1 Introduction

MapReduce is a simple and flexible parallel programming model initially proposed by Google for large scale data processing in a distributed computing environment\(^1\). In this submission, the author presents an implementation of MapReduce for the Cell BE Architecture. The submission includes source code for the runtime, an API customized for the Cell architecture, a benchmark and testing framework, and performance data for synthetic benchmarks representative of different application genres. The main contribution is an efficient, scalable, programmer-friendly, and light-weight parallel runtime for the Cell BE Architecture.

2 MapReduce

MapReduce divides data-parallel work among available processors in a Map phase, groups the output, and then optionally feeds it into a Reduce phase for post-processing. The model is attractive for its simple, high-level interface and its applicability to a range of applications. This work is further motivated by the observation that the types of applications that perform well under MapReduce are the types of applications to which the Cell architecture is especially well suited — applications that are easily divisible into streams of computation — and that the work coordination and scheduling tasks of the MapReduce master processor are control intensive and therefore ideally suited for the Cell’s PPE. Using MapReduce on Cell, developers can focus solely on the computation component of their application, taking advantage of the Cell’s vast performance potential. Communication, explicit memory management, synchronization, and scheduling complications otherwise inherent to programming directly for the Cell platform are automatically handled by the runtime.

3 MapReduce on Cell

The Cell MapReduce runtime’s execution is broken up into 5 phases: Map, Partition, Quick-sort, Merge-sort, and Reduce. In the Map phase, data is divided into equal-sized chunks and handed down to the SPEs to execute the user-provided map function on each data element. The map function outputs a key/value pair. Output elements with the same key are grouped together and handed down to the SPEs to execute the user-provided reduce function in the Reduce phase. The Partition, Quick-sort, and Merge-sort phases accomplish the key grouping as a three-step process of hashing, in-place sorting, and external sorting — hashing divides keys into buckets, and sorting subsequently groups values with identical keys into contiguous memory regions. Figure 1 sketches an example implementation and the execution flow on the Cell architecture for a hypothetical word-count application.

```c
word-count(String str) {
    for each word w in str
        if (defined map[w])
            map[w]++;
        else map[w] = 0;
    emitIntermediate(w, 1);

    reduce(String key, int[] values) {
        result = 0;
        for each v in values
            result += v;
        emit(result);
    }
}
```

Figure 1: Example program and execution mapping

The MapReduce on Cell API is documented in the files `include/mapReduce.h` and `include/mapReduce_spu.h`. This submission includes complete source code for the run-time, Makefiles, instructions to build the run-time, notes on writing, building, and running new applications using this framework, and example applications. An additional technical report describes the design and implementation of the runtime.

\(^1\)MapReduce: Simplified Data Processing on Large Clusters, by Jeffrey Dean and Sanjay Ghemawat. OSDI 2004: Sixth Symposium on Operating System Design and Implementation
4 Execution Analysis

This section presents a brief analysis of the runtime by examining scalability, performance, and efficiency. A set of synthetic benchmarks were developed that cover a diverse application space and target specific features of the model. The benchmarks are: *linearRegression* which computes summary statistics for computing a line of best fit for a set of coordinates, *sort* which sorts input data, and *sqrtCount*, a microbenchmark which takes as input a list of values and counts the number of values that have the same integer square root value (rounded down).

**Scalability with the input data set size:** For all applications, the MapReduce runtime performance scales linearly with the input size as shown in Figure 2a. At very small input sizes, performance scales sub-linearly due to initialization and finalization overheads.

**Scalability with the number of SPEs:** Figure 2b shows the average speedup of the benchmarks relative to one SPE as the number of SPEs is increased. The benchmarks are divided into two categories: *high-compute* benchmarks perform thousands of operation for each data element, and *low-compute* benchmarks perform tens of operation for each data element. While performance scales linearly for the *high-compute* benchmarks, the overhead of the Partition phase dominates the *low-compute* benchmarks, which achieve only a modest speedup with 8 SPEs. The accompanying technical report describes a planned optimization to the runtime for distributing the partitioning overhead across the SPEs to yield an expected two- to four-fold reduction in execution time for these *low-compute* applications.

**Performance:** For a fair performance comparison, the benchmarks were also implemented as single-threaded C programs compiled using gcc at full optimizations on Fedora Core Linux. The execution time was measured on an Intel Core 2 Duo 6600 with a 4MB L2 cache running at 2.4Ghz. Figure 2c shows relative speedup of the MapReduce applications running on a Cell QS20 Blade Server compared to this baseline. It is clear that application performance using MapReduce is sensitive to the amount of computation performed in the map function. Applications that perform many thousands of instructions per map output pair, such as *linearRegression*, outperform their x86 single-threaded counterparts. Other applications such as *sqrtCount* and *sort*, which belong to the *low-compute* category, perform worse, as they are limited by the overheads of partitioning.

**Efficiency:** Any application written for the Cell architecture must handle DMA transfer, synchronization, communication, scheduling, and buffer management. These functions are automatically performed by the MapReduce runtime. The efficiency of the runtime can be measured by how well it can hide these overheads and perform them concurrently with useful computation. For *high-compute* applications that fit the execution model well, the measured overhead is less than 4%. For *low-compute* applications, the overhead ranges from 60% to 98%, although this can be considerably reduced by future enhancements.

5 Summary

**Innovation:** This runtime is the first of its kind for the Cell architecture, freeing programmers from the responsibility of data partitioning, memory management, communication, synchronizing and threading and allowing them to focus exclusively on core application functionality.

**Usability:** The flexibility and simplicity of the programming model and runtime can provide enormous productivity benefits and make the architecture accessible to many domains and novice users.

**Performance:** On compute intensive applications, the runtime shows in excess of a 2.5X performance improvement over a 2.4GHz Intel Core2 processor, with linear scaling as more SPEs are added. The overheads of the runtime are also minimal at less than 4%. The runtime has potential for high performance in real-world applications as it exhibits high performance on synthetic benchmarks that are representative of a diverse application space.

In conclusion, this MapReduce implementation for the Cell BE Architecture provides an efficient, scalable, programmer-friendly, and light-weight parallel runtime that opens up the Cell architecture to a new class of users, while providing productivity benefits to both novice and experienced users alike.