CellMR: A Framework for Supporting MapReduce on Asymmetric Cell-Based Clusters

M. Mustafa Rafique[1], Benjamin Rose[1], Ali R. Butt[1]
[1]*Dept. of Computer Science*
Virginia Tech.
Blacksburg, Virginia, USA
Email: {mustafa, bar234, butta, dsn}@cs.vt.edu

Dimitrios S. Nikolopoulos[1,2]
[2]*Institute of Computer Science*
Foundation for Research and Technology Hellas (FORTH)
GR 700 13, Heraklion Crete
Email: dsn@ics.forth.gr

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. In this paper, we take the next step by investigating the use of multi-core-based systems, especially the popular Cell processor, in a cluster setting. We present CellMR, an efficient and scalable implementation of the MapReduce framework for asymmetric Cell-based clusters. The novelty of CellMR lies in its adoption of a streaming approach to supporting MapReduce, and its adaptive resource scheduling schemes: Instead of allocating workloads to the components once, CellMR slices the input into small work units and streams them to the asymmetric nodes for efficient processing. Moreover, CellMR removes I/O bottlenecks by design, using a number of techniques, such as double-buffering and asynchronous I/O, to maximize cluster performance. Our evaluation of CellMR using typical MapReduce applications shows that it achieves 50.5% *better performance compared to the standard non-streaming approach, introduces a very small overhead on the manager irrespective of application input size, scales almost linearly with increasing number of compute nodes (a speedup of* 6.9 *on average, when using eight nodes compared to a single node), and adapts effectively the parameters of its resource management policy between applications with varying computation density.*

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 [1], [2], [3]. The recent advent of the Cell Broadband Engine (Cell) processor and GPUs as High-Performance Computing (HPC) and data processing engines [4], [5], [6], [7], [8], [9], [10], [11], 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 [10]. In the distributed setting, the accelerators are packaged compute nodes with customized components, such as GPGPU, FPGAs, Clearspeed CSX600 [12] coprocessors, IBM Cell/BE processors, and NVIDIA G8800 GPUs [13], 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.

While parallel programming models for symmetric clusters have been studied at length, the synthesis of parallel programming models for asymmetric parallel architectures is an open problem. In particular, hiding the architectural asymmetry from the programming model, and exploiting the vast computational density of accelerators while they communicate with inherently slower system components remain major challenges. Towards this end, we adapt the MapReduce [14] model for asymmetric HPC clusters boasting accelerator-type compute nodes. MapReduce is a simple model for machine-independent parallel programming at large scales. It provides minimal abstractions, hides architectural details including heterogeneity, and supports transparent fault tolerance. While current implementations of MapReduce accommodate standalone accelerators, e.g., Cell [15], and take into consideration heterogeneity of com-

pute nodes due to virtualization in the task scheduler [16], they do not cope with the asymmetry between the computational density of accelerators and data processing and forwarding capabilities of manager nodes. This asymmetry can lead to severe performance penalties by exposing communication or I/O bottlenecks.

In this paper, we design, develop and evaluate CellMR, an implementation of the MapReduce programming model on asymmetric HPC clusters with large-memory general purpose head nodes and accelerator-type compute nodes. CellMR hides asymmetry and enables high-performance, cost-effective, and scalable data processing. We target HPC clusters with heterogeneous processor architectures, similar to LANL's RoadRunner [10], built however with low-cost compute nodes that capitalize on the compute density of graphics and gaming processors. While our work can be extended to arbitrary hybrid parallel architectures, we evaluate our efforts on a cluster that uses the Cell, arguably one of the dominant asymmetric multi-core processors, as an accelerator. The novelty of CellMR lies in its use of a data streaming approach to effectively support MapReduce computations, and its adaptive resource scheduling that factors in the performance and capabilities of asymmetric components, and strives to overlap completely I/O and communication latencies. CellMR supersedes data transfer and task management libraries for asymmetric accelerator-based architectures, such as IBM's ALF [17], which delegate parameterization and optimization of scheduling data transfers to the application developers. CellMR transparently adapts the parameters of data streaming and task scheduling to the application at runtime, thereby relieving developers of some significant programming effort. CellMR also removes I/O bottlenecks via use of techniques such as asynchronous accesses and double-buffering at multiple levels of the system.

Specifically, this paper makes the following contributions:

- A detailed design of the CellMR framework that enables realizing the MapReduce programming model efficiently on asymmetric clusters comprising accelerator-type compute nodes;
- An exploration of alternative design choices for data streaming and processing and their impact on overall system performance on asymmetric architectures;
- A runtime technique for regulating data distribution and streaming in MapReduce, which is inherently "static" in the way it manages the distribution of data, so as to best bridge the asymmetry between the head node and the accelerator-type compute nodes.
- An adaptation of several well-known MapReduce applications to utilize CellMR and leverage highly cost-efficient Cell-based clusters; and
- A thorough evaluation of CellMR, in particular its data streaming and computation scheduling framework, in terms of scalability, adaptation to varying computation densities, and resource conservation capability.

Our evaluation of CellMR using representative MapReduce applications on an asymmetric cluster shows that CellMR significantly improves system performance (as much as 82.3% for Word Count benchmark) compared to the standard non-streaming scheme used in MapReduce. CellMR adapts effectively to the relative computation to data transfer density of applications by converging to existentially optimal parameters for data decomposition and streaming at runtime. Moreover, for our benchmark applications, the load on the manager node is small and remains constant irrespective of the application input data size. These results indicate that CellMR provides a viable framework for efficiently supporting MapReduce applications on asymmetric HPC clusters.

The rest of this paper is organized as follows. Section 2 details the motivation and background of the technologies that we use in CellMR. Section 3 discusses possible design choices, and details the one we have chosen. Section 4 gives CellMR implementation details. Section 5 presents evaluation of CellMR. Section 6 discusses the implications of the observed results. Finally, Section 7 concludes the paper.

2. Enabling Technologies

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

2.1. Using commodity components

The use of cheap off-the-shelf components in large-scale clusters is well established. Setups from academia, e.g., Condor [18], etc., to commercial data centers, e.g., Google [19], Amazon's EC2 [20], 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) [21], [22] and NVIDIA GPGPU-based graphics engines [23], [24]. 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 [23]. 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

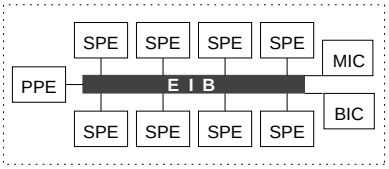

Figure 1. Cell architecture.

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 data transfers and work distribution between heterogeneous components.

2.2. IBM's Cell Broadband Engine

The Cell processor [25] provides a suitable resource that can serve as an accelerator component in CellMR. The availability of the Cell processor in the commodity Sony PS3 setup further makes it economically attractive for deployment on clusters.

The Cell [26], [27], shown in Figure 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 200 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. Based on our previous work [11], where we leveraged this facility to improve I/O performance, we believe that the explicit data management can be exploited and extended by the system manager to provide individual PS3's with necessary data directly in their memories. We use this approach in CellMR.

2.3. MapReduce

MapReduce is an emergent programming model for large-scale data processing on clusters and multi-core processors [14]. The model is simple and intuitive, comprising of two basic primitives, a *map* operation which processes key/value pairs to produce intermediate key/value results, and a *reduce* operation which collects the results in groups that have the same key. MapReduce is highly suitable for massive data searching and processing operations. The model has shown excellent I/O characteristics for traditional clusters, as evident by its successful application in large-scale search applications by Google [14]. Current trends show that the model is considered as a high-productivity alternative to traditional parallel programming paradigms for a variety of applications, ranging from enterprise computing [28], [20] to peta-scale scientific computing [29], [15], [30]. Interestingly enough, MapReduce is chosen as a programming front-end for Intel's Exo-skeleton architecture [31], a heterogeneous multi-core architecture combining Intel CoreDuo cores with graphics accelerators.

Although attractive, MapReduce typically assumes homogeneous components such that any work item can be scheduled for any of the available components. Recent work [16] on Amazon's EC2 [20] addresses performance heterogeneity, but is limited only to issues arising from using virtual machines to support nodes. Inherent architecture heterogeneity remains a problem when the cluster components include specialized accelerators, as the mapping function needs to be extended to factor in differences in the individual component capabilities and limitations. Furthermore, MapReduce in recently developed large-scale systems such as Hadoop [28] assumes that necessary data is available on local disks of processing components. Given limited I/O capabilities of accelerators, this assumption may not hold, thus posing the challenge of providing components with the necessary data in a distributed setting. In this work, we address the challenge of balancing data supply and demand between heterogeneous components.

3. Design

In this section we present the detailed design of our framework, CellMR, which we have developed to efficiently support MapReduce [14] programming model on an asymmetric accelerator-based cluster.

3.1. System Architecture Overview

A typical MapReduce setup consists of a dedicated front-end machine that handles job scheduling and resource management for a number of back-end resources. Since application programmers expect such a setup, we preserve it in CellMR. Similarly as in typical symmetric clusters,

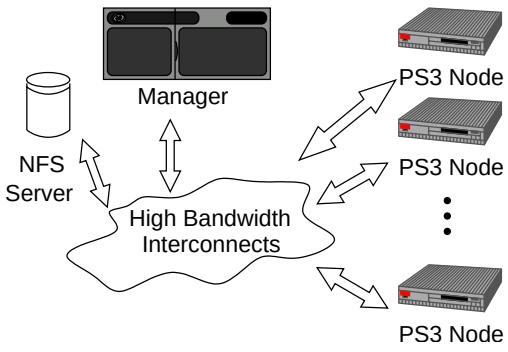

Figure 2. CellMR system architecture.

the front-end node is a general purpose multi-core server with a large amount of DRAM, which acts as a cluster *manager*. The difference is that, in CellMR, the back-end compute nodes are asymmetric cell-based accelerators, PS3s, instead of generic computers. The manager distributes and schedules the workload to the compute nodes. The generic core on the compute nodes then uses MapReduce to map its assigned workload to the accelerator cores. In essence, the programming model of CellMR resembles a two-level MapReduce: the front-end maps workloads to the cell-based back-ends, and the back-end generic core maps the workload to its accelerators.

A high-level view of the CellMR architecture is illustrated by Figure 2. The manager and all the compute nodes are connected via a high-speed network, e.g., Gigabit Ethernet. Application data is hosted on a distributed file system, such as the Network File System (NFS) [32] (used in our implementation) or Lustre [33]. The file service can be hosted either on a dedicated server, or in less demanding setups, on the manager. 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, CellMR is expected to yield a high performance to cost ratio.

The novelty of CellMR lies in its adoption of a streaming approach to supporting MapReduce. Allocating huge workloads to compute nodes in a single map operation, as is the case in standard MapReduce setups, would choke the limited non-computation resources on the asymmetric compute nodes and negate any performance benefits. Instead, CellMR slices the input into small work units and streams them to the compute nodes, which can then be processed efficiently. CellMR employs a number of techniques, such as double-buffering to avoid stalls due to I/O operations and asynchronous data mapping and collection, with the aim to derive peak performance from all system components.

3.2. Design Alternatives

A crucial task of CellMR is to efficiently manage chunks of large application data *mapped* 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 CellMR.

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. This is exacerbated by the fact that the available MapReduce runtime implementation [15] for the Cell processor itself requires a lot of memory to store internal data structures. Hence, single division of input data is not a viable approach for CellMR.

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

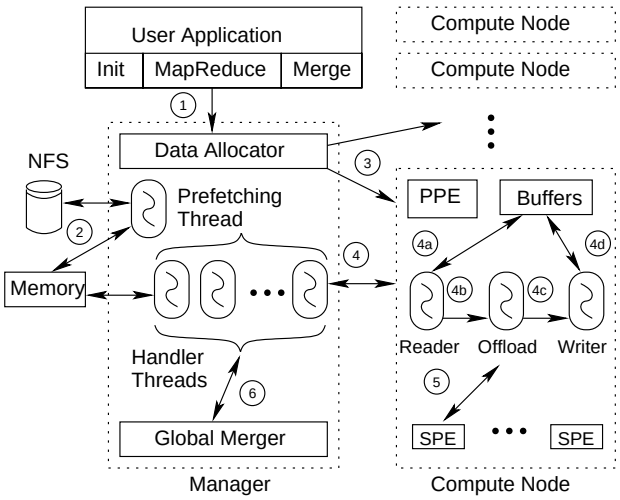

Figure 3. Interactions between CellMR components.

compute node performance can be achieved. We adopt this data streaming approach in CellMR.

3.3. CellMR Operations

In this section, we describe the runtime interactions between CellMR components at the manager and each of the compute nodes. These interactions are depicted in Figure 3.

3.3.1. Manager Operation. The manager is responsible for a number of tasks such as job queuing, scheduling, data hosting, and managing compute nodes. We assume that well-established standard techniques can be used for such manager tasks, and focus on compute-node management role in this discussion. Once an application begins execution (Step 1 in Figure 3), the manager loads a portion of the associated input data from the NFS into its memory (Step 2). This is done to ensure that sufficient data is readily available for compute nodes, and to avoid any I/O bottleneck that can hinder performance. Next, client tasks are started on the available compute nodes (Step 3). 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 (Step 4). Once the manager receives the results, it merges them (Step 6) to produce the final result set for the application. Note that the merge operation on the manager is overlapped with MapReduce operations on the compute nodes. When all the in-memory loaded data has been processed by the clients, the manager loads another portion of the input data into memory (Step 2), and the whole process continues until the entire input has been consumed. This model is similar to using a large number of small map operations in standard MapReduce.

The model described so far can suffer from two potential I/O bottlenecks: the manager can stall while reading data from the disk (NFS), and the compute nodes can stall while data is being copied from the manager to their memories. At both levels, we employ double buffering to avoid delays. The manager uses an asynchronous prefetch thread to pre-load data into a second buffer, while the data in the first buffer is being processed by the compute nodes, and vice versa. Similarly, the compute nodes also use double buffering while copying data from the manager to their memories.

It is critical to handle all communication with the compute nodes asynchronously, otherwise data distribution and collection operations will become sequential, thus reducing the performance to effectively that of a single compute node. This is of course undesirable. Making the interactions between the manager and the compute nodes asynchronous needs careful consideration. If chunks from consecutive input data are distributed to multiple compute nodes, it would require time-consuming complex sorting and ordering to ensure proper merging of the results from individual compute nodes into a consolidated result set. Such sorting would also drastically increase the memory pressure on the manager node and reduce system performance. We address this issue by using a separate handler thread for each of the compute nodes on the manager. Each handler thread works with a consecutive fixed portion of the in-memory data to avoid costly ordering operations. The size of each portion is determined by simply dividing the size of in-memory buffer by the number of available compute nodes. Each handler thread is responsible for receiving all the results from its associated compute node, and for performing an application-specific merge operation on the received data. This design leverages multi-core or multi-processor head nodes effectively.

3.3.2. Compute Node Operation. Application tasks are invoked on the compute nodes (Step 3), and begin to execute the request, process, and reply (Steps 4a to 4d) 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 CellMR 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. One thread is responsible for requesting and receiving new data to work on from the manager (Step 4a). 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 (Step 4b), and then requests more data until all

1. The user-specified 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.

free receiving buffers have been utilized. The offload thread invokes the Offload function (Step 5) 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 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 writing thread (Step 4c), which returns the results back to the manager and also frees the buffer for reuse by the receiving thread (Step 4d). Note that the compute node supports variable size work units, and can dynamically adjust the size of buffers at runtime.

3.4. Dynamic Work Unit Scaling

Efficient utilization of compute nodes is crucial for overall system performance. A key observation in CellMR is that a compute node's performance can be increased many fold by reducing memory pressure, which is in turn tied to the work unit size. A very large work unit results in thrashing on the compute nodes, while an unnecessarily small work unit increases the workload of the manager. In either case, system performance is reduced. The challenge is finding an optimally-sized work unit, which offers the best trade-off between the compute node performance and manager load.

An optimum work unit for running an application on a particular cluster can be manually determined by hard coding different work unit sizes, executing the application, and measuring the execution time for each size. The best unit size is the one for which the application execution time is minimized. However, this is a tedious and error-prone process, and requires unnecessary "test" access to resources, which is difficult to obtain given the ever increasing need for executing "production" tasks on a cluster to maintain high serviceability.

To remedy this, CellMR provides the manager with the option to automatically determine the best work unit size for a particular application. This is done by sending compute nodes varying work unit sizes at the start of the application and recording the completion time corresponding to each work unit. A binary search technique is used to modify the work unit size to determine one that gives the highest processing rate calculated using $(work\ unit\ size)/(execution\ time)$. If the processing rate is the same for two work unit sizes, the larger one is preferred as that minimizes load on the manager. The determined work unit size is chosen as the most efficient for use with the application and employed for the rest of the application run.

All available compute nodes participate in finding the optimal work unit size. Increasing work units are sent to multiple compute nodes simultaneously, although one size is sent to at least two compute nodes to determine average performance. Once a size is determined, it can also be reported to the application user to possibly facilitate optimization for a future run.

3.5. Using CellMR

From an application programmer's point of view the extended MapReduce of CellMR is used as follows. The application is divided into three parts as shown earlier in Figure 3. (i) The code to initialize and utilize the CellMR framework. This corresponds to the time spent in a MapReduce application but outside of the actual MapReduce work (initialization, intermediate data movement, finalization). This part is unique to CellMR and does not have a corresponding operation in standard MapReduce. (ii) The code that runs on the compute node and does the actual work of the application. This is similar to a standard MapReduce application running on a small portion of the input data that has been assigned to the compute node. It includes both the map phase to distribute the workload to the accelerator cores, and the reduce phase to merge the data from them. (iii) 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, and constitute the Global Merge phase that is identical in operation to the reduce phase on each compute node. The only difference is that the Global Merge on the manager works with all the data sets and produces the final results. All these functions are application-specific and should be supplied by the programmer.

4. Implementation

We have implemented CellMR 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 CellMR.

A goal of our implementation is to maintain a constant memory footprint and keep the 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 compute node threads 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 Reader and Writer 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 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. 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 [34] in our implementation due to its robust performance and our familiarity with it.

5. Evaluation

In this section, we detail our evaluation of CellMR using the implementation of Section 4. 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 two quad-core Intel Xeon processors clocked at 3 GHz, 16 GB main memory, 650 GB hard disk, and runs Linux Fedora Core 8. The manager also runs an NFS server to provide user data to the PS3s. The PS3 is a hypervisor-controlled platform, and has 256 MB of main memory (of which only about 200 MB are available for applications), and a 60 GB hard disk. The Cell runs at 3.2 GHz. Of the 8 SPEs of the Cell, only 6 SPEs are visible to the programmer [11], [35] in the PS3. 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 standalone 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 evaluating the dynamic work unit scaling of CellMR. (3) *CellMR* configuration that also uses all nodes but employs CellMR for work unit scaling and scheduling. These configurations allow us to study various aspects of CellMR in detail.

5.2. Methodology

We conducted the experiments using the only publicly available MapReduce implementation [15] for Cell processors. We focus on how CellMR design decisions affect system performance when using the MapReduce model on our cluster.

For our evaluation, we used a number of well-known MapReduce applications. These applications are classified based on the MapReduce phase where they spend most of the execution time, i.e., as either map-dominated or partition-dominated. Map-dominated applications spend more time in distributing the data than processing the partitions, while vice versa is true for partition-dominated applications. A brief description of the benchmark applications that we have ported to CellMR is provided below. More details on these applications can be found in [15].

- *Linear Regression:* This application takes as input a large set of two dimensional points, and determines a line of best fit for them. This is a map-dominated application.
- *Word Count:* This application counts the frequency of each unique word in a given input file. The output is a list of unique words found in the input along with their corresponding occurrence counts. This is a partition-dominated application.
- *Histogram:* This application takes as input a bitmap image, and produces the frequency count of each color composition in the image. This is a partition-dominated application.
- *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 is a partition-dominated application.

For each benchmark, we measured the total execution time under our setup configurations. We also measured the time and number of iterations it takes our dynamic scaling to determine an appropriate work unit size, and how the determined value compares to a manually found one.

5.3. Results

In this section, we first examine how the benchmarks behave under our test configurations. Second, we evaluate the dynamic work unit scaling of CellMR. Third, we study

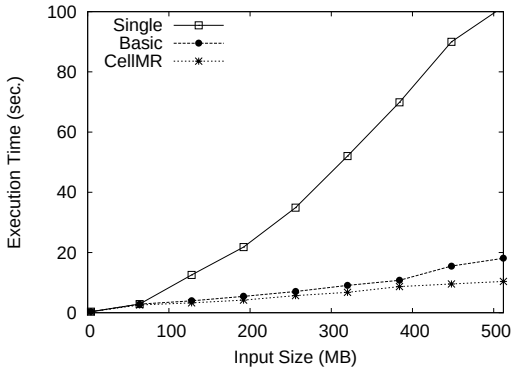

Figure 4. Linear Regression execution time with increasing input size.

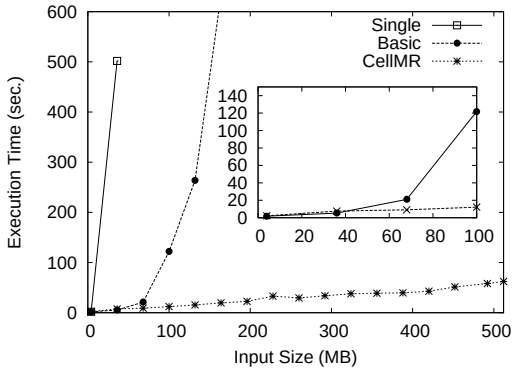

Figure 5. Word Count execution time with increasing input size.

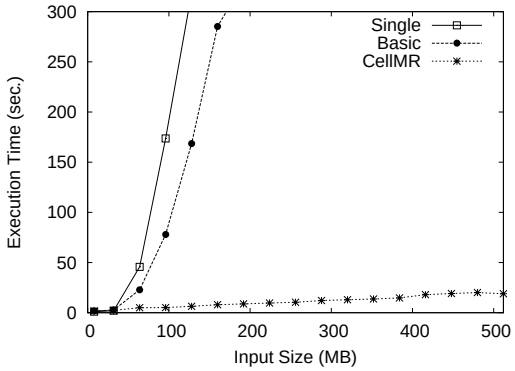

Figure 6. Histogram execution time with increasing input size.

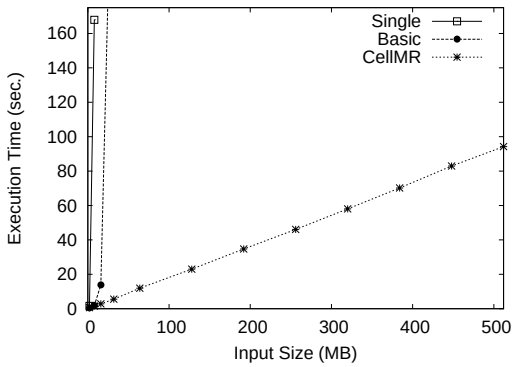

Figure 7. K-Means execution time with increasing input size.

the impact of CellMR on the manager. Finally, we determine how CellMR scales as the number of compute nodes is increased.

5.3.1. 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 4 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, CellMR performs slightly better than *Basic*, 18.9% on average for input size less than 400 MB, mostly due to its data streaming characteristics and work unit size optimizations. However, as the input size is increased beyond 400 MB, the peak virtual memory footprint for *Basic* is observed to grow over 338 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, CellMR is able to dynamically adjust the work unit size to avoid swapping on the PS3s, thus, achieving 24.3% average speedup across all the considered input sizes compared to *Basic*.

Word Count. During our experiments with Word Count, we observed exponential growth in memory consumption relative to the input data size, since each input word would emit additional intermediate data out of the map function. 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 176 MB, and took 631.9 seconds for an input size of 164 MB. Figure 5 shows the results. Here, CellMR 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 6 shows the result for running Histogram under the three test configurations. It can be observed that CellMR 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 160 MB, the peak virtual memory size

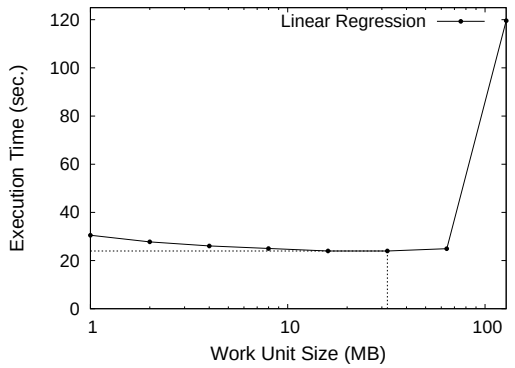

Figure 8. Effect of work unit size on execution time.

grows to 285 MB and it takes 285.3 seconds to complete. On average across all input points less than 192 MB, CellMR does 68.2% better than *Basic* for Histogram. The maximum input size that *Single* and *Basic* can handle without crashing is 192 MB, for which the execution times are 394.8 and 328.6 seconds, respectively.

K-Means. The results for the K-Means benchmark are shown in Figure 7. 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 CellMR 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 319 seconds/iteration compared to 5.5 seconds/iteration of CellMR. The only input size where the *Basic* case doesn't thrash is for 1 MB. In this instance *Basic* outperforms CellMR by 17%, as this input size is too small to amortize the management function overhead of CellMR. However, this is not of concern, as with any input size greater than 1 MB, CellMR does significantly better than *Basic*.

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

5.3.2. Work Unit Size Determination. As discussed earlier in Section 3, the work unit size affects the performance of compute nodes, and consequently the whole system. In this experiment, we first show how varying work unit sizes affect the processing time on a node. For this purpose, we use a single PS3 node connected to the manager, and run

Application	Hand-Tuned Size (Mb)	CellMR		
		Size (MB)	# Iterations	Time (s)
Linear Regression	32	30	16	0.65
Word Count	3	2	8	1.82
Histogram	2	1	4	0.15
K-Means	0.37	0.12	16	1.09

Table 1. Performance of work unit size determination.

Linear Regression with an input size of 512 MB[2]. Figure 8 shows that as the work unit size is increased, the execution time first decreases to a minimum, and eventually increases exponentially. The valley point (shown by the dashed line) indicates the size after which the compute node starts to page. Using a larger size reduces performance. Using a size smaller than this point wastes resources: notice that the curve is almost flat before the valley indicating no extra overhead for processing more data. Also, using a smaller work unit size increases the manager's load, as the manager now has to handle larger number of chunks for a given input size. We argue that using the valley point work unit size is optimal as it provides the best trade-off between the node's and the manager's performance.

Next, we evaluate CellMR's ability to dynamically determine the optimal work unit size. In principle, the optimal unit size depends on the relative computation to data transfer ratios of the applications and machine parameters. We follow an experimental process to discover optimal work unit size. We manually determined the maximum work unit size for each application that can run on a single PS3 without paging to be the optimal work unit size. We compared the manual work unit size to that determined by CellMR at runtime. For each application, Table 1 shows: the work unit size both determined manually and automatically, the number of iterations done by CellMR to determine the work unit, and the time it takes for the determination.

CellMR is able to dynamically determine an appropriate work unit that is close to the one found manually, and the determination on average across our benchmarks takes under 0.93 seconds. This is negligible, i.e., less than 0.5% of the total application execution times when input size is 2 GB. Thus, dynamic work unit scaling in CellMR is efficient as well as reasonably accurate.

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 MapReduce task 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

2. The results are the similar in other applications and input sizes.

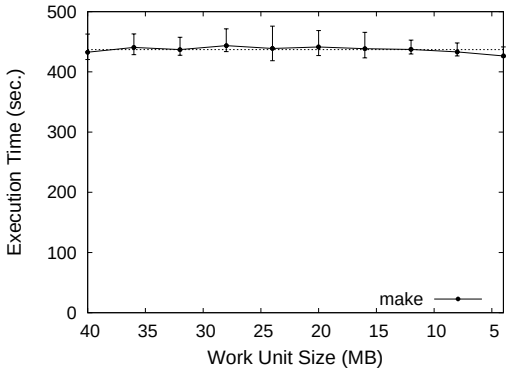

Figure 9. Impact of work unit size on the manager.

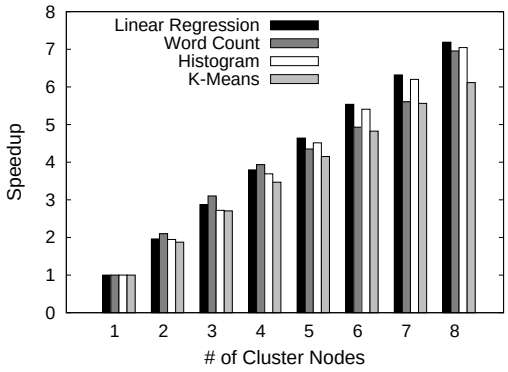

Figure 10. Effect of scaling on CellMR.

experiment 10 times, and record the minimum, maximum, and average time for the compilation as shown in Figure 9. The horizontal dashed line in the figure shows the overall average of average compile time across all work unit sizes. 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 CellMR 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 10 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. However, 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, and 128 MB for K-Means. The curve of K-Means is based on time per iteration, as explained earlier in this section. The figure shows that CellMR scales almost linearly as the number of compute nodes is increased, and this behavior is true for all the benchmarks. However, 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 introduced communication delays even with double buffering, and prevented CellMR 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 CellMR using typical MapReduce benchmark applications on an asymmetric cluster with 8 PS3 compute nodes shows that, compared to a non-streaming approach, CellMR achieves 50.5% better performance on average. Moreover, CellMR 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, CellMR provides a viable framework for efficiently supporting MapReduce applications on asymmetric HPC clusters.

6. Discussion

Our evaluation has shown that asymmetric multi-core processors coupled with general-purpose processors, can be used effectively in asymmetric distributed clusters using our highly scalable programming model, CellMR, for compute- and data-intensive applications. Furthermore, we observed a clear benefit of adopting a streaming approach to bridge the computational and I/O gaps between manager and accelerator based compute nodes by feeding them with the required data to overcome some of their significant shortcomings, i.e. small memory and limited I/O capabilities.

Adopting the streaming approach also exploits strong attributes of asymmetric multi-core processors, such as low-latency and fast internal memory. By careful tuning of the design parameters at runtime, such as optimal work unit size, synchronization and communication parameters, asymmetric multi-core processors can serve as a cost-effective compute nodes for high performance clusters. To this end, an obstacle is the saturation of network bandwidth between the cluster manager and the compute nodes due to their dependency on the manager for OS services such as I/O.

A possible remedy is the use of multiple managers to create a tiered cluster model, where each manager manages a set of compute nodes on a dedicated network. However, such a setup would require an extra merge phase to combine

the results on each manager node. Nonetheless, the global merge would be very similar to the reduce function on each manager, which combines the results from associated compute nodes. Consequently, the programming effort for this extra merge function is only a small delta in addition to the programming effort for result consolidation on the compute nodes, but with considerable benefit of mitigating a bandwidth bottleneck.

7. Conclusion

This paper advocates a streaming approach to implementing programming models for asymmetric distributed clusters, featuring accelerators at their compute nodes and conventional servers at their head nodes. We presented the design, implementation and evaluation of CellMR, a scalable implementation of MapReduce for asymmetric clusters. CellMR retains the simple, portable, and fault-tolerant programming interface of MapReduce, while exploiting multiple interconnected computational accelerators for higher performance. CellMR implements dynamic schemes for memory management and work allocation, so as to best adapt work and data distribution to the relative computation density of the application. These dynamic schemes enable higher performance and better utilization of the available memory resources, which in turn helps economizing on capacity planning for large cluster installations.

Future work on CellMR includes extension in several directions. We plan on exploring the performance of CellMR in accelerator-based systems at large scales, using multiple types of accelerators, including GPUs, as well as multiple head and I/O nodes. This paper provides initial clues on addressing capacity and resource planning issues for asymmetric clusters. We also plan on deploying CellMR for capacity planning and rightsizing of clusters given specific budgets. CellMR has been evaluated as a dedicated resource for running standalone applications, and we intend to evaluate it also in a virtualized setting, to explore performance robustness under dynamic execution conditions. Finally, one of our near-term goals is to use CellMR as a production-level programming framework for scientific data processing.

Acknowledgment

This research is supported by NSF (grants CCF-0746832, CCF-0346867, CCF-0715051, CNS-0521381, CNS-0720673, CNS-0709025, CNS-0720750), DOE (grants DE-FG02-06ER25751, DE-FG02-05ER25689), and IBM through IBM Faculty Awards (grants VTF-874574, VTF-874197). M. Mustafa Rafique is supported through a Fulbright scholarship.

References

[1] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proc. ACM ISCA.*, June 2005.

[2] M. Pericàs, A. Cristal, F. Cazorla, R. González, D. Jiménez, and M. Valero. A Flexible Heterogeneous Multi-core Architecture. In *Proc. IEEE PACT.*, 2007.

[3] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K.I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proc. IEEE ISCA.*, 2004.

[4] D. Bader and V. Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In *Proc. HiPC.*, 2007.

[5] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos. RAxML-CELL: Parallel Phylogenetic Tree Construction on the Cell Broadband Engine. In *Proc. IEEE/ACM IPDPS.*, 2007.

[6] 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 Engg., Ohio State University, 2007.

[7] B. Gedik, R. Bordawekar, and P. S. Yu. CellSort: High Performance Sorting on the Cell Processor. In *Proc. VLDB.*, 2007.

[8] S. Heman, N. Nes, M. Zukowski, and P. Boncz. Vectorized Data Processing on the Cell Broadband Engine. In *Proc. ACM DaMoN.*, 2007.

[9] 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. IEEE/ACM IPDPS.*, 2007.

[10] 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 *Proc. ACM/IEEE SC*, 2008.

[11] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos. DMA-based Prefetching for I/O-Intensive Workloads on the Cell Architecture. In *Proc. ACM CF'08.*, 2008.

[12] ClearSpeed Technology. *ClearSpeed whitepaper: CSX Processor Architecture*, 2007.

[13] Jason Cross. A Dramatic Leap Forward–GeForce 8800 GT, Oct 2007. http://www.extremetech.com/article2/0,1697,2209197,00.asp.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. USENIX OSDI.*, 2004.

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

[16] M. Zaharia, A. Konwinski, and A. D. Joseph. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. USENIX OSDI.*, 2008.

[17] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating Computing with the Cell Broadband Engine Processor. In *Proc. ACM CF'08.*, 2008.

[18] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurr. Comput.:Pract. Exper.*, 17(2-4):323–356, 2005.

[19] L. A. Barroso, J. Dean, and U. Holzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro.*, 23(2):22–28, 2003.

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

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

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

[23] GraphStream, Inc. GraphStream Scalable Computing Platform (SCP). 2006. http://www.graphstream.com.

[24] D. Göddeke, R. Strzodka, J. M. 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.

[25] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation - A performance view. *IBM J. Res. and Dev.*, 51(5):559–572, 2007.

[26] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM J. Res. and Dev.*, 49(4/5):589–604, 2005.

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

[28] Apache Software Foundation. Hadoop. http://hadoop.apache.org/core/.

[29] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proc. IEEE HPCA'07.*, 2007.

[30] Adam Pisoni. Skynet, Apr. 2008. http://skynet.rubyforge.org.

[31] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *Proc. ACM ASPLOS.*, 2008.

[32] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. Summer USENIX*, 1985.

[33] P. Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proc. Ottawa Linux Symposium*, 2003.

[34] Message Passing Interface Forum. MPI2: A Message Passing Interface Standard. *Int. J. of High Performance Computing Applications*, 12(1–2):299, 1998.

[35] J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra. The PlayStation 3 for High-Performance Scientific Computing. *Comp. in Sci. and Engg.*, 10(3):84–87, 2008.