Supporting MapReduce on Large-Scale Asymmetric Multi-Core Clusters

M. Mustafa Rafique[1], Benjamin Rose[1], Ali R. Butt[1]

[1]Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
{mustafa, bar234, butta, dsn}@cs.vt.edu

Dimitrios S. Nikolopoulos[1,2]

[2]Institute of Computer Science
FORTH
Heraklion, GREECE
dsn@ics.forth.gr

Abstract

Asymmetric multi-core processors (AMPs) with general-purpose and specialized cores packaged on the same chip, are emerging as a leading paradigm for high-end computing. A large body of existing research explores the use of standalone AMPs in computationally challenging and data-intensive applications. AMPs are rapidly deployed as high-performance accelerators on clusters. In these settings, scheduling, communication and I/O are managed by general-purpose processors (GPPs), while computation is off-loaded to AMPs. Design space exploration for the configuration and software stack of hybrid clusters of AMPs and GPPs is an open problem. In this paper, we explore this design space in an implementation of the popular MapReduce programming model. Our contributions are: An exploration of various design alternatives for hybrid asymmetric clusters of AMPs and GPPs; the adoption of a streaming approach to supporting MapReduce computations on clusters with asymmetric components; and adaptive schedulers that take into account individual component capabilities in asymmetric clusters. Throughout our design, we remove I/O bottlenecks, using double-buffering and asynchronous I/O. We present an evaluation of the design choices through experiments on a real cluster with MapReduce workloads of varying degrees of computation intensity. We find that in a cluster with resource-constrained and well-provisioned AMP accelerators, a streaming approach achieves 50.5% and 73.1% better performance compared to the non-streaming approach, respectively, and scales almost linearly with increasing number of compute nodes. We also show that our dynamic scheduling mechanisms adapt effectively the parameters of the scheduling policies between applications with different computation density.

1. Introduction

Asymmetric multi-core processors (AMPs) with a mix of general-purpose and specialized cores have become a leading paradigm for high-end computing. AMPs invest a significant portion of their transistor budget in specialized cores to achieve significant acceleration of computational kernels operating on vector data. The rest of the transistor budget accommodates a few conventional cores that run the operating system, communication and I/O stacks, and system services. The superiority of AMPs compared to homogeneous chip multi-processors (CMPs), in terms of performance, scalability, and power-efficiency, has been demonstrated extensively (7; 28; 30). Commercial AMPs such as the Cell Broadband Engine (Cell) are gaining traction among application developers (6; 9; 10; 14; 18; 29; 37; 31), and major processor vendors, including Intel (38) and AMD (4), are announcing AMP product lines.

The recent integration of AMPs in the first Petaflop-scale supercomputer (37) raised several interesting questions regarding the design, implementation and management of large-scale distributed systems that leverage AMPs and general-purpose processors (GPPs). The commoditization and low cost of AMPs and other multi-core computational accelerators such as GPUs (20), indicate a trend toward deploying such processors at scale. Furthermore, the vector processing capabilities of accelerators makes them natural candidates for massive data processing. Although these indicators are promising, designing and programming large-scale parallel systems with heterogeneous components —AMPs and GPPs— is an open challenge. While hiding architectural asymmetry and system scale from parallel programming model are desirable properties (22), they are challenging to implement in asymmetric systems, where exploiting the customization and computational density of AMPs is a first-order consideration. At the same time, provisioning GPPs and AMPs to achieve a balanced system is a non-trivial exercise.

To address these challenges and explore the related design choices, we implemented the MapReduce (13) programming environment for asymmetric clusters boasting AMPs and GPPs. MapReduce implements a simple interface for machine-independent and scale-agnostic parallel programming, and is typically deployed for large-scale distributed data processing on virtualized data centers. MapReduce provides minimal abstractions, hides architectural details, and supports transparent fault tolerance. Earlier research explored MapReduce implementations as a library on standalone AMPs and accelerators (12; 17). Our work explores MapReduce as a distributed homogeneous parallel programming framework which utilizes non-homogeneous multi-core processors "under the hood". Furthermore, while earlier research has extended the MapReduce scheduler to account for heterogeneity of compute nodes (39), such research addressed heterogeneity as an effect of virtualization and contention between MapReduce jobs, rather than as an effect of asymmetry in the computational density of individual hardware components on a single job. We find that hardware asymmetry can lead to severe performance penalties by exposing communication or I/O bottlenecks and address these problems in our implementation.

We present the design and implementation of MapReduce for asymmetric clusters, focusing on a setting with large-memory head nodes using GPPs and small-memory AMPs deployed as computational accelerators. We assume that there is significant variance in the capabilities of head nodes and accelerators, in terms of memory, computing power, and capabilities for running system services such as I/O. These assumptions have been influenced by the LANL RoadRunner (37) cluster setting, however, we believe that they are

realistic and broadly applicable. We present an implementation of MapReduce on clusters using x86 GPPs and Cell-based AMPs, although our design and implementation choices are not bound to specific GPP or AMP architectures. The key aspects that differentiate our design from earlier MapReduce implementations is a global data streaming approach and adaptive resource scheduling via dynamic scheduler parameterization. Cumulatively, these two techniques address heterogeneity in the capabilities and capacities of asymmetric components, by overlapping I/O and communication latencies. Furthermore, these techniques allow us to implement MapReduce efficiently on clusters with different levels of GPP and memory resource provisioning per accelerator, which in turn represent different cost-performance trade-offs. Our scheduling approach differs from that of an earlier data transfer and task scheduling framework for asymmetric clusters developed by IBM (11), which delegates scheduler parameter optimization to the application developer. Instead, our framework adapts transparently the parameters of data streaming and task scheduling to the application at runtime, thereby relieving developers of some significant programming effort.

Specifically, this paper makes the following contributions:

- A new design for realizing the MapReduce programming model on asymmetric clusters of AMPs and GPPs;
- An exploration of alternative design choices for data streaming and processing and their impact on overall system performance of asymmetric cluster architectures;
- A runtime technique for regulating data distribution and streaming in MapReduce, to bridge the asymmetry between GPPs and AMPs;
- An emulation of asymmetric blade cluster settings, using x86 blades and Cell nodes, to study the behaviour of different levels of provisioning of memory and GPP resources for accelerators;
- An evaluation of common MapReduce workloads in terms of scalability, adaptation to various computation densities, and resource conservation capability.

Our evaluation using representative MapReduce applications on an asymmetric cluster shows that our framework can significantly improve performance (as much as 82.3% in the Word Count benchmark) compared to a MapReduce implementation based on static data distribution and scheduling schemes.

The rest of this paper is organized as follows. Section 2 details the motivation and background of technologies that we leverage and adapt in this work. Section 3 examines possible design alternatives and details the one we chose. It also highlights some of the features and optimizations we included, and explains our emulation of Cell-based blades using x86 blades and Cell nodes. Section 4 presents implementation details of realizing MapReduce programming model on asymmetric clusters. Section 5 presents evaluation of our approach. Section 6 discusses the implications of the observed results. Finally, Section 7 concludes the paper.

2. Background & Motivation

In this section, we discuss relevant background and motivation for our work.

2.1 Availability of Commodity Components

The use of commodity off-the-shelf components in large-scale clusters is well established. Setups from academia (e.g., Condor (36)), to commercial data centers (e.g., Google (8), Amazon's EC2 (3)), routinely employ such components to meet their high-performance computing and data processing needs. AMPs and accelerators are currently being commoditized, with products such as the Cell-based Sony PlayStation 3 (PS3) (1; 26) and NVIDIA GPU-based graphics engines (15; 16) offering extremely low-cost alternatives to high-end compute nodes and enabling the use of accelerators in clusters en masse.

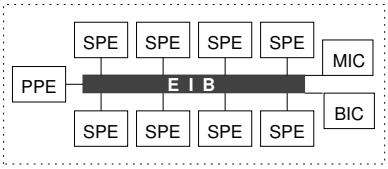

Figure 1. Cell architecture.

Unfortunately, computational accelerators require additional programming effort and pose new challenges to resource management. Programming accelerators typically implies working with multiple ISAs and multiple compilation targets. Accelerators also have much higher compute density and raw performance than conventional processors, therefore decomposing applications into accelerated and non-accelerated components is a task requiring prudence and often, experimentation. Furthermore, the coupling of accelerators with GPPs implies imbalance between the two components, which needs to be countered during scheduling and data distribution. Many accelerators are small-memory compute nodes, with limited on-board storage and limited, if any, support for general-purpose operating system.

2.2 Cell Broadband Engine

The Cell processor (35) is a leading AMP, popularized through the Sony PS3 and later deployed in IBM blades and the first Petaflop-class system (RoadRunner). We use the Cell as the accelerator component in our MapReduce framework.

The Cell (19; 21) (Figure 1), is a heterogeneous chip multiprocessor with one general-purpose PowerPC SMT core (the Power Processing Element – PPE), and eight vector-only cores (the Synergistic Processing Elements – SPEs). SPEs are specialized processors with software-managed private memories. They are designed to accelerate data-parallel (vector) computations. The PPE typically operates as a front-end for scheduling tasks and distributing data between SPEs, as well as for running the conventional operating system. It also provides support to SPEs for executing system calls and services. The on-chip interconnection network of the Cell is a circular ring, termed the Element Interconnect Bus (EIB), which connects all nine cores with memory and an external I/O channel to access other devices, such as the disk and network controller. Each SPE has a 256 KB fast local store, which is a private, software-managed memory. The application programmer is responsible for moving data between main memory and local stores, using DMAs, and for synchronizing data between local stores and main memory, as needed. The application programmer can overlap data transfer latency with computation, by issuing asynchronous DMA requests both from the SPE and the PPE side. In current installations, the PPE runs Linux with Cell-specific extensions that enable user-space libraries to load and execute code on the SPEs.

We use Sony PS3s as compute nodes in this work. A shortcoming of using the PS3 in a realistic setting 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 by adopting appropriate data streaming and staging techniques. The Cell gives application programmer the ability to explicitly manage the flow of data between the main memory and each individual SPE's local store. Based on our earlier work (31), where this facility was leveraged to improve I/O performance, we believe that explicit data management can be exploited and extended by

the system manager to provide individual PS3s with necessary data directly in their memories. We adopt this solution in this work.

2.3 MapReduce Programming Model

MapReduce is an emergent programming model for large-scale data processing on clusters and multi-core processors (12; 13; 17; 32). The model comprises two basic primitives, a *map* operation that processes key/value pairs to produce intermediate key/value results, and a *reduce* operation that collects the results in groups that have the same key. MapReduce is ideal for massive data searching and processing operations. It has shown excellent I/O characteristics on traditional clusters, and has been successfully applied in large-scale data search by Google (13). Current trends show that the model is considered a high-productivity alternative to traditional parallel programming models for a variety of applications, ranging from enterprise computing (3; 5) to peta-scale scientific computing (2; 12; 32). MapReduce has also been chosen as a programming front-end for Intel's Pangea architecture and Exoskeleton software environment (24; 38). Pangea is an AMP integrating Intel CoreDuo cores with graphics accelerators.

MapReduce typically assumes homogeneous components where any work item of map and reduce tasks can be scheduled to any of the available components. Recent work (39) on Amazon's EC2 (3) addresses performance heterogeneity arising from virtualization and contention between jobs for shared resources. Our work addresses architecture heterogeneity instead, which is a limitation when cluster components include specialized accelerators. In this case, the user should consider individual component capabilities to optimally schedule map and reduce tasks. Furthermore, publicly available implementations of MapReduce, such as Hadoop (5), assume that data is available in the local disks of components. Given the limited I/O capabilities of accelerators, this assumption may not hold, thus creating the problem of providing accelerators with the necessary data in a distributed setting. We address this problem with data streaming and dynamic scheduling schemes.

3. Design

In this section, we present the design for supporting MapReduce on an asymmetric cluster of AMPs and GPPs. We discuss the alternatives and parameters that we have considered in our design.

3.1 Architecture Overview

Similarly as in typical MapReduce environments, our setup consists of a dedicated front-end machine that acts as a cluster *manager* for a number of back-end resources. The manager is a general-purpose server with multi-core x86 processors and a large amount of memory. Our setup differs from a conventional homogeneous MapReduce setup in that we employ Cell-based compute nodes as back-end resources. We specifically consider two classes of compute nodes: First, compute nodes with limited (small) memory and I/O capabilities. A Sony PS3 is an example of such a node, where the accelerators do not (and can not) execute the entire system software stack because of limited local memory. Second, compute nodes that are well-provisioned with large amounts of memory and full operating system and I/O capabilities. A Cell-based blade such as the IBM QS20 series (27) would represent such a well-provisioned compute node. We emulate this setup by coupling a dedicated large-memory x86 "driver" node with each PS3.

Note that our design does not require homogeneous compute nodes in a cluster, however we expect this to be the common case. In any case, the manager distributes and schedules the workload to available compute nodes. For small-memory compute nodes, the PPE core on the Cell uses MapReduce to map its assigned workload to the SPEs. For well-provisioned compute nodes, the dedicated driver first collects data and then streams the workload to the attached PS3, which in turn uses MapReduce internally to utilize the SPEs. In essence, our setup implements a multi-level MapReduce: the front-end maps workloads to the accelerator-based back-ends, while the back-end GPPs map the workload to accelerators.

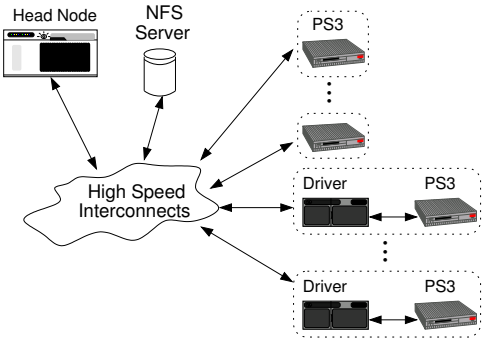

Figure 2. High-level system architecture.

Figure 2 illustrates a high-level view of the system architecture. 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) (33) or Lustre (34). In our implementation, we used NFS for baseline comparative investigation of our alternate approaches. Designing and supporting advanced customized file systems for AMPs is an orthogonal problem, and remains a focus of our future work. Note that our design allows the compute node to be either a small-memory node or a well-provisioned node. The manager is responsible for scheduling jobs, distributing data and allocating work between compute nodes, as well as for providing other support services at the front-end of the cluster. The actual data processing load is carried by the Cell-based accelerators.

For small-memory nodes, we use PS3s as back-end accelerators. For well-provisioned nodes, we envision using Cell-based blade servers, e.g. IBM's QS20, QS21, and QS22 (27). These Cell-based servers each have two Cell processors operating at 3.2 GHz and 1 GB memory per processor. QS22 uses the recent version of Cell, PowerXCell 8i, which adds support for up to 32 GB of slotted DDR2 memory, and provides improved double-precision floating-point performance on the SPEs as compared to its predecessors. For this work, we emulate a well-provisioned Cell-based blade by using a back-end node that is actually a dedicated general-purpose driver connected to a PS3, as seen in Figure 2. This approach provides large physical memory and I/O capabilities for the accelerator, which is typical of Cell-based blades, but at a fraction of the cost. The downside is that the bandwidth between the driver and the PS3 is significantly lower than the memory bandwidth on an actual Cell blade. We discuss this emulation in more detail in the next Section.

The key aspect of our design is the adoption of a streaming approach to supporting MapReduce. Statically decomposing workloads among compute nodes in a single map operation, as is the case in standard MapReduce setups, will oversubscribe the DRAM of small-memory compute nodes for realistic workloads. A streaming approach can be used instead, to split up data in work units that fit in-core on the compute nodes. To avoid stalls due to I/O operations and communication latency and sustain high computation performance across all system components, we employ various optimization techniques, such as prefetching, double buffering and asynchronous I/O.

3.2 Design Alternatives

Efficient allocation of application data to compute nodes is a central component in our design. 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 chunks to the local disks of the compute nodes. The application on the compute nodes can then get the data from the local disk as needed, and 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 lightweight for the manager node as it reduces the allocation task to a single data distribution.

Static decomposition and distribution of data among local disks can potentially be employed for well-provisioned compute nodes. However, for nodes with small memory, there are several drawbacks: (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 slower than those available to the manager; (ii) it requires compute nodes to read required data from disks, which have greater latency as compared to other alternatives, such as main memory; (iii) it entails modifying 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; (iv) it entails extra communication between the manager and the compute nodes, which can slow the nodes and affect overall performance. Hence, this is not a suitable choice for use with small-memory accelerators.

A second 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, for small-memory nodes, this approach can create chunks that are very large compared to the physical memory available at the nodes, thus leading to memory thrashing and reduced performance. This is exacerbated by the fact that available MapReduce runtime implementations (12) require additional memory reserved for the runtime system to store internal data structures. Hence, static division of input data is not a viable approach for our target environment.

The third alternative is to divide the input data into chunks, with sizes based on the memory capacity of the compute nodes. Chunks should still be mapped to virtual memory to avoid unnecessary copying, whereas the chunk sizes should be set so that at any point in time, a compute node can process one chunk while streaming in the next chunk to be processed and streaming out the previously computed chunk. This approach can improve performance on compute nodes, at the cost of increasing the manager's load, as well as the load of the compute node cores that run the operating system and I/O protocol stacks. Therefore, we seek a design point which balances the manager's load, I/O and system overhead on compute nodes, and raw computational performance on compute nodes. We adopt this approach in our design.

3.3 Emulating Cell-Based Blade Servers

Due to lack of availability of Cell-based blades, e.g.,IBM QS, in our hardware setting to build a tightly-coupled blade cluster, we emulate, somewhat crudely, blade servers through a tight coupling of dedicated large-memory drivers and PS3's. The "mimic" blades use a PS3 node directly attached to a dedicated driver node over a Gigabit Ethernet one-to-one connection (Figure 2). Clearly, the Gigabit connection falls short of the actual bus bandwidth between memory and Cell processors on real IBM QS blades. We compensate for this discrepancy by tuning the sizes of work units streamed from the driver to the PS3 and vice versa, so that the high latency of the Ethernet interconnect is not exposed. Overall, the driver is a server-class resource that provides large memory and fast I/O capabilities to make up for the limitations of the PS3. The manager distributes input data to the drivers in chunks that fit in driver memories. These chunks are further split into smaller chunks and streamed in and out of PS3 nodes.

In our MapReduce implementation, the driver manages its attached PS3 similar to how the head node manages drivers. Once the entire chunk provided by the head node is processed and merged by the driver, the results are sent to the head node, which then proceeds to perform a global merge of driver results. In all cases, the architecture of the back-end compute nodes is transparent to the application programmer, with the only observable difference of performance[1].

3.4 Data Management Operations

In this section, we describe the runtime interactions between the various software components at the manager and each of the compute nodes, as depicted in Figure 3. The manager is a component that we implemented from scratch for our hardware setting and subsequently integrated with our MapReduce framework.

3.4.1 Manager Operation

The manager handles job scheduling, data hosting, and data distribution among drivers or compute nodes. We use well-established standard techniques for the first two tasks and focus on compute-node management and data distribution 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 file system (NFS in our current implementation) 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. For well-provisioned compute-nodes with drivers, this step is replaced by direct prefetching on the drivers.

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). For well-provisioned nodes, the result data is directly written to the file system, and the manager is informed of task completion only. Once the manager receives the results, it merges them (Step 6) to produce the final result set for the application. 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 design described so far may suffer from two I/O bottlenecks: the manager can stall while reading data, or the compute nodes can stall while data is being transferred to them either from the manager or from the file system. At both levels, we employ double buffering to avoid delays. An asynchronous prefetch thread is used to pre-load data from the disk into a buffer, while the data in an already-loaded buffer is being processed, and the previously computed buffer is written back. Similarly, the driver, if used, and the compute nodes also use double buffering to overlap data-transfer with computation.

It is critical to handle all communication with the compute nodes asynchronously, otherwise data distribution and collection will become sequential and reduce the performance to effectively

[1] The compute node kernels that run on accelerators should still be modified to exploit any custom characteristics of the accelerators, such as vector units or hardware threads. We believe that this is a task that is orthogonal to our work. Vendor-specific toolchains and libraries can assist the application programmer in customization.

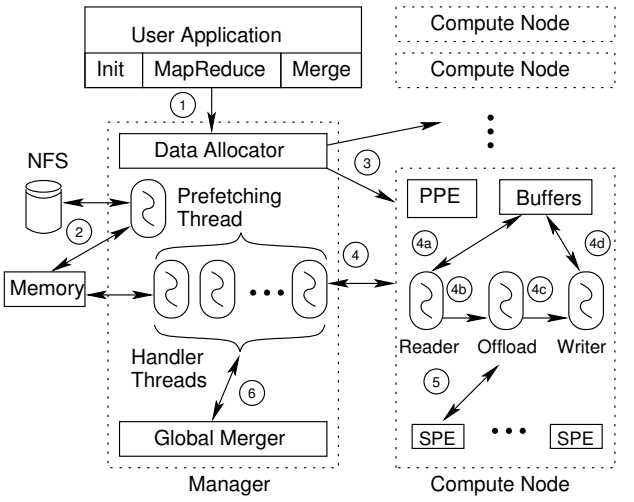

Figure 3. Interactions between framework components.

that of a single compute node. Making the manager's interactions with the compute nodes asynchronous needs careful consideration. If chunks from consecutive input data are distributed to multiple compute nodes, MapReduce would require complex sorting and ordering operations to ensure proper merging of the results from individual compute nodes into a consolidated result set. Such sorting would incur high overhead and increase memory pressure on the manager node, thus reducing system performance significantly. We address this issue by using a separate handler thread on the manager for each of the available compute nodes. Each handler works with a consecutive fixed portion of the data to avoid costly ordering operations. 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. Among other advantages, this design also leverages multi-core or multiprocessor head nodes effectively.

3.4.2 Compute Node Operation

Application tasks are invoked on the compute nodes (Step 3), and begin to execute a request, process, and reply (Steps 4a to 4d) loop. 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*[2] to perform computations on the incoming data, our framework on the compute nodes provides all other functionality, including communication with the manager (or driver) 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 or disk. One thread (Reader) is responsible for requesting and receiving new data from the manager (Step 4a). The data is placed in a receiving buffer. When data has been received, the receiving buffer is handed over to an Offload thread (Step 4b), and the Reader thread then requests more data until all available 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 these parameters are read-write parameters. This is to give the Offload function abilities to resize the buffer, change the data type, and change the data size depending on the application. When the Offload function completes, the recent output buffer is handed over to a Writer thread (Step 4c), which returns the results back to the manager and releases the buffer for reuse by the Reader thread (Step 4d). Note that the compute node supports variable size work units, and can dynamically adjust the size of buffers at runtime.

As pointed out earlier, the driver in our emulated Cell blade server interacts with the accelerator node similarly as the manager interacts with the compute nodes. The difference between the manager and the driver node is that the manager may have to interact and stream data to multiple compute nodes, while the driver only manages a single accelerator node. The driver further splits the input data received from the manager and passes it to the compute node in optimal size chunks as discussed in the following section.

3.5 Dynamic Work Unit Scaling

Balancing compute node utilization with manager load and driver load requires optimizing the work unit size used for data streaming between the manager, resource-constrained and well-provisioned compute nodes, and drivers nodes. An optimal work unit size for an application on a particular cluster can be manually determined through exhaustive searching. Application programmer can hard-code different work unit sizes, execute the application, and measure execution time for each size. This is a tedious process and does not take into account the dynamic behavior of the asymmetric cluster. Furthermore, application performance typically depends on more than one tunable parameter, therefore the parameter search space for the application programmer can grow rapidly beyond manageable proportions.

To remedy this, we provide the manager and the driver node with the option to automatically determine the best work unit size for a particular application. This is done by sending accelerator 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 it minimizes load on the manager or drivers and avoids extra communications between manager, driver and compute nodes. The determined work unit size is chosen as the existentially most efficient for use with the particular application and employed for the rest of the application run.

All available compute nodes participate in finding the optimal work unit size. Optimal work unit size is determined by sending work units of increasing size to multiple compute nodes simultaneously, although one size is sent to at least two compute nodes to determine average performance for a particular work unit size. Once optimal work unit size is determined, it can also be reported to the application user to serve as a suitable starting point for optimizing future runs.

3.6 Using Asymmetric MapReduce

From an application programmer's point of view the asymmetric MapReduce is used as follows. The application is divided into three parts as shown in Figure 3. (i) The code to initialize and use the framework. This corresponds to the time spent in a MapReduce application but outside of the actual MapReduce work (initialization, intermediate data distribution and movement, and finalization). This part is unique to our design and does not have a corresponding operation in standard MapReduce programming model. (ii) The code that runs on the compute nodes and performs actual

[2] 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 GPP PowerPC core on the Cell to produce the output data that is returned to the manager.

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 at the manager/driver nodes every time a chunk is processed at the compute node and result is received by the manager/driver nodes. It also constitutes 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 data sets received from compute nodes or drivers, and produces the final results. These functions are application-specific and should be supplied by the application programmer.

4. Implementation

We have implemented our framework in lightweight libraries for each of the target platforms, i.e., x86 on the manager and drivers, and PowerPC on the compute nodes, with about 1250 lines of C code. The libraries provide the application programmers with necessary constructs for using the framework.

For well-provisioned accelerator nodes, we employed a large number of buffers and direct I/O using the distributed file system (NFS) to conceal any delays due to data transfers between accelerator nodes and storage devices. For small-memory accelerators on the other hand, we aimed to maintain a constant memory footprint and keep the memory pressure in check with large input data. Therefore, we decided to use only two buffers, one for the Reader and Writer thread and one for the Offload thread. These buffers give the compute node more memory to use for computation but still allow overlapping of communication with computation. Note that we also allow the application programmers to modify the number of buffers for sending/receiving data between manager, driver and compute nodes, if required by the application.

Another decision is to determine how to transfer the data between manager, driver and compute nodes. For well-provisioned nodes with drivers, the driver can provide a compute node with parameters such as input file location, starting offset, and size of the chunk to process. The compute node can then use these parameters and read the required data into its memory. However, for small-memory nodes this results in a large number of very small requests and can create contention at the NFS server, resulting in increased I/O times for all compute nodes. We addressed this by implementing a prefetching scheme at the manager node to read the input data in its memory, which avoids the said bottleneck at NFS server. The prefetched data is then distributed to compute nodes using any standard communication mechanism. We used MPI (25) in our implementation for communication and synchronization between manager, driver and compute nodes, due to its proven performance and our familiarity with it.

5. Evaluation

In this section, we present our evaluation of the MapReduce framework outlined in Section 3 and 4. We describe our experimental testbed, the benchmarks that we have used to test our framework, and present the results.

5.1 Experimental Setup

Our base hardware testbed includes eight Sony PS3 compute nodes connected via 1 Gbps Ethernet to a manager node. The manager has two quad-core Intel Xeon 3 GHz processors, 16 GB main memory, 650 GB hard disk, and runs Linux Fedora Core 8. The manager also runs an NFS server. The PS3 is a hypervisor-controlled platform, with 256 MB of main memory (of which 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 (23; 31) on the PS3. Each PS3 node has a swap space of 512 MB and runs Linux Fedora Core 7. To emulate Cell-based blades, we connected each PS3 directly with a driver node, the configuration of which is identical to that of the manager, with the exception that the driver has only 8 GB of main memory. Each driver is connected with the manager via a 1 Gbps Ethernet switch.

For the experiments, we used four different resource configurations. (1) *Single* configuration, which runs the benchmarks on a stand alone PS3, with data provided from an NFS server to factor out any effects of the PS3's slow local disk. *Single* provides a measure of performance of one small-memory, high-performance computational accelerator running the benchmarks. (2) *Basic* configuration uses the manager and compute nodes as follows. The manager equally divides the input at the beginning of the job and assigns it to the compute nodes in one step. The manager then waits for the data to be processed, before merging individual output to produce the final results. *Basic* serves as the baseline for evaluating streaming and dynamic work unit scaling in our framework. (3) *Accelerator-Based* or *AcB* configuration also uses the manager and the compute nodes, but employs our framework for work unit scaling, data streaming, and scheduling. (4) *Blade* configuration, which uses our emulated Cell blades (driver-PS3 couples) and our framework for work unit scaling, streaming, and scheduling.

5.2 Methodology

We conducted the experiments using the only publicly available MapReduce implementation (12) for Cell processors, bearing in mind that this implementation is still amenable to several Cell-specific optimizations (12). The evaluation focuses on how our design decisions affect performance when using MapReduce on an asymmetric cluster with AMPs and GPPs.

For our evaluation, we used four common MapReduce applications, distributed with the Cell MapReduce runtime environment. These applications are classified based on the MapReduce phase where they spend most of the execution time. A brief description of the applications that we have ported to our framework is provided below. More details on these applications can be found in (12).

- *Linear Regression:* This application takes as input a large set of 2-Dimensional points, and determines a linear best fit for the given points. 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 RGB 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 predefined number of clusters with approximately 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 required to determine appropriate work unit size using our dynamic work unit scaling mechanism, and compare it with the manually determined value.

5.3 Results

In this section, we first examine how the applications behave under our experimental configurations. Second, we evaluate dynamic

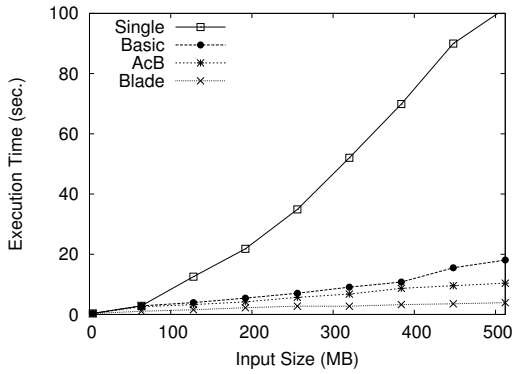

Figure 4. Linear Regression execution time with increasing input size.

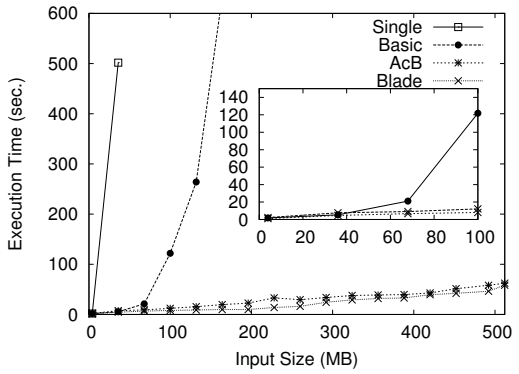

Figure 5. Word Count execution time with increasing input size.

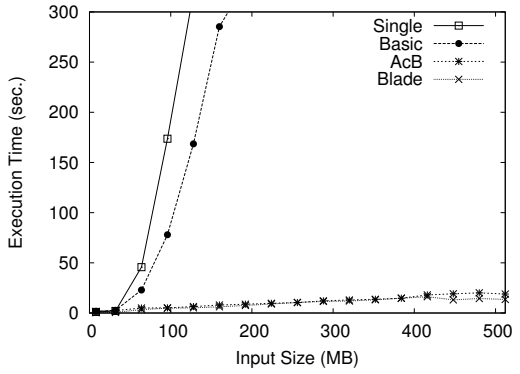

Figure 6. Histogram execution time with increasing input size.

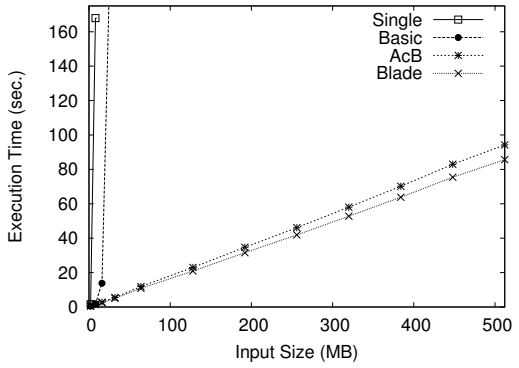

Figure 7. K-Means execution time with increasing input size.

work unit scaling. Third, we study the impact of our design on manager load. Finally, we determine how the design 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 (4 MB) to 2^{29} points (512 MB). Figure 4 shows the results. Under *Single*, the input data quickly starts producing more intermediate data than the available physical memory, resulting in increased swapping, and consequently increasing the execution time. In *Basic*, a smaller fraction of the data is sent to each of the compute nodes in each iteration, which alleviates the memory pressure from these nodes. Initially, *AcB* performs slightly better than *Basic*, 18.9% on average for input sizes less than 400 MB. This is attributed primarily to data streaming in *AcB* and work unit size optimizations. However, as the input size is increased beyond 400 MB, the peak virtual memory footprint for *Basic* on the PS3s is observed to grow over 338 MB, much greater than the 200 MB of available memory, leading to increased swapping. Once *Basic* starts to swap, its execution time increases noticeably. Our framework is able to dynamically adjust the work unit size to avoid swapping on the compute nodes. *AcB* and *Blade* achieve 24.3% and 60.3% average speedup over *Basic*, across all input sizes of Linear Regression.

Blade performs 51.0% better than *AcB*, since each driver node in *Blade* makes use of its large memory to store intermediate results, and performs merge operations on the results produced by the attached PS3 node. In contrast, under *AcB*, the manager has to perform the merge operation on the results received from each of the compute nodes while handling data distribution and other manager operations.

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 abort execution[3]. Similarly, *Basic* was 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. *AcB* and *Blade* are not only able to process any input size, they outperform *Basic* by 20.2% and 54.4% on average, respectively, for the inputs where *Basic* completes without thrashing (input data size < 96 MB, emphasized in the inset in the figure). Once again, *Blade* outperforms *AcB* (by 32.0%), as most of the scheduling and merging tasks are delegated to the drivers.

Histogram Figure 6 shows the result for running Histogram under the four test configurations. It can be observed that *AcB* and *Blade* scale linearly with the input data size. *Basic* initially scales linearly, 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 grows to 285 MB and it takes 285.3 seconds to complete. On average across all input sizes that do not cause thrashing, *AcB* and *Blade* perform 68.2% and 90.0% better than *Basic*,

[3] Execution was aborted by the operating system due to memory shortage and not because of an application error.

Application	Hand-Tuned Size (MB)	Our Framework		
		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.

respectively . The maximum input size that *Single* and *Basic* can handle without thrashing is 192 MB, for which the execution times are 394.8 and 328.6 seconds, respectively. Between *AcB* and *Blade*, *Blade* performs 21.6% better than *AcB* across all input points, since *Blade* can make use of memory available at the driver node to store the intermediate results and perform the merge operations.

K-Means The results for the K-Means benchmark are shown in Figure 7. Note that K-Means uses 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 *AcB* and *Blade* configurations scale linearly, *Single* and *Basic* use up all the available virtual memory with relatively low input sizes. Both *Single* and *Basic* abort 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 *AcB*. The only input size where *Basic* does not thrash is 1 MB, where it outperforms *AcB* and *Blade* by about 17%, as this input size is too small to amortize the management overhead of our approaches. However, this is not of concern, as with any input size greater than 1 MB, both *AcB* and *Blade* do significantly better than *Basic*. Overall, *Blade* performs 9.0% better than *AcB* across all input sizes.

In summary, memory limitations and thrashing notwithstanding, our framework outperforms static data distribution due to better overlap of computation with communication and I/O latency. Our framework 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

In this section, 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 Linear Regression benchmark with an input size of 512 MB [4]. Figure 8 shows the result of this experiment. As the work unit size is increased, the execution time first decreases to a minimum, and eventually increases exponentially. The valley point (shown by a dashed line) indicates the size after which the compute node starts to page. Using a larger size reduces performance since it produces more intermediate data and requires larger buffers to implement double-buffering at compute nodes, which have limited memory available for applications. 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 the same input size. Using the valley point work unit size is optimal as it provides the best trade-off between compute node's and manager's performance, and results in minimal execution time.

Next, we evaluate our framework'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 application and machine parameters, most notably, latencies and bandwidths of the chip, node and network interconnects. 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 *AcB* at runtime. For each application, Table 1 shows: the work unit size determined both manually and automatically, the number of iterations done by *AcB* to determine the best work unit, and the time it takes for reaching this decision.

Our framework 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. Note that optimal work unit size determination is independent of the given input size, and has a constant cost for a given application. Thus, dynamic work unit scaling in our framework is efficient as well as reasonably accurate.

5.3.3 Impact on the Manager

In this experiment, we determine the effect of varying work unit sizes on manager performance. This is done as follows. First, we start a long running job, using Linear Regression benchmark, 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 load on the manager. For each work unit size, we repeat the experiment 10 times, using the *AcB* configuration, 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 compile time across all work unit sizes. Given that the overall average remains within the minimum and maximum times, we can infer that the variation in the compile time curve is within the margin of error. Thus, the relatively flat curve indicates that our framework has a constant load on the manager and can support various workloads without the manager becoming a bottleneck.

5.3.4 Scaling Characteristics

In the next experiment, we evaluate how performance of different benchmark applications scales with the number of compute nodes, using the *AcB* configuration. Figure 10 shows the speedup in performance normalized to the execution time when *AcB* uses 1 compute node. We use the same input size for all runs of any given application. However, the input sizes for different applications are chosen to be large enough so that each application benefits from using 8 nodes: 512 MB for Linear Regression and Histogram benchmarks, 200 MB for Word Count benchmark, and 128 MB for K-Means benchmark. The curve of K-Means is based on time per iteration, as explained earlier in this section. Figure 10 shows that our framework scales almost linearly as the number of compute nodes is increased and this behavior is sustained in all benchmarks with up to seven compute nodes. However, we observe some curving of scalability when the eighth compute node is added. Upon further investigation, we found that network bandwidth utilization with eight compute nodes was quite high, as much as 107 MB/s, compared to the maximum observed value of 111 MB/s on our network, which was measured using remote copy of a large file. This introduces communication delays that are not entirely masked with double buffering, and prevent our framework from achieving a linear speedup. Nevertheless, if the ratio of time spent in computation compared to that in communication is high, as is the case in many scientific applications, near perfect speedup can be obtained. We tested this hypothesis by artificially increasing our compute time for Linear Regression benchmark by a factor of 10, which resulted in a speedup of 7.8 on 8 nodes cluster.

[4] The results are similar for other applications and input sizes.

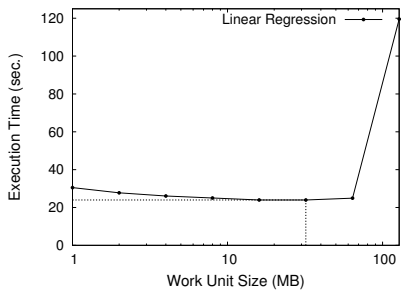

Figure 8. Effect of work unit size on execution time.

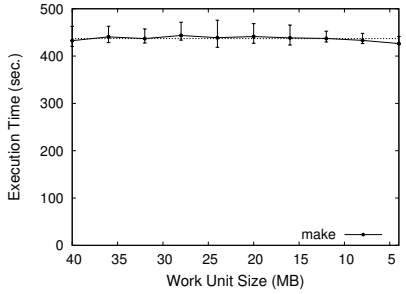

Figure 9. Impact of work unit size on the manager.

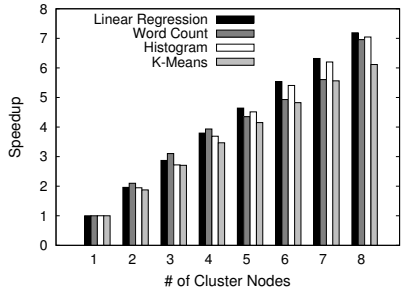

Figure 10. Effect of scaling on our framework.

5.3.5 Summary

Our evaluation shows that with standard MapReduce applications on an asymmetric cluster with 8 PS3 compute nodes, our MapReduce framework achieves 50.5% and 73.1% higher performance, compared to a basic version of the framework which does not exploit streaming, for small-memory and well-provisioned compute nodes, respectively. It utilizes limited memory available at compute nodes efficiently by adapting effectively to the relative computation to data transfer density of applications by converging to nearly optimal work unit sizes. Moreover, it has minimal impact on manager's load, and scales well with increasing number of compute nodes (a speedup of 6.9 on average on an 8 node cluster). Thus, our framework provides a viable solution for efficiently supporting MapReduce on asymmetric clusters.

6. Discussion

Our evaluation has shown that asymmetric multi-core processors coupled with general-purpose and specialized cores, can be used effectively in tightly-coupled asymmetric distributed clusters to support highly scalable programming models, such as MapReduce, for computationally challenging and data-intensive applications. Furthermore, we observed a clear benefit of adopting a streaming approach to bridge the computational and I/O gaps between heterogeneous components, such as AMPs and GPPs, and feed in the resource-constrained accelerators 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 the inherent low-latency and fast memory interconnects, which are strong attributes of AMPs. By careful tuning of the design parameters at runtime, such as optimal work unit size, synchronization and communication parameters, resource-constrained AMPs can serve as a cost-effective components for high performance clusters. To this end, an obstacle is the saturation of network bandwidth between the cluster manager and the small-memory AMPs due to their dependency on the manager for OS services such as I/O.

This can be remedied by using well-provisioned AMPs, which significantly alleviate the memory, computation and I/O pressure from the cluster manager as compared to small-memory AMPs. GPPs on well-provisioned AMPs can perform major scheduling and work distribution tasks, relieve the cluster manager, and thus enable higher system scalability. Moreover, the GPPs make efficient use of their large memories by prefetching the data required by AMPs directly from the network and/or distributed storage without incurring extra overhead at the manager. Finally, high-speed internal/external communication links between GPPs and AMPs can be used to efficiently offload compute kernels of scientific applications and large data to attached AMPs.

7. Conclusion

We presented the design and implementation of a MapReduce programming environment for asymmetric clusters of AMPs and GPPs. Our framework relies on a streaming approach and dynamic scheduling and data distribution techniques to implement an architecture-agnostic, yet scalable programming model for asymmetric distributed clusters, featuring accelerators at their compute nodes. We have been able to preserve the simple, portable, and fault-tolerant programming interface of MapReduce, while exploiting multiple interconnected computational accelerators for higher performance. We presented 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 and to the variability of computational and storage capacities across asymmetric components. Our 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.

Our future work involves extensions of our framework in several directions. We plan on exploring the performance of our design in accelerator-based systems at large scales. We also intend to use other types of accelerators, including GPUs, as well as experiment with alternative design decisions with respect to the head and I/O nodes, including experimentation with high-performance file systems for parallel I/O. We also plan on deploying our framework for capacity planning and rightsizing of clusters given specific budgets. We have evaluated our approach using dedicated resources for running standalone applications. In the future, we intend to evaluate it in a virtualized setting, to explore performance robustness under dynamic execution conditions and contention. Finally, one of our main goals is to use our framework as a production-level programming system for scientific data processing.

Acknowledgments

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

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

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

[4] AMD. The Industry-Changing Impact of Accelerated Computing. 2008.

[5] Apache Software Foundation. Hadoop, May 2007. `http://hadoop.apache.org/core/`.

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

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

[8] L. A. Barroso, J. Dean, and U. Holzle. 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. IPDPS.*, 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] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating Computing with the Cell Broadband Engine Processor. In *Proc. CF'08.*, 2008.

[12] 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, Madison, WI, 2007.

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

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

[15] 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.

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

[17] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *Proc. IEEE PACT.*, 2008.

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

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

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

[21] 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.

[22] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[23] 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.

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

[25] Message Passing Interface Forum. MPI2: A Message Passing Interface Standard. *Int. J. 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] A. K. Nanda, J. R. Moulic, R. E. Hanson, G. Goldrian, M. N. Day, B. D. D'Arnora, and S. Kesavarapu. Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers. *IBM J. Res. Dev.*, 51(5):573–582, 2007.

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

[29] 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. IPDPS.*, 2007.

[30] 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. ISCA.*, 2004.

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

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

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

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

[35] 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.

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

[37] 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. SC*, 2008.

[38] H. Wong, A. Bracy, E. Schuchman, T. Aamodt, J. Collins, P. Wang, G. Chinya, A. Groen, H. Jiang, and H. Wang. Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor. In *Proc. PACT.*, 2008.

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