Bitonic-MapReduce: Optimization of MapReduce on the Cell B.E. Architecture with a Bitonic Sort

Senior Honors Thesis

Matt Sinclair
Advisor: Karthikeyan Sankaralingam
Department of Computer Sciences
University of Wisconsin-Madison
sinclair@cs.wisc.edu

Abstract

The Cell B.E. Architecture is a novel, heterogeneous, multi-core architecture that offers opportunities for significant performance. However, a lack of programmer familiarity with explicitly parallelizing code and difficulty using its unique software-managed memory model make writing programs for the Cell difficult, even for experienced programmers. However, if tools can be made to abstract away the issues that frustrate programmers, then programmers can concentrate on what they do best – coding. MapReduce for the Cell processor is a previously implemented runtime that provides a tool to abstract away these issues from programmers; it explicitly parallelizes the user's code internally and handles all memory management internally. However, MapReduce for the Cell processor is slowed by a bottleneck in its sorting and grouping phases. This bottleneck is caused by the use of quicksort, which is unable to exploit the data-level parallelism the Cell generates much of its computational power from. In this work, we replace quicksort with Cellsort, an optimized bitonic sort implementation which can better exploit the Cell's fine-grained and course-grained parallelism. After converting Cellsort to sort (key, value) pairs, a requirement of the MapReduce framework, Cellsort was integrated into MapReduce. Finally, the characteristics of the Cellsort and MapReduce models were studied.

1 Introduction

The Cell processor is a heterogeneous, multi-core architecture [5] that provides high performance at relatively low power. It addresses the challenges of design complexity, memory latency, and power using a novel design. In order to avoiding the costs of accessing memory, the Cell processor uses software-managed caches, which move the complexity of accessing memory from the hardware to the programmer with a DMA (Direct Memory Access) software-managed memory model. Removing the hardware-caching of data from the processor alleviates the power consumption issues that plague many processors. A major advantage of the Cell's design is its heterogeneous, multi-core approach which offers a magnitude of performance increase over conventional architectures.

However, to harness this power programs must be explicitly written to exploit the specialization of the Cell and must be explicitly partitioned so that they can be assigned to the different cores. This partitioning requires knowledge of sophisticated programming techniques. Additionally, many programmers are unfamiliar with the software-managed memory of the Cell processor. Together, these two issues make writing effective programs for the Cell process extremely difficult.

MapReduce for the Cell processor [6] is a runtime tool that makes writing programs for the Cell processor easier. It does this at runtime by parallelizing a users' code based on provided Map and Reduce functions. It also internally handles all memory accesses. While MapReduce for the Cell processor represents a significant step forward in providing programmers useful tools to help write programs for the

Cell processor, it is slow for some types of applications, particularly ones that require significant amounts of sorting to be done. The biggest opportunity for speeding it up is in its sorting and grouping phases. This report focuses on optimizations to the MapReduce for the Cell runtime in these phases.

Sensitivity studies [6] showed that the bottleneck of MapReduce for the Cell processor is in the sorting technique for the sorting and grouping phases. These phases currently use quicksort. While quicksort is a fast sorting algorithm, it does not work well with highly parallel architectures such as the Cell. Quicksort is not capable of exploiting data-level parallelism, from which the Cell derives much of its computational power. This issue makes quicksort ill-suited for the Cell architecture. Recently Cellsort [3], an optimized bitonic sort [1], was shown to perform better on the Cell processor than quicksort. This makes it well-suited to replace quicksort in the MapReduce for the Cell processor framework. In this work, we extended Cellsort to sort (key, value) pairs and then integrated it into the MapReduce framework.

The original Cellsort only sorted keys (which is why we refer to it as *BitSort_keys*), while MapReduce for the Cell processor sorts (key, value) pairs. Because Cellsort was designed to fill the entire data section of a local store (LS, to be covered in *Section 2.1*), this required significant modifications, as the keys could only take up half of the space in the data section of an LS now (with the values taking up the other half).

We replaced quicksort with Cellsort in MapReduce for the Cell processor; this implementation is referred to as Bitonic MapReduce for the Cell processor, or Bitonic MapReduce for short. Cellsort is a tiered sorting approach; it uses a *local sort* for small amounts of elements, then an *in-core sort*, and finally an *out-of-core sort*. The in-core sort builds upon the local sort and similarly for the out-of-core sort. In this work, we implement the local and in-core sorts. In future work, we will implement the out-of-core sort. After integrating Cellsort into the MapReduce for the Cell processor framework we performed functional and performance testing of the new Cellsort, which we refer to as *BitSort_pairs*, and Bitonic MapReduce. The functionality testing tested the correctness of the results for datasets of various sizes; it showed that BitSort_pairs and Bitonic MapReduce functioned correctly for datasets up to 256 KB in size, the max in-core sorting size.

The Cellsort performance testing compared sort execution time with the number of items sorted for BitSort_pairs, BitSort_keys, and quicksort. The results of these tests confirmed the results from the original Cellsort implementation [3] and found that the overhead of sorting (key, value) pairs is minimal; BitSort_pairs requires only a small amount more of time than BitSort_keys did. Because the out-of-core sorting is not yet implemented, only the Map Intensity tests from the MapReduce for the Cell processor micro-benchmark [6] could be performed. As Map Intensity increases, more work is done in the Map function. The Map Intensity tests show that Bitonic MapReduce requires less sorting time than MapReduce for the Cell processor does and that the overall execution time is approximately linear. Additionally, performance tests for increasing input size showed that Bitonic MapReduce provides significant performance increases over MapReduce for the Cell processor as the parallelism (and input size) increase.

The rest of the paper is organized as follows. In *Section 2*, we present background on the Cell processor, MapReduce, bitonic sort, and Cellsort. In *Section 3*, we discuss the implementation of the new Cellsort and Bitonic MapReduce. In *Section 4*, the results of the functionality and performance testing are presented and analyzed. In *Section 5*, provides some lessons learned during this project. In *Section 6*, we discuss future Bitonic MapReduce work. Finally, in *Section 7*, we conclude.

2 Background

2.1 Cell Processor

The Cell is a radical departure from previous architectures; multi-core architectures generally use a homogeneous approach, providing multiple copies of the same core while the Cell processor uses a heterogeneous approach. It is optimized for large-scale, parallelized floating-point computations. This makes the Cell processor ideal for computation-intensive workloads and graphics-rich computations [7]. The Cell is able to achieve this massive floating-point computational power through its highly parallelized architecture. Detailed information on the Cell processor's design and implementation can be found elsewhere [5, 7].

An architectural diagram of the Cell processor can be found in *Figure 1*. The Cell processor's PPE (PowerPC Processing Element) is the master of the Cell processor – all work sent to the processor is sent to the PPE. The PPE is then responsible for distributing this work to the eight identical SPEs (Synergistic Processing Elements). These SPEs, the novel part of the Cell processor, are the slaves of the system. They are optimized for fast, computation-intensive processing instead of being optimized for general-purpose processing. As a result, the SPEs provide a significant portion of the computational power in the Cell processor [7].

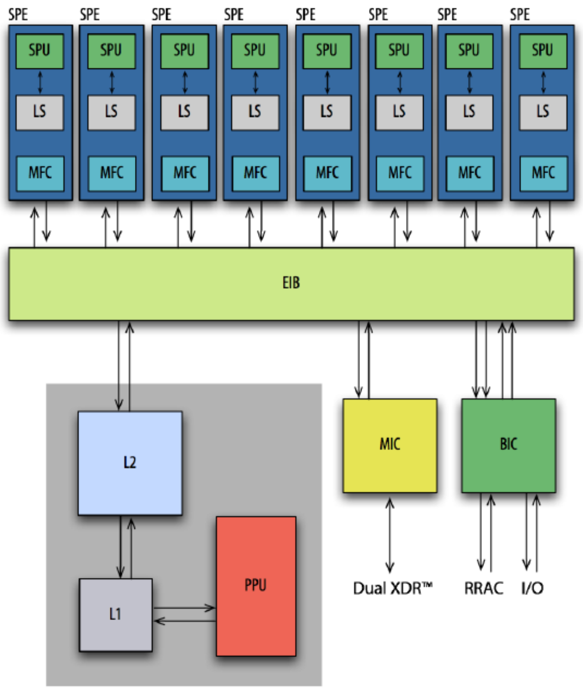

Figure 1: An overview of the Cell B.E. Architecture, reproduced from [2]. There are eight identical SPEs, which provide the bulk of the computational power of the Cell processor. There is also a single PPE which acts as the master of the processor. The EIB is the bus which the PPE and SPE communicate across.

Another unique feature of the Cell processor is its memory management system. The Cell processor uses a software-managed memory model. Each SPE has its own 256 KB memory called a Local Store (LS). The LS is responsible for storing both the program code and the data for a given SPE. The SPEs are not capable of accessing main memory directly, an important feature of the model. To read or write from a SPE's LS, a software command must be issued [8]. The command communicates with main memory, reading data from main memory and placing it into the LS or writing data from the LS into main memory. Thus, the LSs require explicit software management (typical strategies manage memory in hardware), which provides the opportunity to prevent memory latency from severely degrading performance.

The Cell's memory model is designed to reduce total memory access latency by reducing accesses to main memory. The Cell avoids this slowdown by accessing main memory as infrequently as possible. By using explicit software commands to transfer information between the LS and main memory, the Cell processor potentially accesses main memory infrequently. When it does access main memory, it transfers blocks of data at a time, which is beneficial because it amortizes the cost of accessing main memory over multiple blocks of data. Also, because the Cell's memory model is asynchronous, multiple software commands can be in-flight simultaneously [8].

2.2 MapReduce

MapReduce, originally designed at Google, is a parallel programming model [4]. After users specify a *Map* and a *Reduce* function, MapReduce automatically parallelizes the user's code, and then distributes the parallelized code to be processed on multiple machines. The automatic parallelization and distribution of the user's code allows programmers to obtain high performance without having to manage the details of parallel programming and distributed systems [4].

MapReduce parallelizes the user's code and distributes it to multiple machines through the use of *Master* and *Worker* threads. The workflow for this process can be seen in *Figure 2*. The Master thread is responsible for dividing up the work to be done and sending it to various distributed machines. The Worker threads receive the work that the Master assigns them and then perform the assigned work on a machine in the distributed machine network.

The ability of MapReduce to parallelize and distribute a user's work without requiring the user to manually write parallelized code is crucial in its extension to the Cell processor [6]. Understandably,

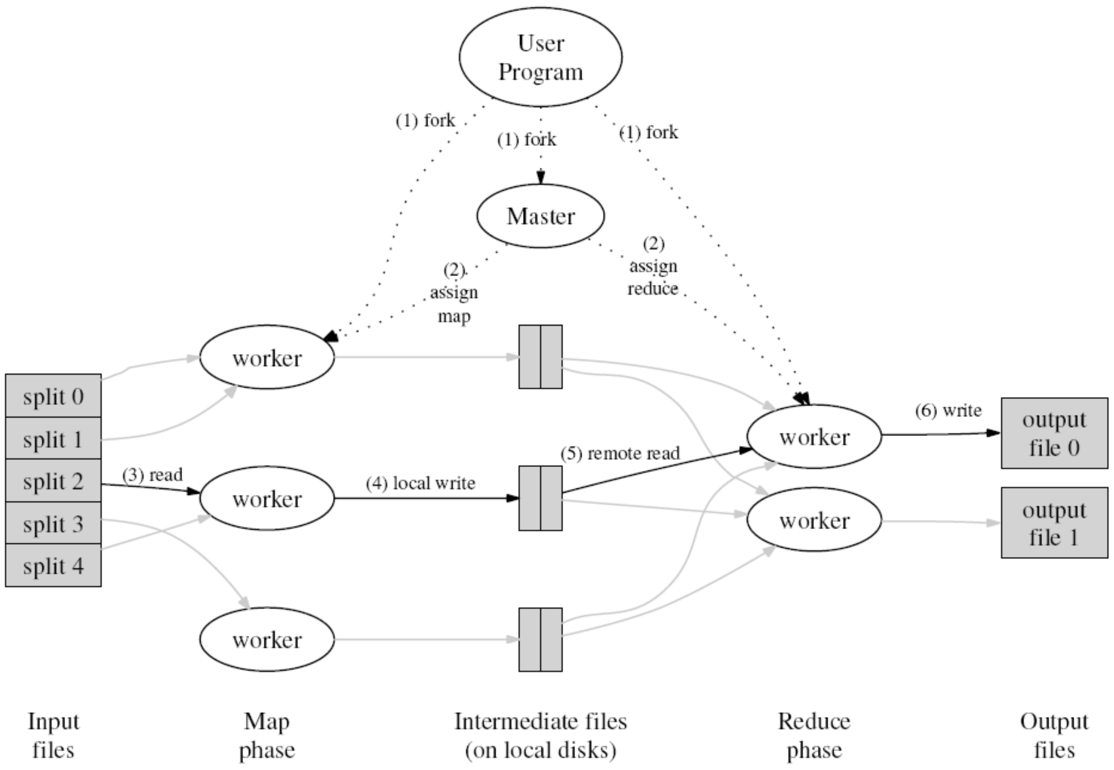

Figure 2: The MapReduce Framework, reproduced from [4]. Users provide *Map* and *Reduce* functions. Then, MapReduce automatically parallelizes the user's code and then executes the parallelized code on a large cluster of machine.

MapReduce for the Cell processor operates very similarly to original MapReduce (*to be referred to as MapReduce*). After the user specifies the Map and Reduce functions, MapReduce for the Cell processor then automatically parallelizes the user's code and internally handles the memory calls, abstracting this issue away from the programmer. The Cell processor's PPE performs the duties that MapReduce's Master thread performed and the SPEs perform the tasks of the Worker threads.

2.3 – Bitonic Sort and Cellsort

Bitonic sorts [1] perform many comparisons in parallel. Larger bitonic sorts are composed of multiple levels of smaller bitonic sorts, which makes bitonic sort similar to a parallelized merge sort. *Figures 3* and *4* show examples of bitonic sorts for n-items and 8-items. For readers who are interested in more details on how bitonic sorts work, [1] provides a very good explanation of them. The ability to construct larger bitonic sorts out of smaller bitonic sorts allows a tiered or hierarchical sort such as Cellsort to be constructed.

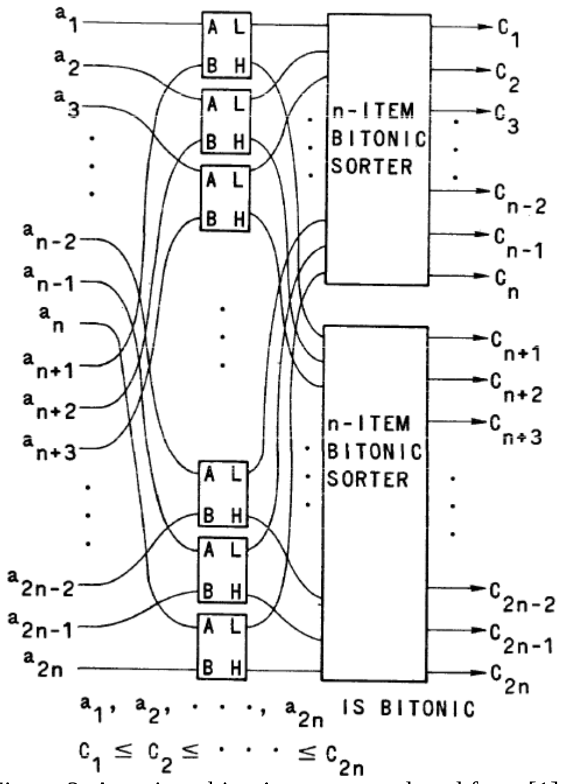

Figure 3: An n-item bitonic sort, reproduced from [1]. A bitonic sort builds upon itself, creating larger bitonic sorts from smaller ones. For the bitonic sort to work correctly it requires that the sequences be *bitonic*.

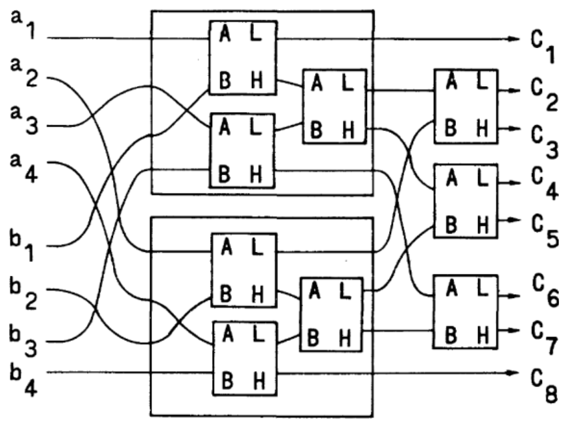

Figure 4: An 8-item bitonic sort, reproduced from [1]. This sort is identical to that found in *Figure 4*. Larger bitonic sorts are composed of multiples levels smaller bitonic sorts.

Cellsort is an optimized bitonic sort for the Cell processor. It was designed to take advantage of the highly parallel nature of the Cell processor and it uses a 3-tiered sorting approach, shown in *Figure 5*. The local sort is performed if the number of elements to be sorted can fit inside the LS of a single SPE. If the number of elements is larger than the number of elements that can fit inside the LS of a single SPE, but smaller than the number of elements that can fit inside the LS of all of the SPEs, then an in-core sort is performed. In an in-core sort, each SPE performs a local sort, and then the results from each SPE are merged. An out-of-core sort is performed if the number of elements it greater than the number of elements that can fit inside the LSs of all of the SPEs. This means that we will have to access main

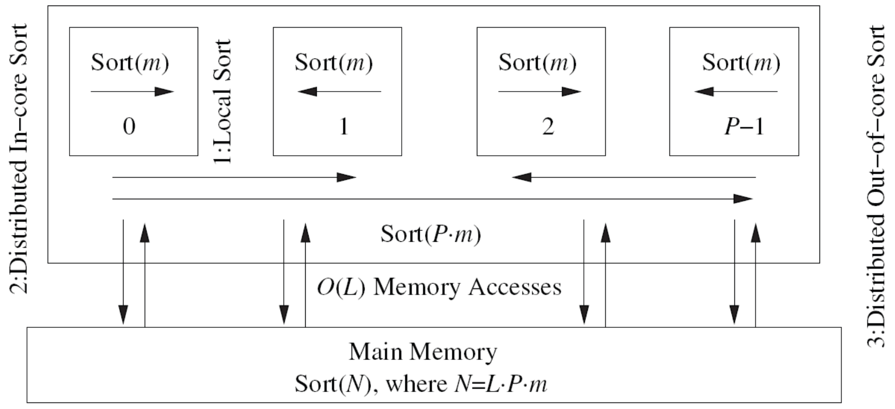

Figure 5: The tiered sorting technique of Cellsort, reproduced from [3]. Cellsort is a bitonic sort that optimizes its local and in-core sorts to avoid accessing main memory. In this figure, m represents the number of items a single SPE can sort, P represents the number of SPE's available, and L represents the number of accesses to main memory.

memory, which we didn't need to do in the local and in-core sorts because we exploited the locality of the LSs. To perform an out-of-core sort, we perform break the elements into sets, such that each set can fit inside the LSs of all of the SPEs. Then, we perform an in-core sort on those elements. This process is repeated until all elements have been sorted, and then the results are merged together.

A key trait of the local and in-core sorts is their exploitation of the locality of the LSs. This minimizes the number of accesses to main memory, which allows high performance sorting to be obtained. It achieves this efficient and optimized sorting through the use of SIMD (single instruction, multiple data) instructions. SIMD instructions are similar to vector instructions; they are 128-bits wide, while normal instructions are 32- or 64-bits wide. Thus, SIMD instructions allow multiple values to be processed simultaneously. Cellsort obtains better performance on the Cell processor than quicksort because it uses SIMD instructions.

3 Implementation

Our work extends Cellsort [3] from sorting only keys (*BitSort_keys*, a single array) to sorting (key, value) pairs (*BitSort_pairs*, two arrays, with dependencies between the arrays). After extending Cellsort to sort (key, value) pairs, we then integrate it into MapReduce for the Cell processor. The resulting implementation we refer to as *Bitonic MapReduce*.

The extension of BitSort_keys to sort (key, value) pairs is necessary because MapReduce for the Cell processor requires the use of (key, value) pairs. Functionally, in BitSort_keys, whenever two keys were swapped, they were swapped as in *Figures 3* and *4*. In BitSort_pairs, the keys are swapped and the associated values are also swapped.

After modifying BitSort_keys to sort (key, value) pairs, we verified the functionality of BitSort_pairs. The results showed that BitSort_pairs correctly sorted elements for the local and in-core sorting levels. We hope to implement out-of-core sorting in future work. After verifying the functionality of BitSort_pairs, we integrated BitSort_pairs into the MapReduce for the Cell processor framework. We used several tests to ensure the functional correctness of our implementation. The output of these tests confirmed that our sort was functioning correctly for both local and in-core sorts.

Integrating Cellsort into the MapReduce for the Cell processor framework has several implications. First, it provides a faster sorting kernel for Bitonic MapReduce to use while abstracting away from the programmer the details of the sort (while the sort itself does not make it easier to program, it does enable MapReduce for the Cell processor to more efficiently convert programs to a parallelized, Cell-friendly form). More efficient sorting will enable better processor resource utilization. The integration of bitonic sort into MapReduce for the Cell processor represents an internal improvement that can be leveraged to execute parallel programs. Additionally, the use of the quicksort and merge sorts in the sorting and grouping phases of MapReduce for the Cell processor required the use of several threads in order to perform these tasks. While MapReduce for the Cell processor used eight threads (two threads per SPE) with the use of quicksort, Bitonic MapReduce uses only one thread. Thus, using Cellsort removes seven threads from the runtime when four SPEs are used; this removes the overhead of swapping values between threads and improves performance.

4 Results

In this section, we present our methodology and then evaluate our BitSort_pairs and Bitonic MapReduce implementations.

4.1 Methodology

We ran all of our tests on a Playstation 3, which contains a Cell processor with six SPEs. In our tests we used up to four SPEs. Additionally, we used Yellow Dog Linux 6.1 as the operating system and we installed the Cell SDK 3.1 to enable compilation and testing. All of our code was written in C and C++, with GCC being used as the compiler and O3 being used as the optimization level. After completing the integration of BitSort_pairs into Bitonic MapReduce, we performed functional and performance tests on BitSort_pairs and Bitonic MapReduce.

For the testing of BitSort_pairs, we used various inputs, ranging in size, in powers of two, from 16 KB to 16 MB, of random integers. In these tests, we kept the inputted stream of keys and values identical to simplify the verification process. The metric used in the testing of the new Cellsort was sort execution time. The relationship between dataset size, number of SPEs, and Cellsort sorting type can be found in *Table 1*. We then compared the results for BitSort_pairs to results obtained from quicksort and BitSort_keys.

Dataset size	**# of SPEs**	**BitSort_pairs sorting type**
$\leq$ 64 KB	1	Local
128 KB	2	In-core
256 KB	4	In-core
> 256 KB	4	Out-of-core

Table 1: Relationship between Dataset size, number of SPEs, and BitSort_pairs sorting type.

For the testing of Bitonic MapReduce, a square root test was used. The output of Bitonic MapReduce for this test was then verified. MapReduce for the Cell processor used several different tests for performance testing. One of the tests used was a micro-benchmark [6], which tested how MapReduce for the Cell processor performed for six different types of parameters that fully classify the behavior of applications in the MapReduce for the Cell processor runtime. Because we only implemented the local and in-core sorts, a full evaluation of these six parameters could not be undertaken. Only the Map Intensity parameter tests worked with only the local and in-core sorts. Map Intensity is a measure of how much work the user is doing in the inputted Map function [6]. The Map Intensity was varied from 1 – 256 to simulate increasing Map Intensity. We also varied the number of SPEs from 1 to 4 in our tests. MapReduce for the Cell processor also used a suite of real-world tests but these tests also require the use of the out-of-core sort. Thus they were not evaluated either.

4.2 Cellsort Results

When these tests were run, the expectation was that sorting the (key, value) pairs would require more time than either quicksort or BitSort_keys; in the worst case, BitSort_pairs could take twice as long the other sorts. BitSort_pairs sorts (key, value) pairs, whereas BitSort_keys and quicksort only sort keys – half the amount of information BitSort_pairs sorts.

Figure 6 compares the sorting performance of BitSort_keys, BitSort_pairs, and quicksort. The encouraging news is that BitSort_pairs requires nowhere near twice as long as the other sorts. *On average, BitSort_pairs required 16.1% longer time than BitSort_keys and 8.5% longer time than quicksort*. This amount of time required to sort the extra array of elements is quite small. This is likely because the extra array doesn't require any additional comparisons in the bitonic sort. Recall from *Section 3* that the items in the Values array are simply swapped if the bitonic sort swaps the items in the Keys array. Since the arrays are always in the LS, swapping them requires very little extra time. It will be interesting to see if this holds for sorts that require out-of-core sorting.

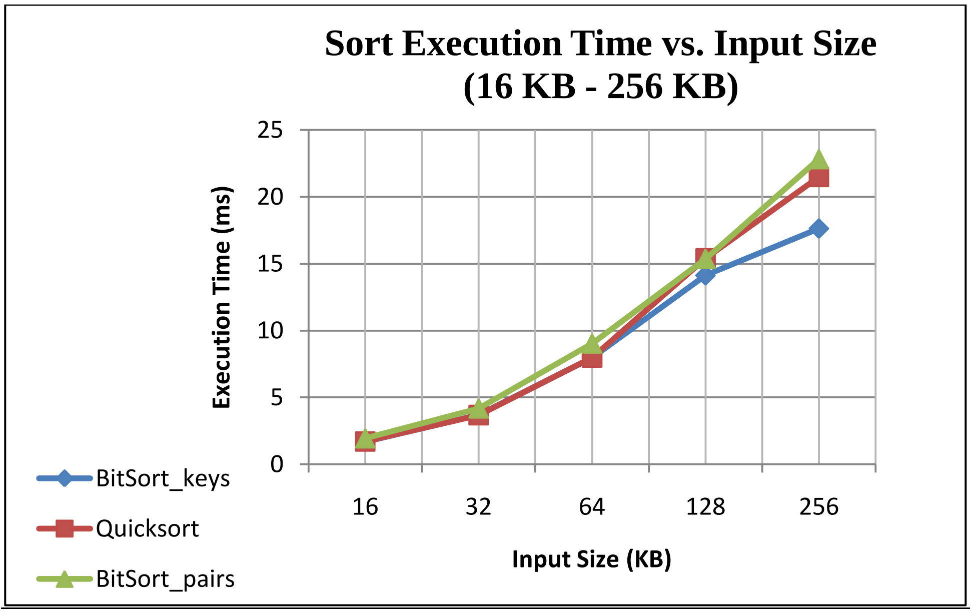

Figure 6: Plot of Sort Execution Time vs. Input Size for quicksort, BitSort_keys, and BitSort_pairs (16 KB – 256 KB). The input sizes in this graph range from 16 KB – 256 KB (**local and in-core sorts**).

Figures 7 and *8* compare the execution times of BitSort_keys and quicksort on datasets that require out-of-core sorting. *Figure 7* confirms the result that BitSort_keys is faster than quicksort on the Cell processor, especially for sorts that require more SPEs. *Figure 8* also demonstrates the prohibitive cost of accessing main memory – doubling the input size causes a quadratic increase in execution time.

These tests were done to demonstrate the functionality and performance of BitSort_pairs as compared to BitSort_keys and quicksort on the Cell processor. They showed that the new Cellsort functions correctly for datasets up to 256 KB in size. The results also showed that while sorting (key, value) pairs takes

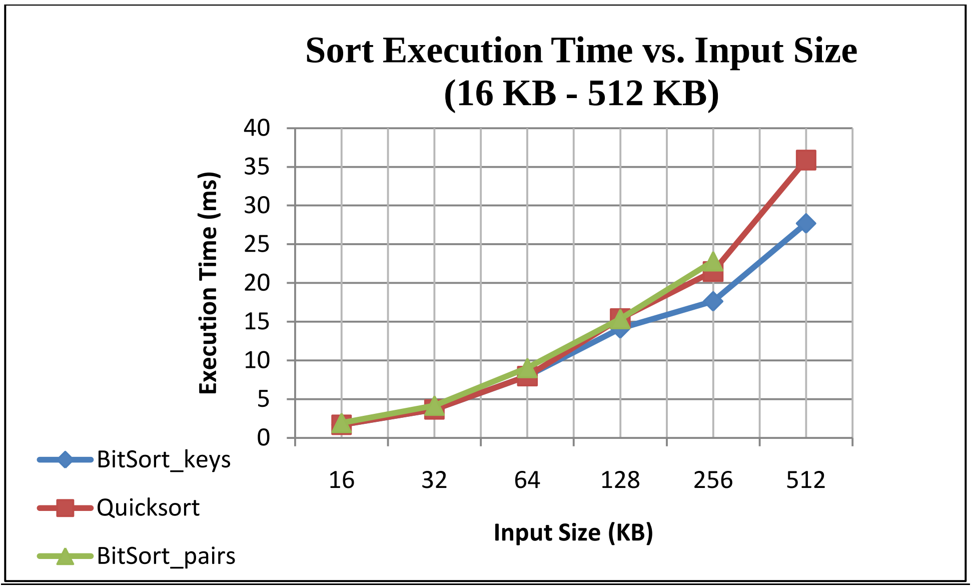

Figure 7: Plot of Sort Execution Time vs. Input Size for quicksort, BitSort_keys, and BitSort_pairs (16 KB – 512 KB). The input sizes in this graph range from 16 KB – 512 KB (**local, in-core, and out-of-core sorts**).

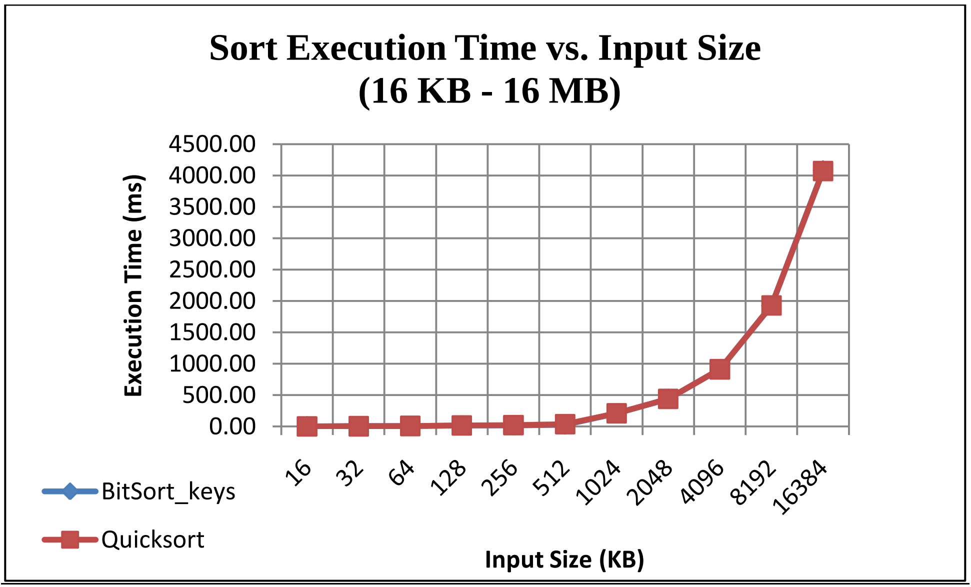

Figure 8: Plot of Sort Execution Time vs. Input Size for quicksort, BitSort_keys, and BitSort_pairs (16 KB – 16 MB). The input sizes in this graph range from 16 KB – 16 MB (**local, in-core, and out-of-core sorts**). Note that the lines are nearly on top of each other due to the huge increase in execution time.

longer than sorting only keys, the amount of additional time needed to perform this sorting is small. Additionally, the results of the original Cellsort tests were confirmed.

4.3 Bitonic MapReduce Results

The expectation is that execution time scales linearly with increasing Map Intensity because the Map Intensity tests do not modify input size. Instead, the Map Intensity tests modify how much work is done in the Map function. The expectation for overall execution time is that there will not be a significant performance improvement, since the Map Intensity tests are map-dominated, which means that the amount of time required for sorting is small. It is also expected that Bitonic MapReduce will require less sorting time than MapReduce for the Cell processor and that both will behave in a linear pattern. The results for sorting time are shown in *Figures 9* and *10* and the results for overall execution time are shown in *Figure 11*.

Figure 9 shows how sort time varies against Map Intensity for a single SPE for the original MapReduce for the Cell processor and for Bitonic MapReduce. Our expectation for a linear behavior compared to sorting time was not met in this test. The sort times for MapReduce for the Cell processor vary significantly. This can mean several things. We could have not run enough samples to obtain an accurate measurement; the trend to the data is definitely upward, which was the expectation, so it's possible the Map Intensity = 32 tests are an outlier. It is also possible our implementation of the timing metric is incorrect. Or the sort time, only one component of the total time

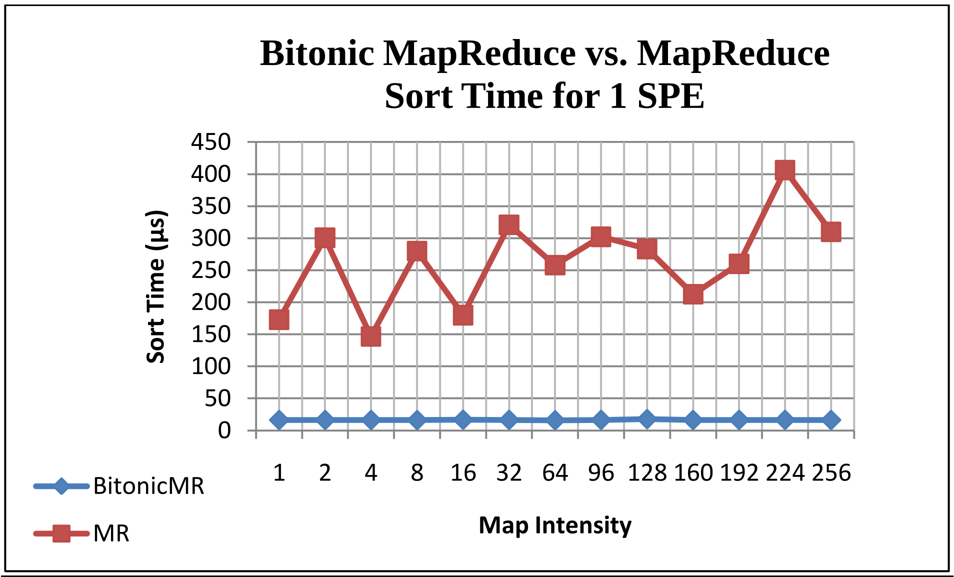

Figure 9: Sort Time vs. MapIntensity (1 SPE) – Bitonic MapReduce vs. MapReduce.

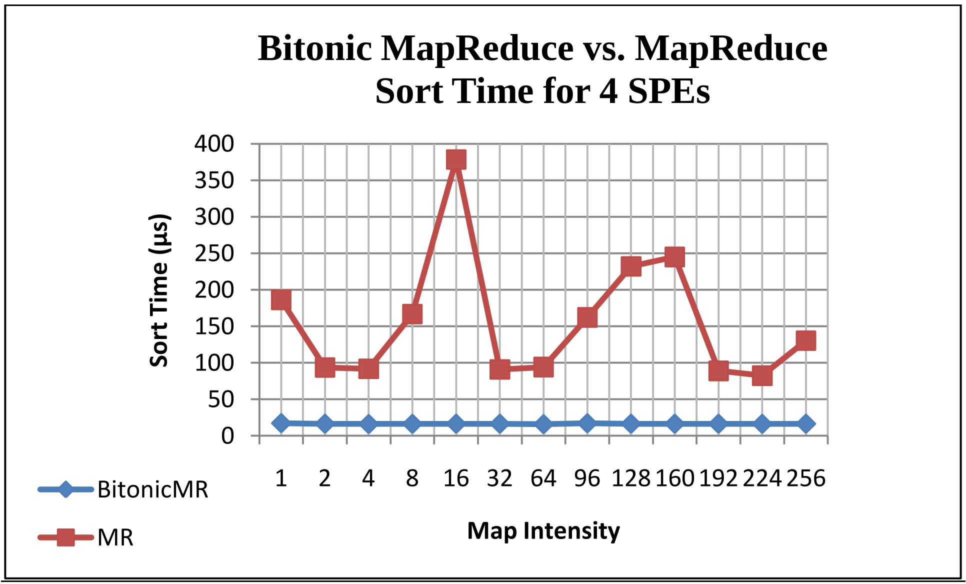

Figure 10: Sort Time vs. MapIntensity (4 SPEs) – Bitonic MapReduce vs. MapReduce.

spent executing, could be non-linear. Regardless, these results definitely show the need for further tests to be performed.

The results in *Figure 10* (4 SPE tests) are similar in that they show a non-linear behavior for MapReduce for the Cell. The results for Bitonic MapReduce in both *Figures 9* and *10* show that Bitonic MapReduce requires less time sort time than MapReduce for the Cell processor in all cases. *For 1 SPE Bitonic MapReduce requires 12.2% less time than MapReduce. For 4 SPEs Bitonic MapReduce requires 2.6% less time than MapReduce.* While there remain some questions as to the validity of the results, this result does match expectations.

Figure 11 shows that the overall execution times for MapReduce and Bitonic MapReduce are *nearly identical* for all Map Intensities (ranging from 1 – 256) when a single SPE is used. We also ran the same MapIntensity tests on four SPEs and found that the execution time was also nearly identical in this case.

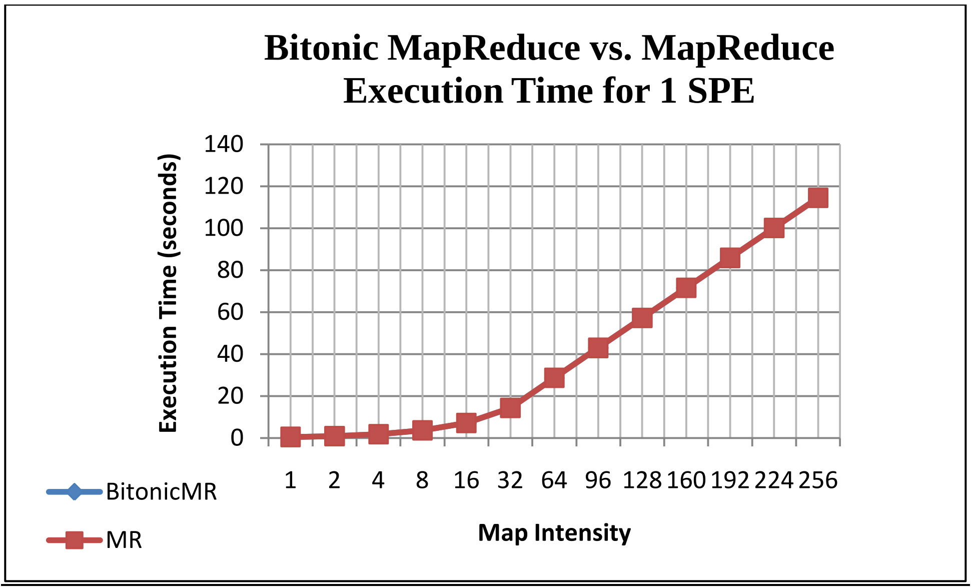

Figure 11: Overall Execution Time vs. MapIntensity (1 SPE) – Bitonic MapReduce vs. MapReduce.

In addition to the Map Intensity tests, we also obtained performance results for sorting time and execution time for the square root test. We expect the results from these tests to show that execution time increases significantly as input size increases, especially for MapReduce for the Cell processor, which uses quicksort. As input size increases, more SPEs are used to sort the data. The use of more SPEs causes more overhead from the extra threads used in quicksort. In comparison, Bitonic MapReduce should not show significant increases as the input size increases. Because we're using only local and in-core sorts, the Bitonic MapReduce sort times should be approximately identical due to the parallelization and lack of overhead of BitSort_pairs. The results from these tests are shown in *Figures 12* and *13*.

Figure 12 compares the amount of time MapReduce and Bitonic MapReduce spend sorting for input sizes ranging from 2048 to 262144 (2 KB to 256 KB), the max in-core sort size. As the input size increases, Bitonic MapReduce requires little extra sorting time, while MapReduce requires significantly more time. This is expected, as the additional overhead MapReduce requires for its threads to perform quicksort reduce performance. As the input size increases, performance should decrease – a result we confirm here. *On average, MapReduce requires over seven times more time than Bitonic MapReduce to sort the same number of elements.*

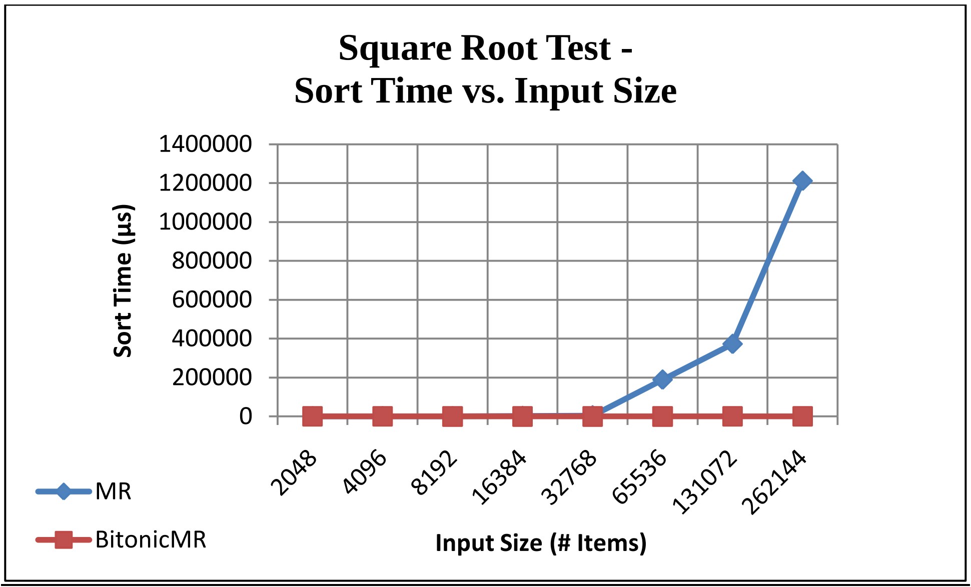

Figure 12: Square Root Test, Sort Time vs. Input Size – Bitonic MapReduce vs. MapReduce.

Figure 13 compares the execution time of MapReduce and Bitonic MapReduce for the square root test. When the input size is large enough that multiple SPEs are required (128 KB and 256 KB – in-core sorts), MapReduce requires significantly more time than Bitonic MapReduce. When a single SPE is used, however, MapReduce offers similar performance to Bitonic MapReduce. *For a 128 KB input, Bitonic MapReduce performs the same operations as MapReduce in two-thirds the time. For a 256 KB input, Bitonic MapReduce performs the same operations as MapReduce in half the time.* For a single SPE, quicksort has fewer threads and less overhead, so its performance is not degraded by the parallelization of the Cell processor. However, as the parallelization increases and more SPEs are used, that overhead begins to degrade the performance of quicksort. We expect the trend of Bitonic MapReduce providing better performance than MapReduce as input size increases to continue in future work once the out-of-core sort is integrated.

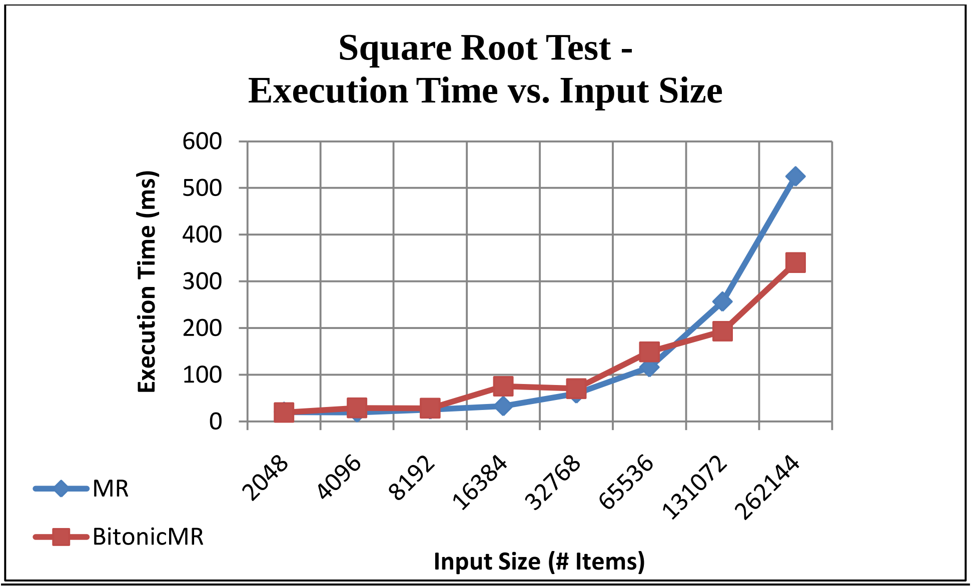

Figure 13: Square Root Test, Overall Execution Time vs. Input Size – Bitonic MapReduce vs. MapReduce.

These tests were done to determine how Bitonic MapReduce responds to increasing Map Intensity. As Map Intensity increases, more work is done in the Map function. The sorting time results did not meet our expectations, as they were non-linear. However, the results for overall execution time of the Map Intensity tests did match our expectations, as there was approximately no difference in total execution time between MapReduce and Bitonic MapReduce. The results from the square root test show that Bitonic MapReduce is able to obtain better performance than MapReduce for the Cell processor as input size increases. Overall, we found that Bitonic MapReduce requires less time to sort the items than MapReduce for the Cell processor in all cases. Further testing with the extremities for out-of-core sort will be undertaken in future work; these results will show if there are performance improvements in a different types of applications and with the out-of-core sort.

5 Lessons Learned

This section summarizes our lessons learned throughout the process of developing the runtime. Specifically, we discuss the process of setting up the Playstation 3 environment, issues in programming the Cell processor, and difficulties debugging Cell programs, specifically the barrier() bug we encountered.

5.1 Setting up the Playstation 3 Environment

Previous work in our research group on the Cell processor was done via the IBM Virtual Loaner Program, which allowed researchers access to Cell processors at IBM. However, this program has been discontinued. After looking at options such as the Georgia Tech Cellbuzz cluster [9] and the UW-Madison Metronome [10], we decided that the best course would be to purchase a Playstation 3 and install the necessary tools on it.

The directions on the Yellow Dog Linux website indicate that its distributions come with the Cell SDK (3.1 in the case of YDL 6.1) pre-installed such that you only need to install YDL 6.1 and the Cell SDK will be installed simultaneously. However, this is not true. The Yellow Dog Linux distribution did come with the Cell debuggers (ppu-gdb and spu-gdb), but it didn't come with the full Cell SDK 3.1. Thus, in order to get the full SDK working on the Playstation 3, we needed to download the Cell SDK from IBM's website and manually install it.

5.2 Programming the Cell

Previous work on the MapReduce for the Cell processor project had been done with Cell SDK 2.1 [6]. However, Yellow Dog Linux 6.1, the recommended Linux distribution for the Playstation 3, was only compatible with Cell SDK 3.1 and up. This required us to upgrade our code to be compatible with the new SDK. There were two main issues we encountered.

The first issue was Makefile-related. In Cell SDK 2.1, our Makefiles used `/opt/cell/sdk/make.footer` to link in with the Cell Makefiles (internally, `make.footer` calls `make.header`, which was located in the same folder). However, in Cell SDK 3.1, the `make.footer` and `make.header` files have been moved to `/opt/cell/sdk/buildutils/`. This required modifying our Makefiles to use `/opt/cell/sdk/buildutils/make.footer`, which then linked to make.header. Note that `/opt/cell/sdk` represents where the Cell SDK was installed; it may be different on your system if it was installed somewhere else.

The second issue was related to the files that were included in our C and C++ program files. In Cell SDK 2.1, the XL C compiler had been used. This compiler allowed the user get away without declaring the `spu_intrinsics.h`, `spu_internals.h`, or `spu_mfcio.h` files. The file that our code needed was `spu_internals.h`, which is included in both the `spu_mfcio.h` and `spu_instrinsics.h` header files. So, by including either `spu_mfcio.h` or `spu_intrinsics.h`, the implicit declaration issue (when using the gcc compilers) disappears.

5.3 Debugging Cell Programs

Perhaps the most frustrating part about using the Cell processor is debugging code on the Cell processor. Because the Cell processor is a multi-core, heterogeneous architecture, there are numerous threads at any given time. While IBM has created the ppu-gdb and spu-gdb debuggers to aid in this process, it is still difficult at times to obtain the needed information.

An error that we ran into frequently when debugging our code was the "SPU_ADDR18" error. This error occurs when your SPE code is taking up more space than a SPE can hold. Of course, optimizing your code generally removes this error, but that prevents you from debugging your code. The easiest way we found to alleviate this issue was to remove print statements from our SPE code, which take up a lot of space on the SPEs. This is a result of the Cell design, which prevents SPEs from accessing the operating system. Instead of printing information out, use the ppu-gdb or spu-gdb debuggers to stop execution at the point you are debugging.

The main issue that is preventing us from completing the out-of-core sort is a bug in the *barrier*() function, which is used by the in-core and out-of-core sorts. The barrier function is a function described in the original Cellsort paper [3]; it is responsible for synchronizing the SPEs as Cellsort is progressing. The issue in the *barrier*() function is a DMA transfer issue.

In this function, there is an *mfc_get*() command that is causing a bus error. When adding NOPs before this error, the error disappears, but re-emerges later on in the code as several values in the values array are overwritten during the final merge. When running the debugger on the *btn_core_g*() [3] function, where the values are being overwritten, it appears that the values we want are not arriving in time, which causes the swaps to be done incorrectly. At first, this seemed to have to do with the temporary variables being uninitialized, but even after fixing this issue, the same garbage values are written into the values array (but not the keys array). Even worse, when running the debugger, the values that we expect to be swapped arrive on time because the execution has been sufficiently slowed down. Adding too many NOPs into the code causes the SPU program text to be corrupted, which prevents that solution from being used.

6 Future Work

Updating BitSort_pairs for out-of-core sorting is of utmost importance. Most real datasets will use out-of-core sorting due to the limited 256 KB memory space of the SPE's local stores. Integrating the out-of-core Cellsort sort into the Bitonic MapReduce framework should be trivial.

After updating Bitonic MapReduce for the out-of-core sorts, the embedded timing and performance metrics in Bitonic MapReduce may need to be updated to obtain accurate timing and performance metrics. There also exist some new performance analysis tools [11-14] for the Cell. If these metrics prove to be more useful than the embedded MapReduce metrics, they will be used instead of the embedded metrics. Otherwise they will be used for further analysis.

After the timing modifications are made to the Bitonic MapReduce framework, complete performance results can be obtained and analyzed. Completing these tasks will allow us to obtain an accurate, comprehensive understanding of how Bitonic MapReduce performs for a data set of any size.

7 Conclusions

The Cell processor's novel, heterogeneous design offers an order of magnitude performance increase over conventional architectures. However, writing programs for the Cell processor requires the user to explicitly parallelize their code and to use the non-standard software-managed memory protocol. Many programmers simply do not possess the knowledge necessary to successfully perform these tasks, which makes writing programs for the Cell processor very difficult without tools to help them more easily harness the power of the Cell processor.

MapReduce for the Cell processor is a simple runtime that abstracts away the issues of manually writing parallelized code and dealing with the Cell's memory model. Instead, users only need to supply the Map and Reduce functions. However, MapReduce for the Cell processor is slow due to a bottleneck in its sorting and grouping phases. This bottleneck is caused by the use of quicksort, which is ill-suited for the Cell processor because it cannot take advantage of the parallelism that the Cell processor offers.

In this research, we replaced quicksort with Cellsort, a bitonic sort optimized for the Cell processor, which is able to exploit its data-level parallelism. Cellsort has been shown to be faster than quicksort on the Cell processor. Replacing quicksort with Cellsort makes MapReduce for the Cell processor faster, which allows the processors resources to be allocated more efficiently. Converting Cellsort to sort (key, value) pairs instead of keys only incurs only minimal overheads due to the Cell's ability to store both arrays in the local store(s) of SPE(s). The integration of Cellsort and MapReduce to create Bitonic MapReduce has been verified for datasets up to 256 KB. Preliminary performance studies show that BitSort_pairs requires 16.1% longer time to sort (key, value) pairs on the Cell processor as compared to BitSort_keys, which only sorts keys. Additionally, the new Cellsort requires 8.5% longer time than quicksort, which also only sorts keys. The preliminary Map Intensity performance studies show no decrease in execution time. The Map Intensity tests also showed that Bitonic MapReduce offers a 12.2% improvement in sorting time over MapReduce for a single SPE and a 2.2% improvement for four SPEs. The square root tests showed that Bitonic MapReduce offers significant performance improvements over MapReduce as parallelism and the input size increase. Further testing needs to be performed to obtain a complete picture of the performance of the runtime.

Acknowledgements

This work was generously supported by the University of Wisconsin-Madison with a Trewartha Honors Undergraduate Thesis Grant. The author would also like to whole-heartedly thank Marc de Kruijf and Khai Tran for their kind, patient, repeated help along the way.

REFERENCES

[1] K. Batcher, "Sorting networks and their applications," in *Spring Joint Computer Conference*, Atlantic City, New Jersey, April 30 - May 2, 1968, pp. 307-314.

[2] J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra, "The Playstation 3 for high-performance scientific computing," *Computing in Science and Engineering*, vol. 10, pp. 84, 2008.

[3] B. Gedik, R. Bordawekar, and P. Yu, "Cellsort: High performance sorting on the cell processor," presented at Very Large Data Bases (VLDB '07), Vienna, Austria, 2007.

[4] J. Dean and G. Sanjay, "MapReduce: simplified data processing on large clusters," in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04), 6-8 Dec. 2004*, Berkeley, CA, USA, 2004, pp. 137-49.

[5] M. W. Riley, J. D. Warnock, and D. F. Wendel, "Cell broadband engine processor: design and implementation," *IBM Journal of Research and Development*, vol. 51, pp. 545-57, 2007.

[6] M. de Kruijf and K. Sankaralingam, "MapReduce for the Cell BE architecture," *IBM Journal of Research and Development*, vol. 53, 2009.

[7] M. Gschwind, D. Erb, S. Manning, and M. Nutter, "An open source environment or cell broadband engine system software," *Computer*, vol. 40, pp. 37-47, 2007.

[8] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "Scientific computing kernels on the cell processor," *International Journal of Parallel Programming*, vol. 35, pp. 263-298, 2007.

[9] *Main Page - Cellbuzz*. Available: https://wiki.cc.gatech.edu/cellbuzz/index.php/Main_Page

[10] *NMI Build and Test Lab*. Available: https://nmi.cs.wisc.edu/

[11] T. Chen and D. A. Brokenshire. (July 1, 2008, October 15, 2009). BladeCenter QS: Maximizing memory performance - Compare CBEA and general-purpose processor memory access models for maximum memory performance. [Performance Analysis]. Available: http://www.ibm.com/developerworks/library/pa-qsmemperf/

[12] G. Haber. (April 15, 2008, October 15, 2009). Cell/B.E. SDK 3.0 tools, Part 1: Using performance tools. Available: http://www.ibm.com/developerworks/edu/pa-dw-pa-sdk3tool.html

[13] Q. Liang. (February 3, 2009, October 15, 2009). Performance monitor counter data analysis using counter analyzer: use counter analyzer to analyze performance monitor counter data of Power and Cell Broad Engine platform. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/aix/au-counteranalyzer/au-counteranalyzer-pdf.pdf

[14] B. S. C.-C. N. d. Supercomputación. October 15, 2009). Cell Performance Counter device driver for Cell BE. Available: http://www.bsc.es/plantillaH.php?cat_id=307&