster.

  This application is quite a bit different from most of the other applications in that it is more complicated to understand, and it is of an iterative nature. The way this application works is a set of points are provided and will remain constant throughout the execution of the program. The goal of the program is to create clusters of these points so that all the clusters are of approximately equal size.

  As mentioned before, this application is iterative in nature. We initially pick a set of random *cluster points*, or points that represent the center of a cluster. Then each iteration these *cluster points* shift a little bit towards the best solution. When this shift stops or becomes very small, the iterations stop and the application has completed.

  In each iteration, the manager distributes the current set of *cluster points* in the problem space to every compute node. The manager node then reads in the set of points and hands them off to the Cell Connector libraries to begin breaking them up into work units for the compute nodes. Each compute node determines which *cluster point* each input point is closest to. It averages together these points along with the weight of that average (the number of points that went into that average) and these averages are the output of each work unit. There is one average per *cluster point* and these represent where each *cluster point* could move to at the end of the iteration. The merge function on the manager node adds together a total of all the averages, and their weights, to a global average for each *cluster point*. These global averages become the new *cluster points* in the next iteration. This continues until the size of the *cluster point* update is zero or very small.

  - Large amount of input data on the manager node
  - Large amount of input data per work unit
  - Large amount of computation on each compute node
  - Small to medium amount of output data per work unit
  - Small amount of computation in the manager node's merge function

- *Jacobi:* This application performs work on a grid of points. It is an iterative application which updates each point in the grid to the average of all of its neighbors. The simplest example would be a grid of points where every point is initialized to 0.0, while all the border values are updated to 1.0. As Jacobi iterates, it will slowly update every point to a value close to 1.0.

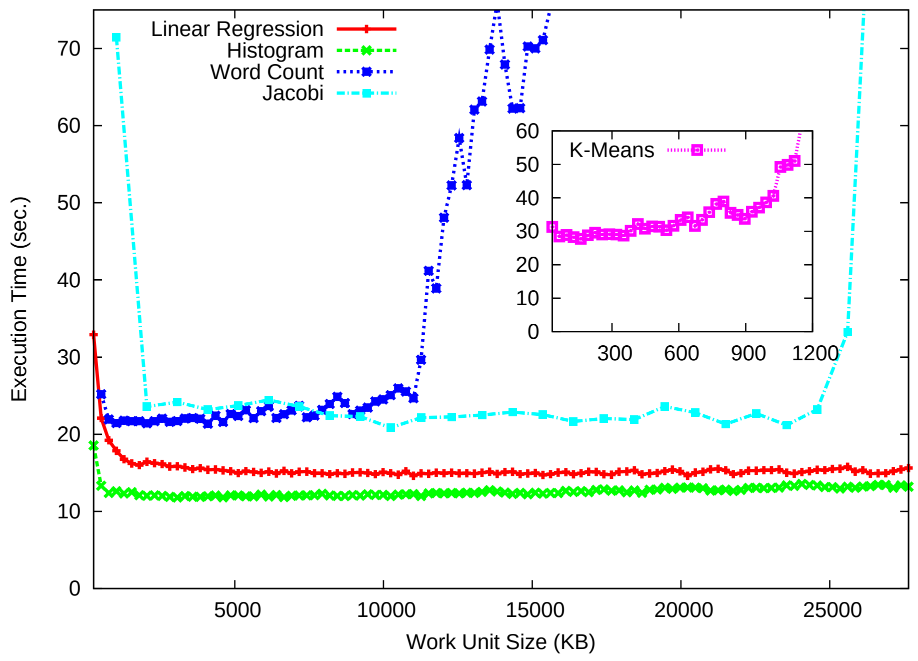
  Implemented with Cell Connector, the whole grid is maintained on the manager node. Work units are sets of rows that are calculated on the compute nodes. However, doing the work partitioning this way means the values in the top and bottom rows of each work unit are not properly updated, since they depend on values that compute node does not have. In order to resolve this, the manager updates those values inside of its merge function. The amount of work for these calculations is minimal compared to total size of the data set, so the manager can handle this workload.

  This application is the only application not implemented with a programmer-friendly library on the compute nodes. Instead it was carefully hand-optimized for the fastest possible computational speed on the Cell BE.

  Traditionally, Jacobi iterates until the residual value becomes very small. However to record accurate measurements in the results section, a set number of iterations was used.

  - Large amount of input data on the manager node
  - Medium amount of input data per work unit
  - Medium amount of computation on each compute node, although this computation is done very quickly
  - Medium amount of output data per work unit

Figure 5.1: Effect of work unit size on execution time.



– Medium amount of computation in the manager node's merge function

For each benchmark, the total execution time under our setup configurations is measured.

5.3 Results

In this section, we first examine how work unit size impacts total execution time and use this to pick the best unit size for the benchmarks. Next, the benchmarks are ran under the setup configurations, measuring performance as input size is scaled. Then we study the impact of Cell Connector on the manager. Finally, we determine how Cell Connector scales as the number of compute nodes is increased.

5.3.1 Work Unit Size Analysis

Figure 5.1 helps us determine what is the correct work unit size for an application. Too small of a work unit and the communication overhead becomes dominate. Too large of a work unit and memory pressure on the compute node causes thrashing. The goal is to find the right point where communication is done efficiently and we maximize the available memory on the compute node without causing any sort of paging.

This experiment held the input size constant for each application across the various work unit sizes. An input size was chosen for each application that was either the largest input size, or a large enough input size to allow each application to run in approximately the same time range.

Looking at figure 5.1 shows there isn't a clear cut work unit size that works best for every application. A range of work unit sizes from 256KB to 30MB were sampled for Linear Regression, Histogram, and Word Count applications, sizes from 32KB to 2MB were sampled for K-Means, and sizes from 1MB to 27MB were sampled for Jacobi. While there are sizes that perform the best, they usually beat out other sizes by a very small fraction. What is seen here is a decent range of very usable work unit sizes to pick from. The programmer does need to have some idea of how much data a single compute node can handle. This shouldn't be a problem if the programmer is familiar enough with the application, otherwise some exploration trials are needed. Another thing to note is as some of the applications use work unit sizes that relatively large to the input size, the offset work unit(s) skews the execution time more and more. The offset work unit(s) is the last work unit(s) when the input size doesn't cleanly divide by the work unit size specified. In this case, one more distribute and collect operation has to be performed by the manager and this can cause a compute node load imbalance. Some of the nodes might not get any work while one or few nodes get the offset work unit(s). To avoid this, a smaller work unit size is preferred when there is a long range of work unit sizes with similar execution times.

For the rest of the benchmarks we use the results in figure 5.1 to determine the best work unit sizes. The best work unit size for Linear Regression is 8 MB; for Histogram is 3 MB; for Word Count is 4 MB; for K-Means is 160 KB; and for Jacobi is 10 MB.

5.3.2 Benchmark Performance

Linear Regression For this benchmark, we chose input sizes ranging from 2^{22} points (file size 4 MB) to 2^{29} points (512 MB). Figure 5.2 shows the results. Under *Single*, the input data quickly becomes larger than the available physical memory, resulting in increased swapping, and consequently increases the execution time. In *Basic*, a smaller fraction of the data is sent to each of the PS3s, which relieves the memory pressure on them somewhat. Initially, Cell Connector performs slightly better than *Basic*, 23.2% on average for input size less than 400 MB, mostly due to its data streaming characteristics. However, as the input size is increased beyond 400 MB, the peak virtual memory footprint for *Basic* is observed to grow over 376 MB, much greater than the 200 MB available memory, leading to increased swapping. Once *Basic* starts to swap, its execution time increases noticeably. In contrast, Cell Connector is able to keep the memory pressure constant among various input sizes to avoid swapping on the PS3s, thus, achieving 29.3% average speedup across all the considered input sizes compared to *Basic*.

Figure 5.2: Linear Regression execution time with increasing input size.

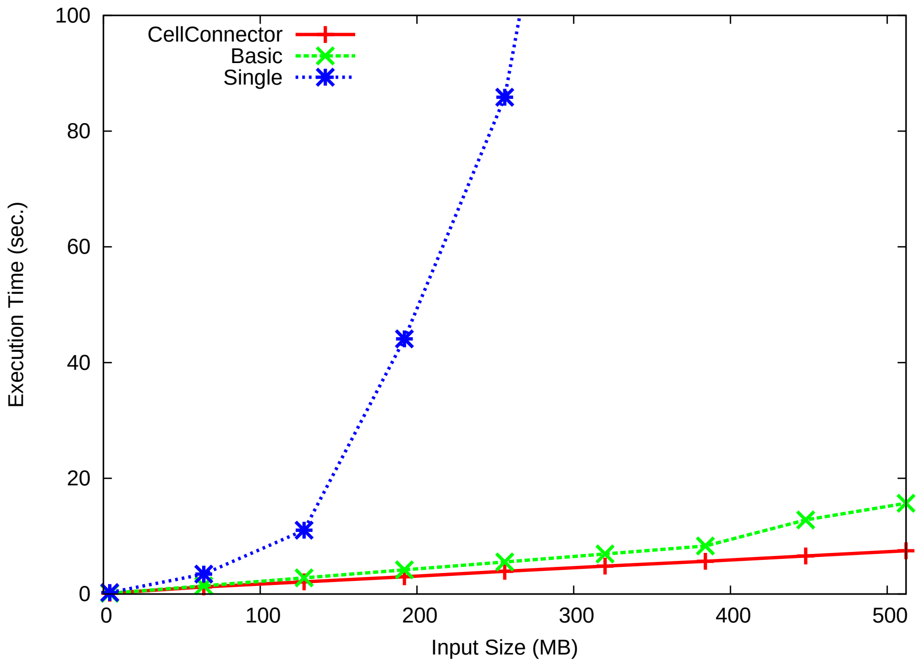


Figure 5.3: Word Count execution time with increasing input size.

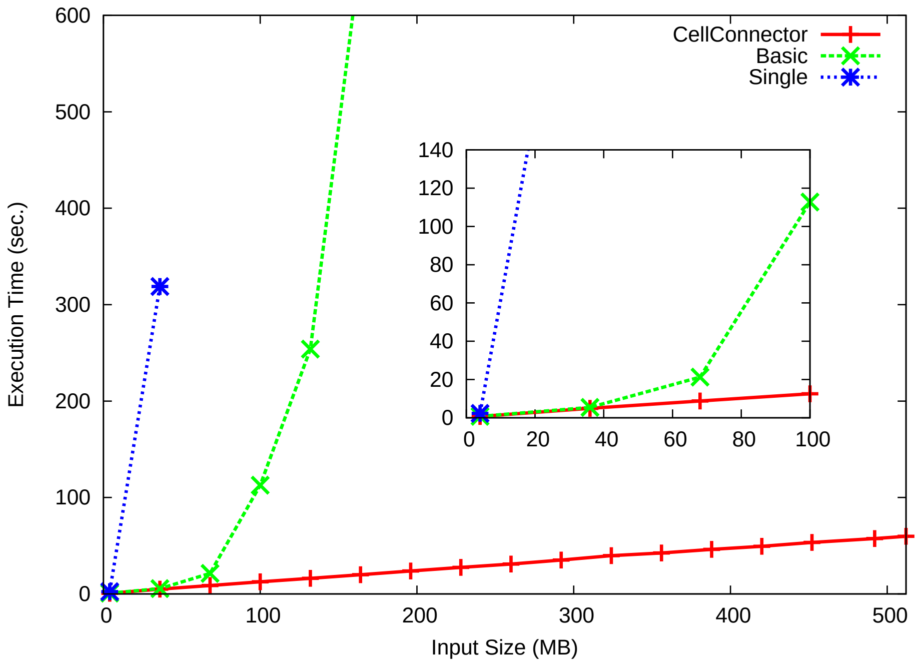
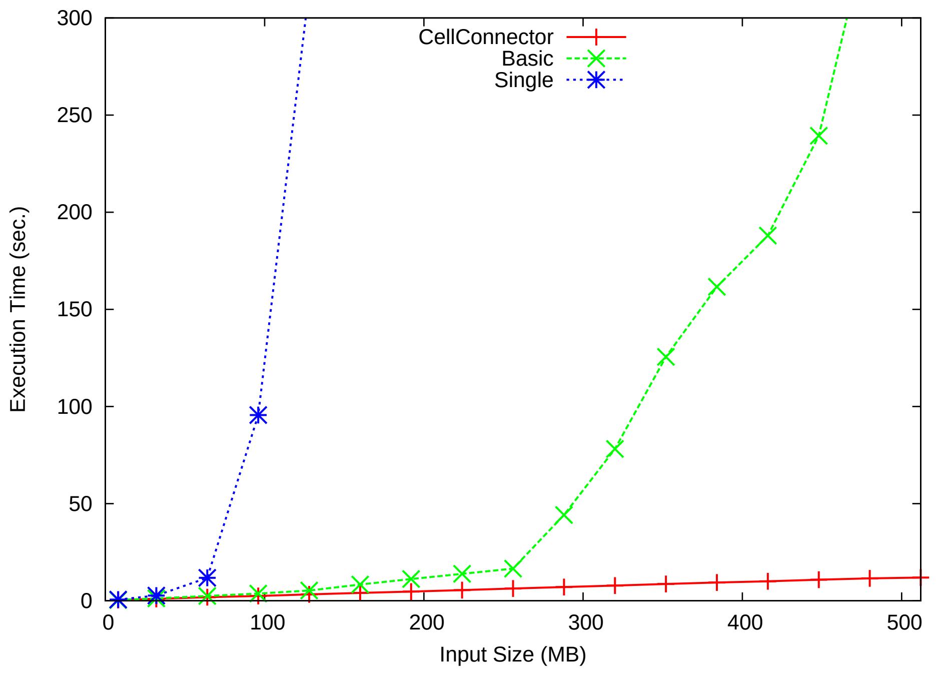


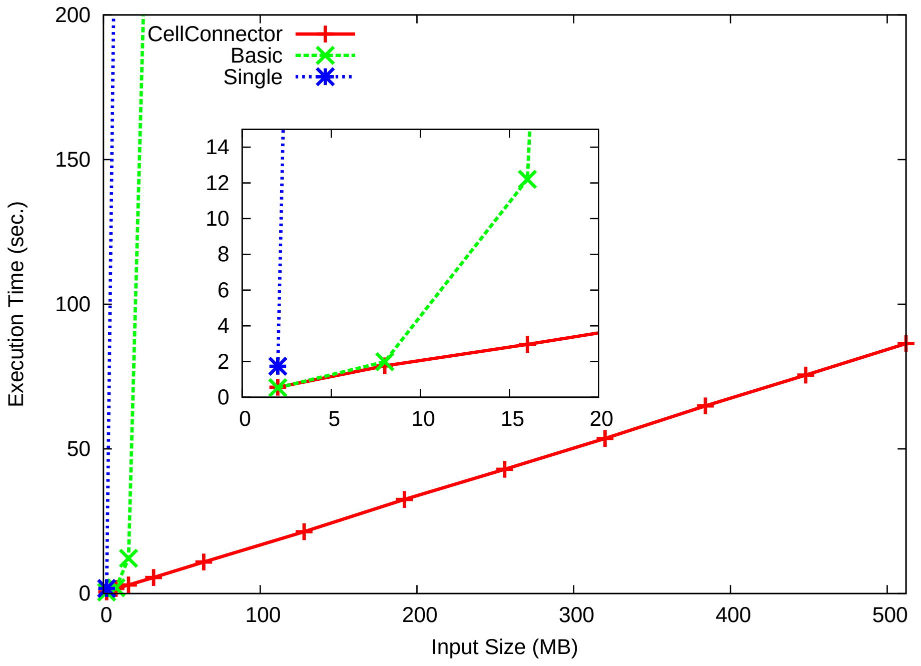
Figure 5.4: Histogram execution time with increasing input size.



Word Count During our experiments with Word Count, we observed exponential growth in memory consumption relative to the input data size, which is the result of the MapReduce framework used on the compute nodes. Therefore, for any input size greater than 44 MB, *Single* experienced excessive thrashing that caused the PS3 node to run out of available swap space (512 MB) and ultimately crash. Similarly, *Basic* was also unable to handle input data sizes greater than $(44 * 4 =)$ 176 MB, and took an enormous 662.9 seconds for an input size of 164 MB. Figure 5.3 shows the results. Here, Cell Connector is not only able to process any input size, it outperforms *Basic* by 65.3% on average for the points, emphasized in the inset in the figure, where *Basic* completed without thrashing (input data size $<$ 96 MB).

Histogram Figure 5.4 shows the result for running Histogram under the three test configurations. It can be observed that Cell Connector scales linearly with the input data size. In contrast, *Basic* only scales initially, but then looses performance as the increased input size triggers swapping, e.g., for an input size of 288 MB it takes 44.1 seconds to complete, as opposed to 16.6 seconds for an input size of 256 MB, a change of 32 MB. On average across all input points less than 256 MB, Cell Connector does 36.6% better than *Basic* for Histogram. The maximum input size that *Single* can handle without crashing is 128MB, while *Basic* can handle 512 MB, for which the execution times are 315.0 and 472.8 seconds, respectively.

Figure 5.5: K-Means execution time with increasing input size.

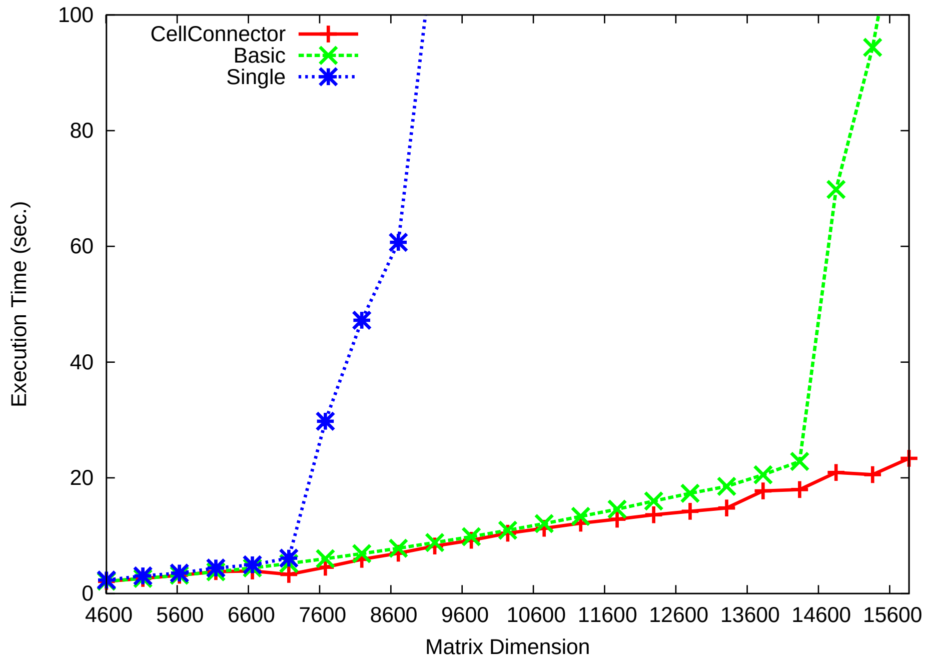


K-Means The results for the K-Means benchmark are shown in Figure 5.5. Note that K-Means use a different number of iterations for different input sizes. Therefore, considering total execution times for different inputs does not provide a fair comparison of the effect of increasing input size. We remedy this by reporting the execution time per iteration in the figure. While the Cell Connector implementation scales linearly, *Single* and *Basic* use up all the available virtual memory with relatively low input sizes. Both *Single* and *Basic* crash for an input size greater than 8 MB and 32 MB, respectively. For 32 MB input size, *Basic* takes over 330 seconds/iteration compared to 5.5 seconds/iteration of Cell Connector. The only input size where the *Basic* case doesn't thrash is for 2 MB. In this instance *Basic* outperforms Cell Connector by 5.7%, as this input size is too small to amortize the management function overhead of Cell Connector. However, this is not of concern, as with any input size greater than 2 MB, Cell Connector does significantly better than *Basic*.

Jacobi The results for the Jacobi benchmark are shown in Figure 5.6. Since the total data set is a large matrix of values, the input size ranged from a 4608x4608 matrix using single-precision floating point elements up to a 15872x15872 element matrix. Since Jacobi is an iterative application, these timings are time spent executing per iteration. Like all the other applications, Cell Connector scales linearly while *Single* and *Basic* fall victim to thrashing before hitting the maximum input size. Cell Connector out performs both *Single* and *Basic* even before thrashing occurs by 11.6% and 14.1% respectively.

An important thing learned from the Jacobi trial is Cell Connector's performance on com-

Figure 5.6: Jacobi execution time with increasing input size.



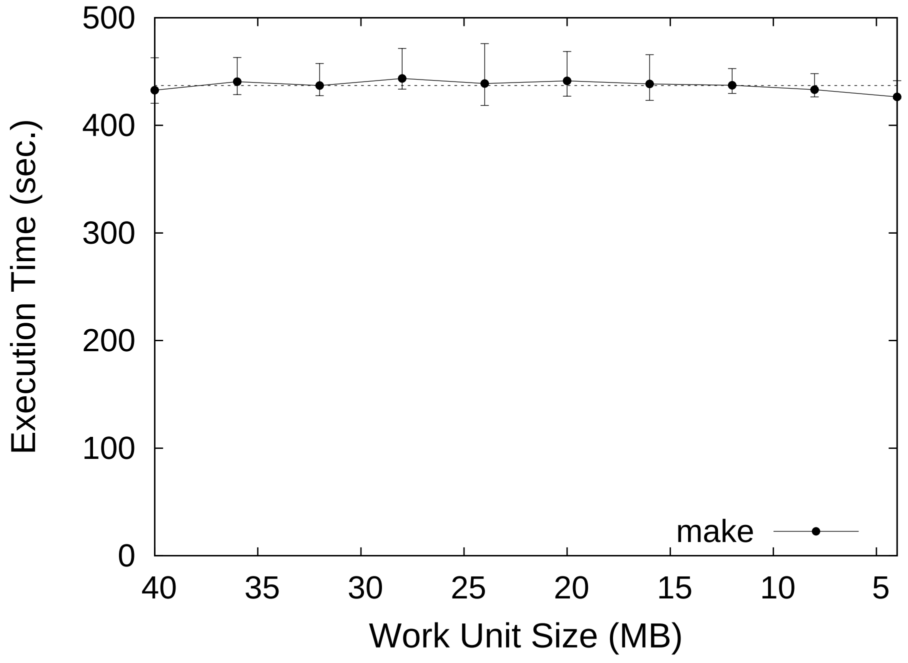
munication bound applications. Ideally, Jacobi would not use a framework such as Cell Connector as only about 2-5% of the total execution time is spent computing on the compute nodes. However, showing that Cell Connector's streaming approach can produce faster results than the other configurations on a communication dominant application is a statement about the communication efficiency of Cell Connector.

In summary, memory constraints non-withstanding, Cell Connector outperforms static data distributions due to better overlap of computation with communication. Cell Connector also improves memory usage and enables efficient handling of larger data sets compared to static data distribution approaches.

5.3.3 Impact on the Manager

In this experiment, we determine the affect of varying work unit sizes on manager performance. This is done as follows. First, we start a long running job (Linear Regression) on the cluster. Next, we determine the time it takes to compile a large project (Linux kernel 2.6) on the manager, while the application is running. We repeat the steps as the work unit size is decreased, potentially increasing the processing requirements from the manager. For each work unit size, we repeat the experiment 10 times, and record the minimum, maximum, and average time for the compilation as shown in Figure 5.7. The horizontal dashed line in the figure shows the overall average of average compile time across all work unit sizes.

Figure 5.7: Impact of work unit size on the manager.

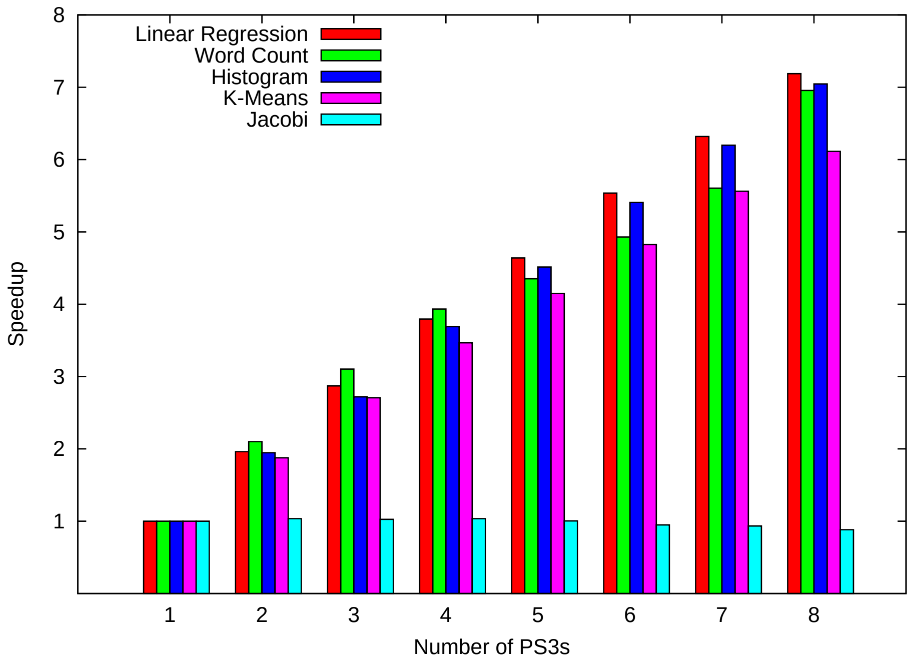


Given that the overall average remains within the minimum and maximum times, we can infer that the variations in the compile time curve is within the margin of error. Thus, the relatively flat curve indicates that Cell Connector has a constant load on the manager and our framework can support various workloads without the manager becoming a bottleneck.

5.3.4 Scaling Characteristics

In the next experiment, we observed how the performance of our benchmarks scale with the number of compute nodes. Figure 5.8 shows the speedup in performance normalized to the case of 1 compute node. We use the same input size for all runs of an application. The input sizes for the different applications are chosen to be large enough to benefit from using 8 nodes: 512 MB for Linear Regression and Histogram, 200 MB for Word Count, 128 MB for K-Means, and a 15872x15872 matrix for Jacobi. The bars of K-Means and Jacobi are based on time per iteration, as explained earlier in this section. The figure shows that Cell Connector scales almost linearly as the number of compute nodes is increased, and this behavior is true for almost all the benchmarks. However, with the exception of Jacobi, we observed that the improvement trend does not hold for all benchmarks when the eighth node is added. Upon further investigation, we found that the network bandwidth utilization for such cases was quite high, as much as 107 MB/s compared to the maximum observed value of 111 MB/s on our network, measured using remote copy of a large file. This is further enforced by the fact Jacobi, a communication bound application with even one node, seems

Figure 5.8: Effect of scaling on Cell Connector.



to not scale at all as more nodes are added. This is to be expected for a communication bound application, especially when the manager and compute nodes both have a gigabit ethernet interface to the network. If the manager node had a 10-gigabit interface then the addition of extra compute nodes would in fact produce speed up.

This introduced communication delays even with double buffering, and prevented Cell Connector from achieving a linear speedup. However, if the ratio of time spent in computation compared to that in communication is high, as is the case in scientific applications, near perfect speedup can be obtained. We tested this hypothesis by artificially increasing our compute time for Linear Regression by a factor of 10, which resulted in a speedup of 7.8.

5.3.5 Summary

Our evaluation of Cell Connector using a variety of benchmark applications on an asymmetric cluster with 4 PS3 compute nodes shows that, compared to a non-streaming approach, Cell Connector achieves 50.5% better performance on average. Moreover, Cell Connector adapts effectively to the relative computation to data transfer density of applications by converging optimal work unit size, minimally loads the manager, and scales well with increasing number of compute nodes (a speedup by a factor of 6.9 on average for an 8 node cluster). Thus, Cell Connector provides a viable framework for efficiently supporting large data applications on asymmetric HPC clusters.

Chapter 6

Cell Stream

The ideas and concepts of Cell Connector can extend to the architecture of a single Cell Broadband Engine chip. As pointed out in section 3.2, the Cell consists of a single conventional processor(Power Processing Element - PPE) connected to eight specialized accelerator processors (Synergistic Processing Elements - SPEs) with a high-bandwidth bus. This chapter introduces, describes, and evaluates this single chip inspiration of Cell Connector called CellStream.

CellStream is a framework for the Cell Broadband Engine that moves data from an external storage quickly and efficiently between SPEs and memory and back out to a storage device. It also supports a drop in kernel on the SPEs so some work can be done on the data as it is streamed through the SPEs.

It has been used as a test application, when used without a computational kernel, comparing various programming models on multi-core processors with explicitly managed memory hierarchies (such as the Cell BE)[32]. In that paper, it was designed to compare how efficient each programming model was at streaming data to the SPEs. Since the handwritten version of CellStream is able to perform at close to peak speeds, it was a good indicator of the overhead introduced with the different programming models.

6.1 Design

On the PPE side of CellStream, there are three threads (implemented with pthreads) that run concurrently on three different buffers.

The Reading Thread is responsible for moving data from the external source into Main Memory (Figure 6.1). The Offloading Thread is responsible for dividing up work among the SPEs, start their execution, and wait for them to finish before handing the buffer to the last thread (Figure 6.2). Since the Offload Thread does very little work on the actual PPE, it is

Figure 6.1: Reading thread reads the first buffer into Main Memory

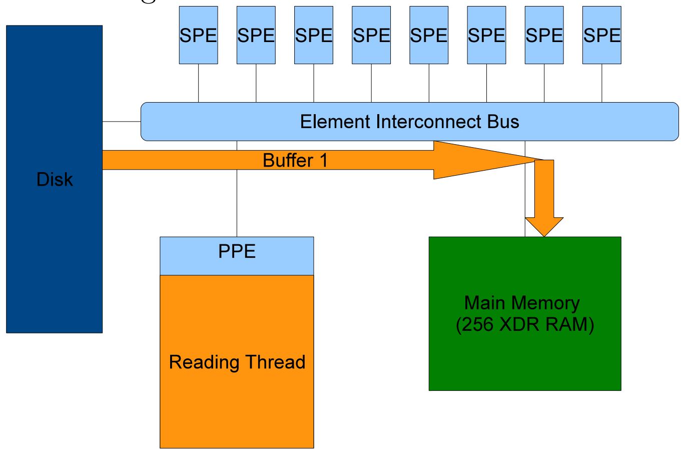


Figure 6.2: First buffer is offloaded to the SPEs while the Reading Thread reads in the second buffer from disk.

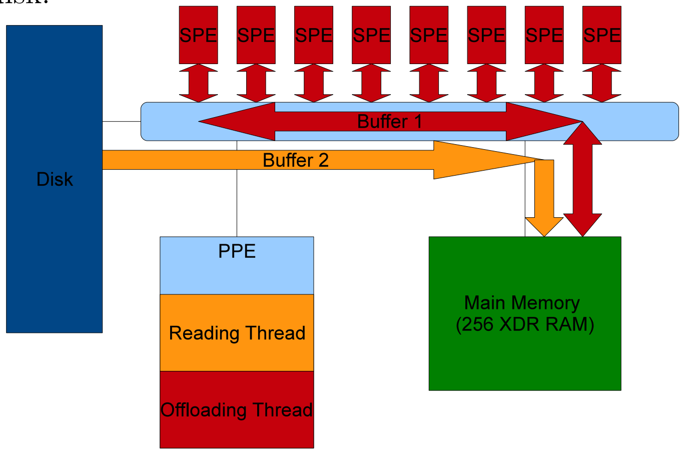


Figure 6.3: Writing Thread writes first buffer back to disk

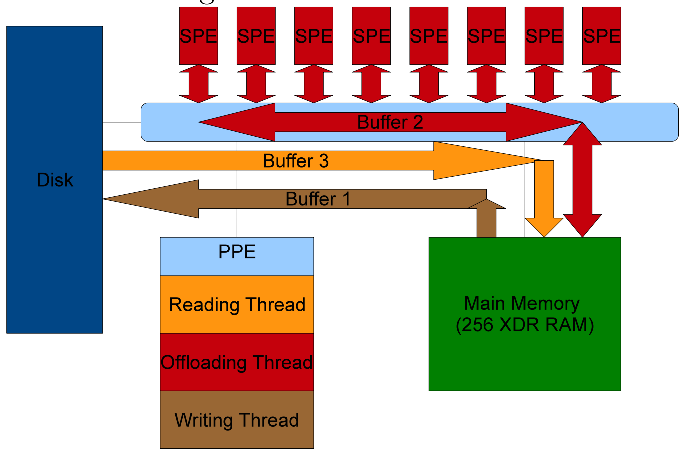
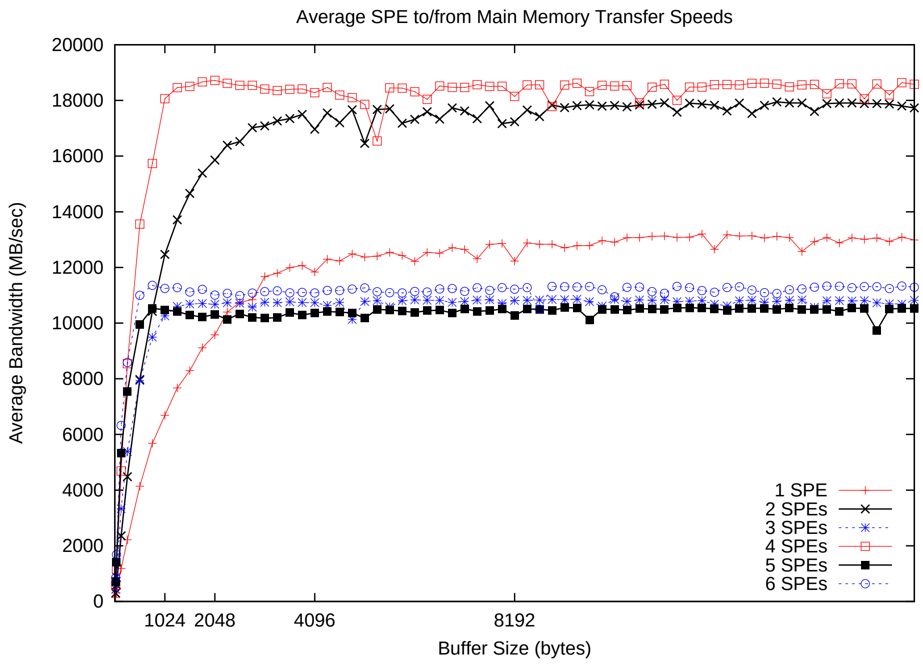


Figure 6.4: CellStream Average SPE Transfer Speed on a Playstation 3 with workloads split on 16-byte boundaries

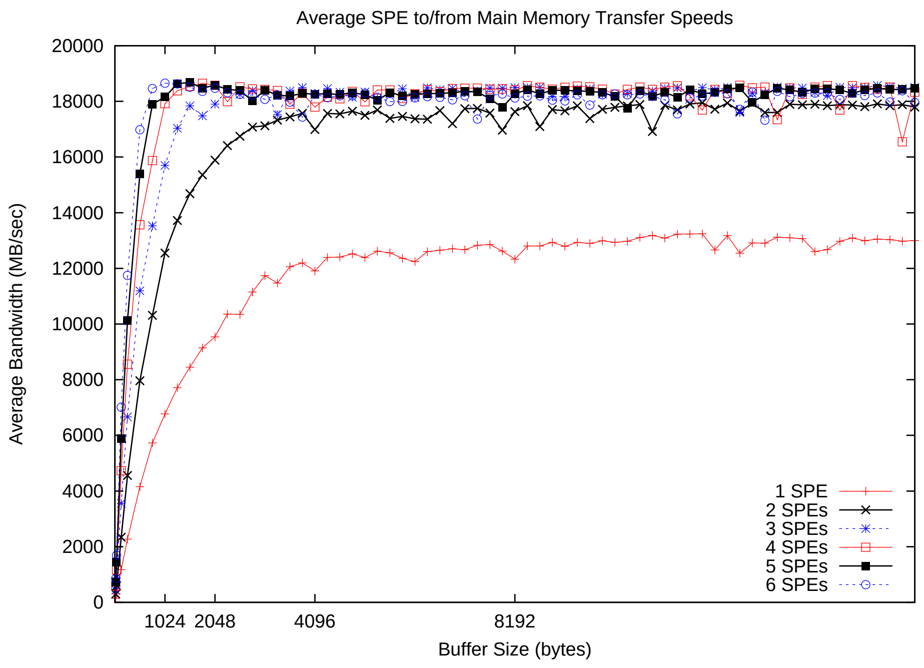


not shown in the figures. The Writing Thread is responsible for moving the now processed data in Main Memory to an external destination (Figure 6.3).

The threads rotate which buffer each is responsible for using pthread wait functions. As soon as the Reading thread fills a buffer, it signals the other threads to wake up and check who has permission to the modified buffer. Since the Reading thread will specify when the buffer is ready to be offloaded, the Offloading thread will wake on this signal and begin offloading the buffer to the SPEs. The same signal and wake up routine happens between the Offloading thread and Writing thread, along with between the Writing thread and Reading thread once a buffer has been written back to external storage. This version of CellStream uses files on disk as the main source of input and the destination of output, although the Reading and Writing threads can be modified to use other kinds of storage (such as the network).

The main concern of the SPE code is to move data in and out of the local store as efficiently and quickly as possible. It utilizes a double buffering technique with DMA fencing. The fencing calls tell the SPE as soon as one chunk is committed to main memory it should start filling it with a new buffer chunk. These calls can be issued all at once and they will be processed in the order they are issued.

Figure 6.5: CellStream Average SPE Transfer Speed on a Playstation 3 with workloads split on cache lines



6.2 Performance

While the DMA benchmark that comes bundled with the Cell SDK measures the maximum communication capabilities of the hardware[21], it does so in a manner that does not maintain data coherency. CellStream was designed to get as close to the maximum bandwidth as possible while operating on data that remains coherent and uncorrupted as it passes through the Cell BE.

The first experiment was done to verify CellStream exhibited the same communication behavior as the DMA benchmark[21] which varied the size of each DMA transfer to find the ideal DMA transfer size. The number of SPEs used was scaled from 1 to 6 (not eight since a Playstation 3 only has 6 available SPEs). Each buffer was split according to how many SPEs would be working on a buffer so each SPE had an equal share of work.

The results shown in Figure 6.4 are surprising. For the case of 1, 2, and 4 SPEs, the results are what was expected. However the results for 3, 5, and 6 SPEs were very low. After further investigation it was revealed that in the cases of 3, 5, and 6 SPEs, the transfer speeds were not equal across all SPEs.

The first SPE would always perform the fastest, but others would generally perform twice as slow. This behavior was consistent when different SPEs worked on different parts of the buffer, such that the SPE that started work at the beginning of the buffer always had transfer speeds that were almost twice that of some of the other SPEs. Since the buffer was split up so that each SPE's starting memory address began on a 16-byte aligned address, part of the requirements of a DMA transfer, some SPEs were not starting on a cache line, which is on a 128-byte boundary. Whenever we could evenly split up the 16MB buffer so that each SPE's starting work address was on a 128-byte boundary, that SPE would perform as expected.

In Figure 6.5, the performance is much closer to the results in the DMA benchmark[21] mentioned earlier. Splitting the buffer so each SPE's first memory access was on a 128-byte memory address boundary clearly show a better average speed.

We now see what we'd expect from the hardware. A single SPE is limited by how fast that SPE alone can access memory. With 2 SPEs there is a sizable jump in average bandwidth inside of the chip. With additional SPEs we see the bottleneck of the memory interface present itself.

6.3 Summary

CellStream is a framework for the Cell BE to efficiently support communication bound applications. When used without a computational kernel, it acts as a bandwidth benchmark for the Cell BE while maintaining data coherency. The results show bandwidth close to results generated by the Cell SDK benchmark, an important result when there is no computational kernel.

Chapter 7

Conclusions

This thesis presents, explains, and evaluates two different project with very similar characteristics. Communication is an area of great importance in any kind of parallel programming, and it is even more so when asymmetric multi-core processors leave communication between cores up to the programmer. Especially when the storage resources at the lowest level is very limited. Moving data in and out of this limited storage as fast as possible becomes the bottleneck of most applications.

Cell Connector advocates a streaming approach as a solution to the problem of limited storage resources on certain high performance commodity hardware. The implementation, design, and evaluation of Cell Connector describes and shows the gains of using this streaming framework on a variety of applications. Abstracts are provided to support programmer use of Cell Connector as a viable framework for a large data application and portability is supported since Cell Connector is able to run on a variety of network communication frameworks.

CellStream addresses the limited memory available in the local store of each SPE in the Cell Broadband Engine by attaining transfer speeds that are a very high percentage of peak bandwidth. Beyond this, it supports a drop in kernel in order for certain applications to capitalize on this high data bandwidth.

These projects are able to address the problem of limited local storage resources in these high computational potential chips through the use of efficiently implemented communication mechanisms. One improves the on chip performance for applications with small data sets that can fit in a single node's memory, while another addresses the problem of when data sets grow to exceed the memory of a single node.

7.1 Future Work

Future work on Cell Connector involves a more thorough exploration of possible communication schemes and ensuring support for those. Also, additional abstracts could be added to the framework to support multiple manager nodes in a more tree-like configuration. The scalability test shows a network bottle neck as we scale up to 8 compute nodes. If we used multiple manager nodes with 8 or fewer compute nodes each, and add an additional merge step to combine the results of the multiple managers, this bottleneck can be avoided.

We have extended CellStream to maximize on-chip bandwidth for data transfers and have a preliminary implementation of CellStream which supports SPE to SPE transfers. Also, support for variable input/output sizes will be supported in the next version of CellStream.

Bibliography

[1] Astrophysicist Replaces Supercomputer with Eight PlayStation 3s. `http://www.wired.com/techbiz/it/news/2007/10/ps3_supercomputer`.

[2] Bitmap File Format. `http://www.digicamsoft.com/bmp/bmp.html`.

[3] OpenMP Specification. `http://www.openmp.org/drupal/node/view/8`.

[4] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). `http://www.amazon.com/b?ie=UTF8&node=201590011`.

[5] D. Bader and V. Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In *Proc. of the 14 th IEEE International Conference on High Performance Computing (HiPC), Lecture Notes in Computer Science 4873*, Dec. 2007.

[6] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, June 2005.

[7] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[8] L. A. Barroso, J. Dean, and U. Hï¿½lzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[9] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos. RAxML-CELL: Parallel Phylogenetic Tree Construction on the Cell Broadband Engine. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, Mar. 2007.

[10] G. Buehrer and S. Parthasarathy. The Potential of the Cell Broadband Engine for Data Mining. Technical Report TR-2007-22, Department of Computer Science and Engineering, Ohio State University, 2007.

[11] ClearSpeed Technology. *ClearSpeed whitepaper: CSX processor architecture*, 2007.

[12] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 3–12, New York, NY, USA, 2008. ACM.

[13] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell B.E. Architecture. Technical Report TR1625, Department of Computer Sciences, The University of Wisconsin-Madison, Madison, WI, 2007.

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI 2004*, pages 137–150, 2004.

[15] B. Gedik, R. Bordawekar, and P. S. Yu. Cellsort: High performance sorting on the cell processor. In *Proc. of the 33rd Very Large Databases Conference*, pages 1286–1207, 2007.

[16] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. M. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for fem calculations on a gpu-enhanced cluster. *Parallel Computing.*, 33(10-11):685–699, 2007.

[17] GraphStream, Inc. GraphStream scalable computing platform (SCP). 2006. `http://www.graphstream.com`.

[18] S. Heman, N. Nes, M. Zukowski, and P. Boncz. Vectorized Data Processing on the Cell Broadband Engine. In *Proc. of the Third International Workshop on Data Management on New Hardware*, June 2007.

[19] IBM Corp. Cell Broadband Engine Architecture (Version 1.02). 2007.

[20] Jason Cross. A Dramatic Leap Forwardï¿$\frac{1}{2}$GeForce 8800 GT, Oct 2007. `http://www.extremetech.com/article2/0,1697,2209197,00.asp`.

[21] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez. Performance analysis of cell broadband engine for high memory bandwidth applications. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:210–219, 2007.

[22] e. a. Kahle, J.A. Introduction to the cell microprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[23] J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra. The playstation 3 for high-performance scientific computing. *Computing in Science and Engineering*, 10(3):84–87, 2008.

[24] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[25] Message Passing Interface Forum. MPI2: A message passing interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):299, 1998.

[26] Mueller. NC State Engineer Creates First Academic Playstation 3 Computing Cluster. `http://moss.csc.ncsu.edu/~mueller/cluster/ps3/coe.html`.

[27] M. Pericàs, A. Cristal, F. Cazorla, R. González, D. Jiménez, and M. Valero. A Flexible Heterogeneous Multi-core Architecture. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–24, Sept. 2007.

[28] F. Petrini, G. Fossum, J. Fernández, A. L. Varbanescu, M. Kistler, and M. Perrone. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[29] K. R., D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[30] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos. Dma-based prefetching for i/o-intensive workloads on the cell architecture. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 23–32, New York, NY, USA, 2008. ACM.

[31] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos. Cellmr: A frame for supporting mapreduce on asymmetric cell-based clusters. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[32] S. Schneider, J.-S. Yeom, B. Rose, J. Linford, A. Sandu, and D. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[33] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation - a performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[34] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.