CELLSORT: High Performance Sorting on the Cell Processor

Buğra Gedik
bgedik@us.ibm.com

Rajesh R. Bordawekar
bordaw@us.ibm.com

Philip S. Yu
psyu@us.ibm.com

Thomas J. Watson Research Center, IBM Research, Hawthorne, NY 10532

ABSTRACT

In this paper we describe the design and implementation of CELLSORT – a high performance distributed sort algorithm for the Cell processor. We design CELLSORT as a distributed bitonic merge with a data-parallel bitonic sorting kernel. In order to best exploit the architecture of the Cell processor and make use of all available forms of parallelism to achieve good scalability, we structure CELLSORT as a three-tiered sort. The first tier is a SIMD (single-instruction multiple data) optimized bitonic sort, which sorts up to 128KB of items that cat fit into one SPE's (a co-processor on Cell) local store. We design a comprehensive SIMDization scheme that employs data parallelism even for the most fine-grained steps of the bitonic sorting kernel. Our results show that, SIMDized bitonic sorting kernel is vastly superior to other alternatives on the SPE and performs up to 1.7 times faster compared to quick sort on 3.2GHz Intel Xeon. The second tier is an in-core bitonic merge optimized for cross-SPE data transfers via asynchronous DMAs, and sorts enough number of items that can fit into the cumulative space available on the local stores of the participating SPEs. We design data transfer and synchronization patters that minimize serial sections of the code by taking advantage of the high aggregate cross-SPE bandwidth available on Cell. Results show that, in-core bitonic sort scales well on the Cell processor with increasing number of SPEs, and performs up to 10 times faster with 16 SPEs compared to parallel quick sort on dual-3.2GHz Intel Xeon. The third tier is an out-of-core[1] bitonic merge which sorts large number of items stored in the main memory. Results show that, when properly implemented, distributed out-of-core bitonic sort on Cell can significantly outperform the asymptotically (average case) superior quick sort for large number of memory resident items (up to 4 times faster when sorting 0.5GB of data with 16 SPEs, compared to dual-3.2GHz Intel Xeon).

[1]The term "out-of-core" does not imply a disk-based sort in the context of this paper. However, relation to external sorting is strong (see Sections 2 and 3 for details).

1. INTRODUCTION

Sorting is, unquestionably, one of the most fundamental operations in computer science [16]. It has important uses in large-scale data intensive applications in many fields, ranging from databases [9] to computer graphics [22] to scientific computing. With the ever increasing main memory sizes and the current trend in computer architecture towards multi-core processors, the importance of main memory-based high-performance parallel and distributed sorting has become more prominent. In this paper, we describe how large number of memory resident items can be sorted using distributed bitonic sort on Cell – a heterogeneous multi-core processor [12]. Main memory-based sorting on Cell is particularly interesting from the perspective of traditional external sorting, since the co-processors on Cell are not connected to the main memory via a cache hierarchy, but instead use DMAs to transfer blocks of data to/from the main memory.

Originally designed for gaming consoles, the Cell processor has created significant interest in scientific and commercial domains. High-end Cell blade servers for general computing are commercially available [17], and research on porting various algorithms to this unconventional, yet extremely powerful processor are under way in many application domains [2, 4, 21, 26].

There exists a large body of prior work on sorting for a variety of distributed and parallel computers. However, the unique properties of the Cell processor, and heterogeneous multi-core architectures in general, necessitate revisiting this fundamental problem once again. A heterogeneous multi-core architecture is often characterized by a main processing element accompanied by a number of co-processors. For instance, the Cell processor consists of the PPE (PowerPC Processing Element) which serves as the main processing element, and the eight SPEs (Synergistic Processing Elements) which are the co-processors providing the bulk of the processing power. SPEs are pure SIMD (single-instruction, multiple data) processors. Furthermore, SPEs do not have conventional caches, but instead are equipped with local stores, where the transfers between the main memory and the local stores are managed explicitly by the application software. This is a common characteristic of heterogeneous multi-core processors, such as network processors [13].

In this paper, we describe the design and implementation of CELLSORT – a high performance sorting algorithm for the Cell processor. An implementation of CELLSORT will appear in the upcoming version 3.0 of the Cell Broadband Engine SDK, with its source code publicly available on IBM developerWorks. CELLSORT is based on *distributed bitonic*

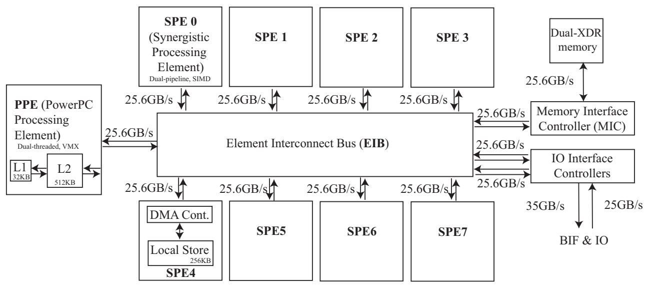

Figure 1: Architecture of the Cell processor

merge with a *SIMDized bitonic sorting kernel*. It can be used as a module to support large-scale disk-based sorts, although we do not investigate this perspective of the problem here. There are two major challenges in crafting an efficient distributed sort for the Cell processor. First, the local sorting kernel has to be carefully selected and optimized. Most of the comparison-based sorts involve unpredictable branches and are not suitable for SIMD-only processors like the SPEs on Cell. Second, the data transfers performed during in-core and out-of-core sorts (see Section 3 for definitions) should be carefully designed to minimize synchronization overhead and should take advantage of the asynchronous DMA capability to hide the memory transfer delays as much as possible.

Our results reported in this paper show that:

(*i*) SIMDization of local sorts is crucial to achieving high performance. When properly optimized, SIMDized bitonic sort kernels are significantly superior to quick sort kernels (which are not easily SIMDized) for the local sorts on the SPEs. Interestingly, the same argument does not hold for SSE-enhanced bitonic sort on Intel Xeon processors (not pure SIMD processors, unlike the SPEs).

(*ii*) Distributed in-core sort has little communication overhead. This is due to our effective design of the data transfer and SPE synchronization patterns, use of asynchronous DMAs to hide data transfer delays, and the large aggregate bandwidth available for cross-SPE communication. Compared to parallel quick sort on a dual-core 3.2Ghz Intel Xeon processor, in-core bitonic sort using 16 SPEs can sort floats up to 10 times faster.

(*iii*) The relative advantage of bitonic sort on the Cell processor decreases as we go out-of-core and sort larger number of items. The sort becomes memory I/O bound due to smaller bandwidth available for accessing the main memory, compared to cross-SPE bandwidth. Yet, 16 SPEs can sort 0.5GB of floats up to 4 times faster compared to parallel quick sort on a dual-core 3.2Ghz Intel Xeon processor.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the Cell processor. Section 3 describes the basics of sorting on Cell. Sections 4.1, 4.2, and 4.3 describe local, in-core, and out-of-core bitonic sort, respectively. Section 5 gives the communication and computation complexity of out-of-core bitonic sort. Experimental results are presented in Section 6. Section 7 presents the related work and we conclude the paper in Section 8.

2. CELL PROCESSOR

The Cell processor is a 64-bit single-chip multiprocessor with 9 processing elements − 1 general purpose processing element called PPE and 8 special purpose co-processors called SPEs. Each SPE is a 128-bit RISC processor specialized for data-rich, compute-intensive applications. Since SPEs are pure SIMD processors, high application performance is strongly tied to heavy use of vectorized operations.

Each SPE has full access to coherent shared memory. However, an important difference between the SPEs and the PPE is *how* they access the main memory. Unlike the PPE, which is connected to the main memory through two level of caches, SPEs access the main memory with direct memory access (DMA) commands. Instead of caches, each SPE has a 256KB private local store. The local stores are used to hold both instructions and data. The load and store instructions on the SPE go between the register file and the local store. Transfers between the main memory and the local store, as well as the transfers between different local stores, are performed through *asynchronous* DMA transfers. This is a radical design compared to conventional architectures and programming models, because it explicitly parallelizes the computation and transfer of data. On the down side, it is programmers' task to manage such transfers and take advantage of the high aggregate bandwidth made available by the Cell architecture, especially for cross-SPE transfers [15].

Besides data parallelism from rich set of SIMD instructions, SPEs also provide instruction-level parallelism in the form of dual pipelines. Reducing the dependencies among instructions within sequential code blocks can improve application performance by utilizing these dual-pipelines.

Another important architectural constraint is the lack of branch prediction hardware on the SPEs. Hence, SPE applications should avoid using conditionals as much as possible to keep the pipeline utilization high. Figure 1 gives a basic view of the Cell processor.

3. SORTING ON CELL: THE BASICS

Any sorting algorithm designed to run on the Cell processor using multiple SPEs needs to satisfy the following constraints of the Cell architecture: (*i*) distributed programming model, (*ii*) limited local memory per SPE, (*iii*) higher bandwidth for transfers across local memories, compared to transfers to/from the main memory, and (*iv*) data-parallelism via SIMD instructions.

Therefore, sorting on Cell calls for a three-tiered approach. Figure 2 presents an outline of the complete algorithm. At the innermost first tier, one needs a sorting kernel that is effective for sorting items that are local to an SPE. We refer to this as the *single-SPE local sort*. Since the Cell processor provides higher bandwidth for cross-SPE memory transfers compared to the bandwidth available for main memory access, the second tier is optimized for distributed sorting of items that are stored in the local stores of the participating SPEs. This is called *distributed in-core sort*. All of the data transfers during an in-core sort are between the SPEs or within an SPE. The term "in-core" refers to the set of SPEs connected by the EIB bus (see Figure 1). When the set of items to be sorted does not fit into the collective space provided by the local stores of the SPEs, the sort has to be taken "out-of-core", which involves moving data back-and-forth between the local stores and the main memory. We

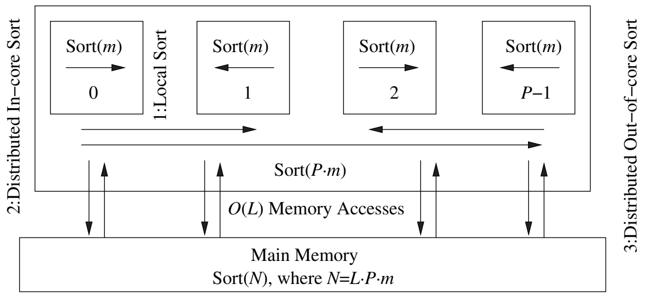

Figure 2: Three-tiered distributed sort on Cell, using bitonic merge.

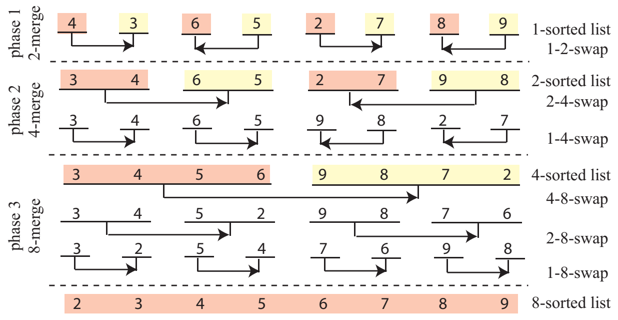

Figure 3: Bitonic sort of 8 values

call this *distributed out-of-core sort*. The distributed out-of-core sort makes use of the distributed in-core sort, which in turn makes use of the local sort. CellSort is based on distributed bitonic merge with a SIMDized bitonic sorting kernel. The distributed bitonic merge is used both for the in-core and out-of-core tiers and is itself SIMDized.

Let N be the total number of items to be sorted using P processors. Let m denote the size of items that each processor can sort in its local memory using the local sorting kernel. Hence, $P \cdot m$ is the maximum number of items that can be sorted in-core. Finally, $L = \frac{N}{(P \cdot m)}$ determines the number of memory runs for an out-of-core sort. Let $C_{\mathcal{U}}(m)$ be the computational cost of sorting m items using a sorting kernel $\mathcal{U}$. Then the total cost of sorting N items over P processors using the sorting kernel $\mathcal{U}$ and a distributed bitonic merge is $(L \cdot P) \cdot C_{\mathcal{U}}(m) + \mathcal{O}(\frac{N}{P} \cdot \lg N \cdot \lg (L \cdot P))$. In this paper, we experimentally demonstrate that the proposed algorithm scales very well as the number of processors is increased. Our implementation assumes that N, m, and P are powers of 2.

4. SORTING ON CELL: THE ALGORITHMS

In this section, we describe basic bitonic sort, local sort with SIMDized bitonic kernel, distributed in-core sort, and distributed out-of-core sort, respectively.

4.1 Single-SPE Local Sort

Local sort is one of the most performance critical parts of the full-scale out-of-core sort on the Cell processor. Different sorting kernels can be used to perform the local sorts. The choice of the sorting kernel should take into account the properties of the SPEs. Sorting algorithms with heavy branching, recursion, and non-contiguous memory access patterns should be avoided. The latter is especially important, since SIMD acceleration can be effectively leveraged only if the sorting algorithm performs same operations on items within a contiguous memory block successively. Despite its non-optimal asymptotic complexity, bitonic sort lends itself to an efficient implementation on the SPE, by virtue of its many desirable properties, meeting all the requirements we have listed so far. We will first describe a basic version of the bitonic sort and then give details on how bitonic sort can be optimized using SIMD instructions.

4.1.1 Power-of-two Bitonic Sort

Here we will describe bitonic sort from a procedural point of view. The theoretical basis of bitonic sort can be found elsewhere [3]. We assume that the number of items to be sorted, m, is an exact power of 2.

Bitonic sort makes use of a key procedure called *bitonic merge*. Given two equal length lists of items, sorted in opposing directions, the bitonic merge procedure will create a combined list of sorted items. Bitonic sort makes use of successive bitonic merges to fully sort a given list of items. Concretely, bitonic sort involves $\lg m$ phases, where each phase consists of a series of bitonic merge procedures. The first phase merges each even-indexed item (index starts from 0) with the item immediately following it, in alternating directions. This produces a list where consecutive 2-item blocks are sorted, in alternating directions. Similarly, the second phase of bitonic sort involves merging each even-indexed 2-item block with the 2-item block immediately following it, producing a list where consecutive 4-item blocks are sorted in alternating directions. The sort continuous in this manner until the list of items is fully sorted in ascending order after the $\lg m$ th phase. Following the ith phase ($i \in [1..\lg m]$), we have a list where consecutive k-item blocks, where $k = 2^i$, are sorted. All even indexed k-item blocks are sorted in ascending order, whereas all odd indexed ones are sorted in descending order. We call this list a *k-sorted* list. We refer to the ith phase of bitonic sort as the *k-merge* phase, where $k = 2^i$ and a k-sorted list is generated.

Illustration of k-merge phases: Figure 3 gives an illustration of bitonic sort for $m = 8$. For now we will only focus on the status of the 8-item list after the k-merge phases (lists below dashed horizontal phase separators). The dark shades in the figure represent the blocks sorted in ascending order, whereas light shades represent blocks sorted in descending order. Note that there are $\lg m = 3$ phases of the sort, namely a 2-merge phase to yield a 2-sorted list, a 4-merge phase to yield a 4-sorted list, and an 8-merge phase to yield the final sorted list.

We now describe the details of k-merge phases. A k-merge phase involves $\lg k$ number of steps that perform *compare-and-swap* operations. We first describe these steps in the context of the first k-items, i.e. a bitonic merge of two sorted (in opposite directions) lists of size $k/2$ each. The first step involves performing compare-and-swaps of consecutive $k/2$-item blocks. The second sub-step involves performing compare-and-swaps of consecutive $k/4$-item blocks, and so on. In summary, the ith step involves performing compare-and-swaps of consecutive j-item blocks, where $j = k/2^i$. Note that we have $j = 1$ for the last step ($i = \lg k$) and thus perform compare-and-swap of each even-indexed item with the item immediately following it. Compare-and-swap of

Algorithm 1: Power-of-two bitonic sort. Note: **bw-xor** and **bw-and** represent bitwise-and and bitwise-xor

```
BITONIC_SORT($data$, $m$)
(1)  for $k = 2$ to $k \leq m$ step $k \leftarrow 2 \cdot k$ {$k$-merge phases}
(2)      for $j = k/2$ to $j > 0$ step $j \leftarrow j/2$ {$j$-$k$-swap steps}
(3)          for $i = 0$ to $i < m$ step $i \leftarrow i+1$ {compare-and-swaps}
(4)              $l \leftarrow i$ bw-xor $j$ {index of item from odd $j$-block}
(5)              if $l < i$ then $i = i + j - 1$ {skip odd $j$-block}
(6)              else {perform compare-and-swap of items}
(7)                  if ($i$ bw-and $k = 0$) and ($data[i] > data[l]$)
(8)                      SWAP($data[i]$, $data[l]$) {even $k$-block}
(9)                  if ($i$ bw-and $k \neq 0$) and ($data[i] < data[l]$)
(10)                     SWAP($data[i]$, $data[l]$) {odd $k$-block}
```

two j-item blocks involve comparing individual items from the first block with their corresponding items in the second block and performing a swap if the item from the first block is larger than the item from the second block. For instance, performing compare-and-swap on the lists $A1 = [6, 5, 7, 1]$ and $A2 = [3, 9, 2, 4]$ yields $A1 = [3, 5, 2, 1]$ and $A2 = [6, 9, 7, 4]$.

Now, consider the ith step of the k-merge in the context of the complete list (not for the first k-items). The same procedures described for the first k-items are repeated for every k-item block, but with alternating compare-and-swap directions to result in alternating orders. That is, we perform compare-and-swap of consecutive j-item blocks ($j = k/2^i$) for the whole list, but alternate the compare-and-swap direction after every k-item block. In other words, while performing the j-item compare-and-swaps within an even indexed k-block we perform swaps iff the items from the first j-blocks are larger, whereas within an odd indexed k-item block we perform swaps iff the items from the first j-blocks are smaller. We refer to the ith step of the k-merge as the j-k-*swap* step where $j = k/2^i$.

Illustration of j-k-swap sub-steps: Let us go back to Figure 3 and examine the j-k-swap steps of the k-merge phases. The arrows in the figure represent the compare-and-swaps performed between j-item blocks, including the direction of the compare-and-swap. Note that the compare-and-swaps performed as part of the k-j-swap steps for a k-merge phase have a single direction within each k-item block, which alternates after each k-item block. For instance, during the 4-merge phase (phase 2 in the figure) all compare-and-swaps performed within the first 4-item block are ascending, whereas they are descending for the second 4-item block.

When m is a power of 2, bitonic sort lends itself to a very straight-forward non-recursive implementation based on the above description. We provide the pseudo code of bitonic sort in Algorithm 1. Note that, bitonic sort is an in-place sort. Its computational complexity is easy to derive, since the number of operations performed is independent of the values of the sorted items. As a result, bitonic sort has equal worst, best, and average case asymptotic complexities. For m items, bitonic sort involves $\lg_2 m$ phases. The ith phase, which is a k-merge ($k = 2^i$), involves $i = \lg k$ steps. Each step is a j-k-swap with $\Theta(m)$ complexity. As a result, the complexity of bitonic sort is $(\sum_{i=1}^{\lg m} i) \cdot \Theta(m) = \Theta(m \cdot \lg^2 m)$.

Even though basic bitonic sort is simple to understand and implement, its out-of-core and in-core distributed variants designed for the Cell processor are significantly more complex and harder to implement. Even the single-SPE, local implementation of bitonic sort is non-trivial due to the intricacies of an efficient SIMD implementation.

4.1.2 SIMDized Bitonic Sorting Kernel

We now describe how bitonic sort can be accelerated through the use of SIMD instructions available on the SPEs. We will assume that the sorted items (keys) are 32-bit integers or floats, so that each 128-bit vector contains 4 items. Items in the form of key/value pairs can be sorted by separating keys from values, attaching value pointers to keys, and then sorting the key/pointer pairs.

The SIMD support can be used to perform the compare-and-swaps using fewer number of instructions. This is especially effective when the blocks on which we perform compare-and-swaps consist of at least one vector, that is we have $j \geq 4$ during a j-k swap step of the sort. Let a and b be two corresponding vectors from j-item blocks to be compare-and-swapped. An SIMD comparison instruction $t = \mathit{cmpgt}(a, b)$ can be used to create a mask such that a following SIMD select instruction $a' = \mathit{select}(a, b, t)$ can be used to yield the smaller of the corresponding items of the two vectors, that is the lower half of the compare-and-swap result. Similarly, $b' = \mathit{select}(b, a, t)$ will yield the larger items, that is the higher half of the compare-and-swap result. In total, the SIMD implementation requires 1 comparison and 2 select instructions to complete the compare-and-swap of two vectors that contain a total of 8 items. Another important observation is that, the SIMD implementation of the compare-and-swap does not involve any conditional statements. This is a significant advantage, considering that the SPEs do not have branch prediction hardware.

When we have a j-k-swap step with $j < 4$ ($j \in \{1, 2\}$), the exploitation of SIMD instructions is not as straightforward compared to the case of $j \geq 4$ we have described so far. This is because the blocks to be compare-and-swapped fall into the boundaries of a single vector. It is important to note that all k-merge phases ($\lg m$ of them) of a bitonic sort involve a j-k-swap step with $j = 1$ and half of the k-merge phases involve a j-k-swap step with $j = 2$. In fact $3/(\lg m + 1)$ fraction of the j-k-swaps are for $j < 4$. Even though this fraction approaches to zero as m becomes larger, it is still a large fraction for practical purposes. Given that local sort can handle at most 32K number of items (128KB of data) due to the limited local store space on SPEs, the fraction of j-k-swap steps with $j < 4$ constitute at least 18.75% of the total. Therefore, it is important to optimize for the case of $j < 4$, which includes five sub-cases, namely: $\langle j=2, k=4\rangle$, $\langle j=2, k\geq 8\rangle$, $\langle j=1, k=2\rangle$, $\langle j=1, k=4\rangle$, and $\langle j=1, k\geq 8\rangle$. Our experimental results show that optimizing these sub-cases has a profound impact on performance (see Section 6.1).

Even though each sub-case require a different SIMD implementation, their SIMDization share a common structure. Given two vectors covering consecutive items, say a and b, we first perform two *shuffles* to organize these items into two new vectors a' and b' such that the pair of items to be compare-and-swapped are located at the corresponding positions of the two new vectors. Then the compare-and-swap of paired items are performed using 1 compare and 2 select instructions as before. Another two shuffles are applied to bring the items back to their proper positions. The two shuffles performed at the beginning and at the end match each other for the cases of $\langle j=2, k\geq 8\rangle$ and $\langle j=1, k\geq 8\rangle$,

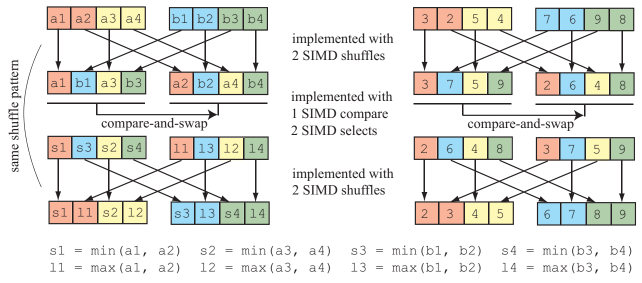
same shuffle pattern
implemented with 2 SIMD shuffles
implemented with 1 SIMD compare 2 SIMD selects
implemented with 2 SIMD shuffles
compare-and-swap
compare-and-swap
s1 = min(a1, a2) s2 = min(a3, a4) s3 = min(b1, b2) s4 = min(b3, b4)
l1 = max(a1, a2) l2 = max(a3, a4) l3 = max(b1, b2) l4 = max(b3, b4)

Figure 4: SIMDization of j-k swap, when $j = 1$, $k \geq 8$. On the right: 1-8-swap step from Figure 3

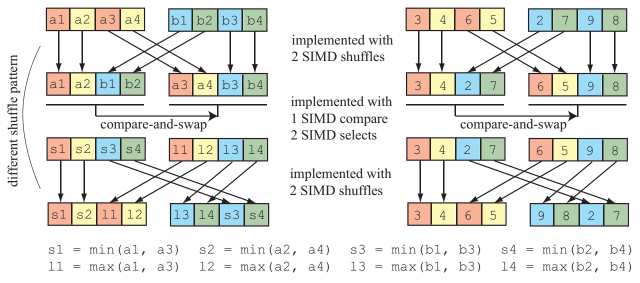
different shuffle pattern
implemented with 2 SIMD shuffles
implemented with 1 SIMD compare 2 SIMD selects
implemented with 2 SIMD shuffles
compare-and-swap
compare-and-swap
s1 = min(a1, a3) s2 = min(a2, a4) s3 = min(b1, b3) s4 = min(b2, b4)
l1 = max(a1, a3) l2 = max(a2, a4) l3 = max(b1, b3) l4 = max(b2, b4)

Figure 5: SIMDization of j-k swap, when $j = 2$, $k = 4$. On the right: 2-4-swap step from Figure 3

whereas they do not match for the cases of $\langle j = 2, k = 4\rangle$, $\langle j = 1, k = 2\rangle$, and $\langle j = 1, k = 4\rangle$. The latter is due to the fact that the compare-and-swap direction changes within a 2-vector block ($k \leq 4$).

We now give two examples of SIMDization for $j < 4$, concretely for the sub-cases of $\langle j = 1, k \geq 8\rangle$ and $\langle j = 2, k = 4\rangle$. Figure 4 illustrates the former. Since $j = 1$, we need to compare-and-swap consecutive items. This requires distributing each 2-item block across two vectors. The first row in Figure 4 shows the shuffles needed to achieve this. Different shades are used in the figure to mark different pair of items to be compare-and-swapped. Note that after the first set of shuffles, the two vectors has the exact same shading order. At this stage, we apply a vector compare-and-swap over the shuffled vectors (see the second row in the figure). Then the exact same shuffles are re-applied to bring back together the separated 2-item blocks (see the third row in the figure). In total 7 SIMD instructions are used, 4 more than the number of instructions needed for the case of $j \geq 4$.

Figure 5 illustrates the sub-case of $\langle j = 2, k = 4\rangle$. Since $j = 2$, we need to compare-and-swap consecutive 2-item blocks. This requires distributing each 4-item block across two vectors by locating the first 2-item block to the first vector and the second 2-item block to the corresponding positions of the second vector. The first row in Figure 5 shows the shuffles needed to achieve this. We then apply a vector compare-and-swap over the shuffled vectors, and then a new set of shuffles are applied to bring back together the separated 4-item blocks. There is a change in the shuffles performed, because the compare-and-swap direction is reversed for the second 4-item block.

Besides SIMDization, implementing bitonic sort efficiently on the SPEs also require unrolling loops and avoiding branches as much as possible. These optimizations provide smaller percentage of dependency stalls and branch misses, better utilization of the dual-pipelines, and higher CPI (cycles per instruction) in general.

4.2 Distributed In-core Sort

Distributed in-core sort works by performing local sorts using the bitonic sorting kernel and then using in-core bitonic merge to yield the final result. Algorithm 2 gives the pseudo code of the distributed in-core sort. In this subsection we are going to assume that $L = 1$, thus we have $N = P \cdot m$.

Concretely, distributed in-core sort involves two stages. In the first stage, all SPEs perform local sorts. The P local sorts are performed in parallel. SPE i, $i \in [0..P-1]$, sorts the items in ascending order if i is even, and in descending order if i is odd. After the first stage, we have a k-sorted list (in the local stores) where $k = m$. The second stage involves performing $\lg P$ number of k-merge phases, the first one for $k = 2 \cdot m$, the last one for $k = P \cdot m = N$, and doubling k after each phase. Before each k-merge phase, the set of consecutive k/m SPEs do a barrier synchronization between themselves to make sure the results from the last phase are finalized. Some of the j-k-swap steps of the $\lg P$ number of k-merge phases can be performed locally (see line 25 in Algorithm 2 for the local), whereas others require cross-SPE data transfers (see lines 5-24 in Algorithm 2).

Distributed cross-SPE j-k swaps

Recall that a k-merge phase involves $\lg k$ number of j-k swap steps. For $k \geq 2 \cdot m$, the first $\lg(k/m)$ number of j-k swap steps ($j \geq m$) involve cross-SPE data transfers, whereas the rest can be performed locally by each SPE. For the cross-SPE j-k swaps to be efficient, we need to minimize the amount of data transfers performed between the SPEs. In what follows, we provide a way to partition the cross-SPE j-k-swap computation across the P SPEs such that each SPE performs a compare-and-swap using $m/2$ local items and $m/2$ remote items, which results in minimal amount of data transfers.

To perform a cross-SPE j-k-swap we first group the SPEs into $N/(2 \cdot j)$ number of groups, where consecutive $(2 \cdot j)/m$ SPEs are put into the same groups. The SPEs within the same group, say group i (i starts from 0), perform the compare-and-swap of the j-item block indexed $2 \cdot i$ with its consecutive j-item block. SPE l will either compare-and-swap the first half of its items with the first half of the items of SPE $(l + j/m)$ or it will compare-and-swap the second half of its items with the second half of the items of SPE $(l - j/m)$. The former is performed if $\lfloor (l \cdot m)/j \rfloor$ is even, and the latter is performed otherwise. After each j-k-swap, the SPEs within the same groups do a barrier synchronization among themselves. This whole procedure is best illustrated with an example.

Figure 6 illustrates a cross-SPE j-k swap, when $j = 2 \cdot m$, $k = 4 \cdot m$, and $P = 8$. As a result, we have $N/(2 \cdot j) = 2$ SPE groups, each containing $(2 \cdot j)/m = 4$ SPEs. The first 4 SPEs perform the compare-and-swaps of the first (indexed 0) $j = 2 \cdot m$-item block with the second j-item block. The second 4 SPEs perform the compare-and-swaps of the third j-item block with the fourth j-item block. For each SPE, the figure marks the local items that are not involved in a cross-SPE transfer (lightly shaded) and the local items that are involved in a cross-SPE transfer (darkly shaded).

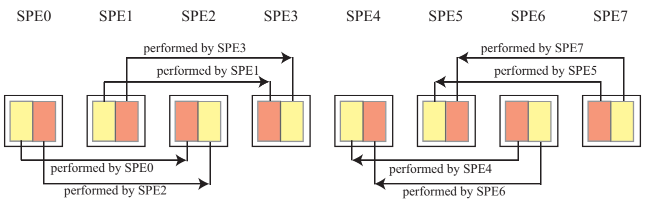

Figure 6: Cross-SPE j-k swap, when $j = 2 \cdot m$, $k = 4 \cdot m, P = 8$

Note that, for each SPE exactly half of the items in its local store are not involved in a cross-SPE transfer, which is the optimal behavior. As an example consider SPE 2 ($l = 2$). Since $(l \cdot m)/j = 1$ is odd, SPE 2 compare-and-swaps its second half of items with the second half of items from SPE 0 ($l - j/m = 0$).

Hiding the cross-SPE transfer delays

Recall that during a cross-SPE compare-and-swap, each SPE has to fetch $m/2$ number of items from a remote SPE, perform compare-and-swap of the remote items with their local counterparts, and put one half of the result back ($m/2$ items) into the remote SPE. There are two important issues in implementing this efficiently:

1. The local store of the SPE does not have enough space to hold all of the $m/2$ items, thus the operation has to be carried out in smaller batches of m_d items.

2. Asynchronous DMAs can be used to hide the transfer delays by overlapping compare-and-swap processing with cross-SPE data transfers.

This is implemented as follows. Let us assume the SPE has already brought in the ith remote m_d-item block. Before processing it, the SPE issues asynchronous DMA commands to (i) put the results from the $(i-1)$th m_d-item block back to the remote SPE and (ii) fetch the $(i+1)$th m_d-item block from the remote SPE . When the processing of the ith m_d-item block is finished, the SPE will check the completion status of the issued DMAs and wait for their completion in case they are still pending. Most of the time this involves no waiting. This kind of double-buffering is very effective in hiding memory transfer delays, because the Cell processor provides high cross-SPE interconnect bandwidth and thus the in-core sort does not become memory I/O bound.

4.3 Distributed Out-of-core Sort

Distributed out-of-core sort works by performing a series of in-core sorts and then using an out-of-core bitonic merge to yield the final result. Algorithm 3 gives the pseudo code of the distributed out-of-core sort.

Concretely, out-of-core sort involves two stages. In the first stage (see lines 1-7 in Algorithm 3), we perform L in-core sorts for each consecutive $(P \cdot m)$-item block, using alternating sort directions. These L in-core sorts are performed sequentially, where each in-core sort uses all P SPEs in parallel as described in the last subsection. After the first stage, we have a k-sorted list (in the memory) where $k = P \cdot m$. The second stage involves performing $\lg L$ number of k-merge phases, the first one for $k = 2 \cdot P \cdot m$, the last one for $k = L \cdot P \cdot m = N$, and doubling k after each phase.

Algorithm 2: Distributed in-core sort with bitonic merge. Note: **bw-and** represents bitwise-and operation.

```
IN_CORE_SORT($m$, $P$)
(1)  $s \leftarrow$ my SPE index {index starts from 0}
(2)  LOCAL_SORT($s$) {apply sorting kernel (sort in proper
         direction: desc. if $s$ is odd, asc. if $s$ is even) }
(3)  for $k \leftarrow 2 \cdot m$ to $k \leq m \cdot P$ step $k \leftarrow 2 \cdot k$
(4)      BARRIER($k/m$) {barrier($n$) means synchronize
             SPEs $[i..(i+n)]$, $i = n \cdot \lfloor s/n \rfloor$ }
(5)      for $j \leftarrow k/2$ to $j \geq m$ step $j \leftarrow j/2$
(6)          $i_s \leftarrow s \cdot m$ {index of the 1st item to be CASed}
(7)          if ($s$ bw-and $j/m$) $\neq 0$
(8)              $i_s \leftarrow i_s + m/2 - j$ {$i_s$ is remote}
(9)          $i_e \leftarrow i_s + m/2$ {1+ index of last CASed item}
(10)         if ($i_s$ bw-and $k$) = 0 then $d \leftarrow$ ‘ascending’
(11)                                  else $d \leftarrow$ ‘descending’
(12)         for $i \leftarrow i_s$ to $i < i_e$ step $i \leftarrow i + m_d$
(13)             $u \leftarrow i - s \cdot m$ {index at local store}
(14)             if $u < m/2$ {using fist half of local items}
(15)                 $B_1$ : $local\_data[u..(u + m_d - 1)]$
(16)                 $B_2 \leftarrow$ DMA read from SPE no $(i+j)/m$ its
                         $local\_data[u..(u + m_d - 1)]$
(17)                 COMPARE-AND-SWAP($B_1$, $B_2$, $d$)
(18)                 DMA write $B_2$ into SPE no $(i+j)/m$ at its
                         $local\_data[u..(u + m_d - 1)]$
(19)             else {using second half of local items}
(20)                 $B_1 \leftarrow$ DMA read from SPE no $i/m$ its
                         $local\_data[u..(u + m_d - 1)]$
(21)                 $B_2$ : $local\_data[u..(u + m_d - 1)]$
(22)                 COMPARE-AND-SWAP($B_1$, $B_2$, $d$)
(23)                 DMA write $B_1$ into SPE no $i/m$ at its
                         $local\_data[u..(u + m_d - 1)]$
(24)         BARRIER($2 \cdot (j/m)$)
(25)     LOCAL_MERGE() {apply local merge (merge in proper direction: desc. if $\lfloor s \cdot m/k \rfloor$ is odd, asc. otherwise)}
```

Before each k-merge phase, the SPEs do a barrier synchronization between themselves to make sure that the results from the last phase are finalized. The j-k-swap steps of the $\lg L$ number of k-merge steps cannot all be performed using in-core bitonic merge and as a result some of them require out-of-core data transfers (see lines 10-26 in Algorithm 3).

Distributed out-of-core j-k swaps

A k-merge phase involves $\lg k$ number of j-k swap steps. For $k \geq 2 \cdot P \cdot m$, the first $\lg(k/(P \cdot m))$ number of j-k swap steps ($j \geq P \cdot m$) involve out-of-core memory transfers, whereas the rest can be performed in an in-core manner. In what follows, we provide a way to partition the out-of-core j-k-swap computation across the P SPEs, such that no synchronization is needed after each out-of-core j-k swap. This is unlike the in-core j-k swaps discussed earlier, which were optimized to minimize the cross-SPE data transfers and thus required more complex synchronization patterns.

To perform an out-of-core j-k-swap, each SPE operates independently. m-item blocks are assigned to SPEs in a round-robin fashion. For instance, SPE l will compare-and-swap the m-item block with index i against the m-item block with index $i + j/m$, iff $i \mod P = l$ and $\lfloor i \cdot m/j \rfloor$ is even. If $\lfloor i \cdot m/k \rfloor$ is even, then the compare-and-swap is performed in ascending order. Otherwise, it is performed in descending order. The importance of this communication pattern is that, no synchronization is needed between the k-j-swap steps, since the SPEs are assigned fixed exclusive portions of the data during j-k-swap steps of a k-merge phase. This is best illustrated with an example.

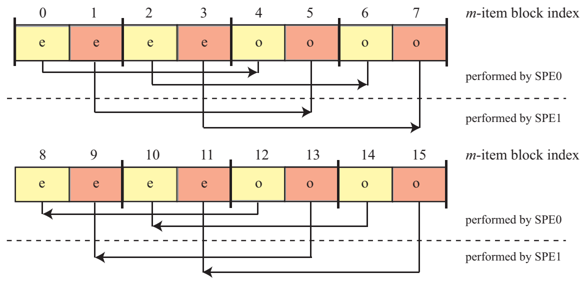

Figure 7: Out-of-core j-k swap, $j=4\cdot m, k=8\cdot m, P=2, L=8$

Figure 7 illustrates an out-of-core j-k swap, when $j = 4\cdot m$, $k = 8\cdot m$, $P = 2$, and $L = 8$. The m-item blocks processed by SPE 0 are lightly shaded, whereas the ones processed by SPE 1 are darkly shaded in the figure. An m-item block with index i is marked with 'e' iff $\lfloor i\cdot m/j\rfloor = \lfloor i/4\rfloor$ is even, and with 'o' otherwise. Note that SPE 0 compare-and-swaps m-item blocks 0, 2, 8, and 10 against m-item blocks 4, 6, 12, and 14, respectively. The blocks in the former list are all marked 'e' and satisfy $i \bmod P = l$ for $P = 2$ and $l = 0$, that is $i \bmod 2 = 0$. The blocks in the second list are $j/m = 4$ positions further in the order than their counterparts in the first list (m-item block with index i is compared against the m-item block with index $i + j/m$). The first two m-item block compare-and-swaps performed by SPE 0, that is $\langle 0 \rightarrow 4\rangle$ and $\langle 2 \rightarrow 6\rangle$ are performed using ascending order, since $\lfloor i\cdot m/k\rfloor = \lfloor i/8\rfloor$ is even for $i = 0$ and $i = 2$. The second two m-item block compare-and-swaps performed by SPE 0, that is $\langle 8 \leftarrow 12\rangle$ and $\langle 10 \leftarrow 14\rangle$ are performed using descending order, since $\lfloor i/8\rfloor$ is odd for $i = 8$ and $i = 10$.

Hiding the memory transfer delays

Similar to the case of in-core sort, the memory transfers during j-k-swap steps are done in chunks of m_d items each and double buffering is applied to hide the memory transfer delays. However, there are two significant differences.

1. None of the two m_d-item blocks that are to be compare-and-swapped are residents of the local store of the SPE performing the operation. Thus the overall communication cost in terms of number of items transferred is doubled compared to that of an in-core implementation. Recall that for an in-core sort, one of the m_d-item blocks is always local to the SPE.

2. All DMA transfers are to/from the main memory. Since the bandwidth to the main memory is smaller than the aggregate bandwidth between the SPE local stores, the transfer delays are larger for the out-of-core sort. As we will later discuss in the experimental results, out-of-core sort becomes memory I/O bound.

Distributed in-core j-k swaps

When we have $j < P\cdot m$, the j-k-swaps can be performed in an in-core manner. These j-k-swaps are collectively performed through L number of in-core merges, one for each $(P\cdot m)$-item block (see lines 27-32 in Algorithm 3). An in-core merge is similar to an in-core sort, in the sense that it includes cross-SPE merges and local merges. We do not further discuss in-core merges.

Algorithm 3: Distributed out-of-core sort with bitonic merge. Note: **bw-xor** and **bw-and** represent bitwise-xor and bitwise-and, respectively.

```
OUT_OF_CORE_SORT($m$, $P$, $L$)
(1)  $s \leftarrow$ my SPE index {index starts from 0}
(2)  for $i \leftarrow 0$ to $i < L$ step $i \leftarrow i+1$
(3)      $u \leftarrow (s + i\cdot P)\cdot m$
(4)      DMA read from memory $data[u..(u+m-1)]$
                                   to $local\_data[0..(m-1)]$
(5)      BARRIER($P$) {sync. before starting in-core sort}
(6)      IN_CORE_SORT($i$) {sort $i$th $P\cdot m$-item block (sort in proper
         direction: desc. if $i$ is odd, asc. if $i$ is even)}
(7)      DMA write to memory $data[u..(u+m-1)]$
                                   from $local\_data[0..(m-1)]$
(8)  for $k \leftarrow 2\cdot P\cdot m$ to $k \le L\cdot P\cdot m$ step $k \leftarrow 2\cdot k$
(9)      BARRIER($P$) {sync. before starting $k$-merge}
(10)     for $j \leftarrow k/2$ to $j \ge P\cdot m$ step $j \leftarrow j/2$
(11)         $d \leftarrow$ 'ascending' {initial direction}
(12)         for $i \leftarrow s\cdot m$ to $i < L\cdot P\cdot m$
(13)             $l \leftarrow i$ bw-xor $j${index of 1st item in pair $j$-block}
(14)             if $l < i$ {item $i$ is in odd indexed $j$-block}
(15)                 $i \leftarrow i + j$ {skip this $j$-block }
(16)                 if ($i$ bw-and $k$) = 0 then $d \leftarrow$ 'ascending'
(17)                                   else $d \leftarrow$ 'descending'
(18)                 continue{go to next $j$-block}
(19)             $i_s \leftarrow i$; $i_e \leftarrow i_s + m$
(20)             for $i \leftarrow i_s$ to $i < i_e$ step $i \leftarrow i + m_d$
(21)                 $B_1 \leftarrow$ DMA read from mem. $data[i..(i+m_d-1)]$
(22)                 $B_2 \leftarrow$ DMA read from memory
                                   $data[(i+j)..(i+j+m_d-1)]$
(23)                 COMPARE-AND-SWAP($B_1$, $B_2$, $d$)
(24)                 DMA write $B_1$ into mem. $data[i..(i+m_d-1)]$
(25)                 DMA write $B_2$ into memory
                                   $data[(i+j)..(i+j+m_d-1)]$
(26)             $i \leftarrow i_s + m\cdot P$ {index of the 1st item in the next
                                   $m$-block assigned to $s$}
(27)     for $i \leftarrow 0$ to $i < L$ step $i \leftarrow i+1$
(28)         $u \leftarrow (s + i\cdot P)\cdot m$
(29)         DMA read from memory $data[u..(u+m-1)]$
                                   to $local\_data[0..(m-1)]$
(30)         BARRIER($P$) {sync. before starting in-core merge}
(31)         IN_CORE_MERGE($i$) {apply in-core merge to the $i$th $(P\cdot$
             $m)$-item block (merge in proper direction: desc. if $\lfloor i\cdot$
             $P\cdot m/k\rfloor$ is odd, asc. otherwise)}
(32)         DMA write to memory $data[u..(u+m-1)]$
                                   from $local\_data[0..(m-1)]$
```

5. COMPLEXITY ANALYSIS

In this section, we present the computation and communication complexity of the complete out-of-core bitonic sort on the Cell processor.

Multi-SPE out-of-core sort is also fully and symmetrically parallelized. The resulting computational complexity of the sort using P SPEs is given by $\mathcal{O}(\frac{N}{P}\cdot \lg^2 N)$. We have $N = L\cdot P\cdot m$. However, deriving the communication complexity of out-of-core sort is slightly more involved. We divide the communication complexity into two categories, namely *i-transfers* and *o-transfers*. The number of i-transfers is the number of items sent/received to/from the local store of an another SPE. The number of o-transfers is the number of items sent/received to/from the main memory.

The first stage of the out-of-core sort involves L number of in-core sorts of N/L items each. During an in-core sort, $\lg P\cdot(1+\lg P)/2$ number of j-k-swaps are performed, where each j-k-swap makes N/L i-transfers. This brings the total number of i-transfers to $(N/L)\cdot((1+\lg P)\cdot\lg P)/2$. Moreover, $2\cdot N$ o-transfers are made to bring in the items to local stores and to write them back to memory. In summary, the total number of i-transfers and o-transfers made for the first stage is $(N/2) \cdot (1 + \lg P) \cdot \lg P$ and $2 \cdot N$, respectively.

The second stage of the out-of-core sort involves $\lg L \cdot (1 + \lg L)/2$ number of out-of-core j-k-swaps and a number of in-core j-k-swaps. During an out-of-core j-k-swap, $2 \cdot N$ o-transfers are made. Thus the total o-transfer cost of out-of-core j-k-swaps is $N \cdot \lg L \cdot (1+\lg L)$. The in-core j-k-swaps are performed as $L \cdot \lg L$ number of in-core merges, where an in-core merge involves makes $(N/L) \cdot \lg P$ i-transfers. As a result, the total i-transfer cost for the in-core j-k-swaps is $N \cdot \lg L \cdot \lg P$. Moreover, each in-core merge makes $2 \cdot (N/L)$ o-memory transfers to DMA in the items to local stores and to write them back. As a result, the total o-transfer cost for the in-core j-k-swaps is $2 \cdot N \cdot \lg L$. In summary, the total number of i-transfers and o-transfers made for the second stage is $N \cdot \lg L \cdot \lg P$ and $(3 + \lg L) \cdot N \cdot \lg L$, respectively.

In summary, out-of-core sort makes

- $(\lg L + (1 + \lg P)/2) \cdot N \cdot \lg P$ i-transfers, and
- $2 \cdot N + (3 + \lg L) \cdot N \cdot \lg L$ o-transfers.

For $L > P$, asymptotically the i-transfer and o-transfer complexities of out-of-core sort are $\mathcal{O}(N \cdot \lg L \cdot \lg P)$ and $\mathcal{O}(N \cdot \lg^2 L)$, respectively.

6. EXPERIMENTAL RESULTS

In this section we present experimental results comparing the performance of local, in-core, and out-of-core sort to various alternatives on various processors. In particular we compare the following configurations: (*i*) Distributed bitonic sort using up to $P = 16$ SPEs (3.2GHz each) available in an IBM QS20 Cell blade, with the following sorting kernels: basic bitonic, SIMDized bitonic, shell sort, and quick sort, (*ii*) Quick sort on the PPE (3.2GHz), (*iii*) Single and dual-core (using OpenMP) quick sort on 3.2GHz Intel Xeon and 3GHz Intel Pentium 4 machines, (*iv*) Single-core SSE-enhanced bitonic sort on aforementioned Intel machines. The sorts on the Intel processors are compiled using the `icc` compiler with all optimization on. The Cell sorts are compiled using the `gnu` tool chain. Maximum number of items that can be sorted using local sort is $m = 32$K (128KB of data) and using in-core sort is $N = P \cdot m = 16 \cdot 32\text{K} = 512$K (2MBs of data). We sort up to 0.5GB of data (128M number of items) using out-of-core sort, since the memory available to us in our test machine was 1GB. It is important to note that the sort time for in-core and local sorts also include the time to transfer items to/from the main memory, in order to be fair in comparison against main memory based algorithms.

6.1 Single-SPE Local Sort

We now present experimental results for single-SPE local sorts. We compare the sort time of SIMDized bitonic sort on SPE to basic bitonic sort on SPE, quick sort on SPE, and quick sort on PPE.

Table 1: Single-SPE local sort performance

32K	SIMD bitonic	quick	no-shuffle bitonic	basic bitonic	PPE quick
ints	.0025 secs	.0073	.0178	.0550	.0038
floats	.0025 secs	.0073	.0178	.0550	.0090

Table 1 lists the sort times in seconds for integer and float sorts using different algorithms, with 32K items. In this table, we also included a variant of bitonic sort called *no-shuffle* bitonic sort, which does not implement the SIMD shuffle-based optimizations designed for the j-k-swap steps with $j < 4$. We make four observations from Table 1. First, we observe that basic bitonic sort takes 22 times longer compared to the optimized SIMD bitonic sort. This attests to the importance of careful SIMDization, loop unrolling, and branch avoidance. Second, we observe that leaving out the optimizations for j-k-swap steps with $j < 4$ causes the sort time to increase to around 7.12 times the sort time of fully optimized SIMD bitonic sort. Recall from Section 4.1.2 that such j-k-swap steps constitute only 18.75% of the total number of j-k-swaps (for 32K items). However, implementing those steps with scalar operations results in a much higher cost, as evidenced by the six fold increase in the sort time. Third, we see that quick sort on the SPE is a better choice than the simple bitonic sort, but still takes approximately 3 times longer compared to the SIMDized bitonic sort. Note that quick sort has average case complexity of $\mathcal{O}(N \cdot \lg N)$, which is asymptotically smaller than $\mathcal{O}(N \cdot \lg^2 N)$ of bitonic sort. However, quick sort cannot be efficiently SIMDized. For the current local store sizes, the SIMDized bitonic sort prevails against the quick sort. Last, we observe that quick sort on the PPE comes closest to the SIMdized bitonic sort on the SPE, but still takes around 1.5 times longer when sorting integers. The performance of PPE quick sort drops when floats are sorted (it takes more than 2.3 times longer compared to sorting integers), whereas there is no difference in SPE performance when sorting integers or floats.

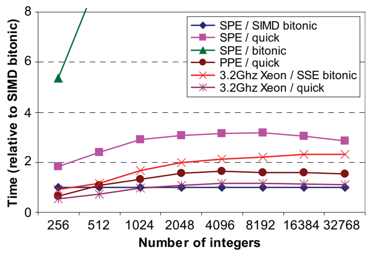

Figure 8: Local sort, integers

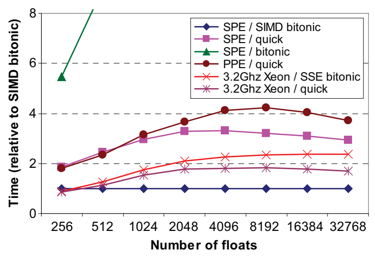

Figure 9: Local sort, floats

Figures 8 and 9 plot the sort time of different algorithms, relative to that of SIMD bitonic sort, with varying number of items (up to 32K items) for integers and floats, respectively. For this experiment, we also compare the performance against quick sort and SIMDized bitonic sort (using SSE and SSE2 instructions) on a 3.2Ghz Intel Xeon processor. We observe that, unlike SPEs, Xeon favors quick sort over SIMDized bitonic sort. In fact, SIMDized bitonic sort on Xeon is 2.4 times slower compared to SPE when sorting 128KB of items. On the other hand, quick sort on Xeon is around 1.7 times slower when sorting floats and only 1.1 times slower when sorting integers, both for 128KB of data. It is important to point out that the pure SIMD nature of the SPEs, their large 128-bit register file, rich set of SIMD instruction set, and their lack of branch prediction hardware result in SIMDized bitonic sort to prevail over quick sort, which is not the case for SSE enhanced bitonic sort on the Intel processors such as the Xeon.

Table 2: Cycle statistics

metrics	basic bitonic	SIMD bitonic	quick sort
CPI (cycles per instruction)	2.26	1.05	3.39
Single issued cycles	28.9%	42.5%	20.5%
Double issued cycles	3.6%	22.2%	2.6%
Stalls due to branch	40.1%	22%	40.3%
Stalls due to dependency	22.5%	10.8%	33.8%
Other (including nops)	3.7%	2.5%	2.8%

Table 2 lists the cycle statistics for basic and SIMDized bitonic sorts as well as the quick sort on the SPE. Note that the CPI is significantly lower with the optimized SIMD implementation (less than half of the CPI of basic bitonic sort and one third of the CPI of quick sort). Moreover the SIMDized implementation with heavy loop unrolling and branch avoidance result in smaller percentage of dependency and branch misses. Quick sort on the SPE has worse CPI and stall percentage than both basic and SIMDized bitonic sort, but as we have observed earlier it is faster than the basic bitonic sort, since it has a smaller total cycle count.

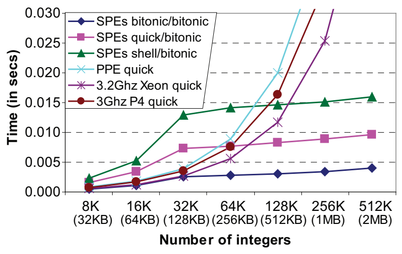

Figure 10: In-core sort, integers

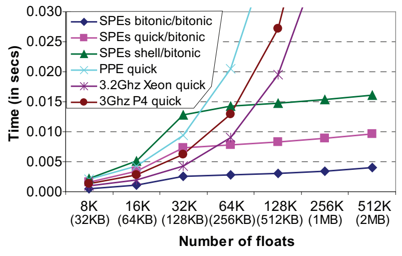

Figure 11: In-core sort, floats

6.2 Distributed In-core Sort

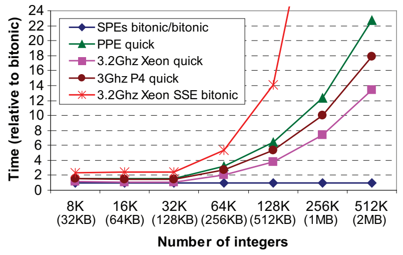

Figure 12: In-core sort, integers

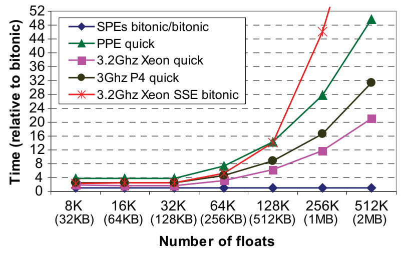

Figure 13: In-core sort, floats

We now present results from multi-SPE in-core sort. We compare 6 different approaches. The first three are SPE sorts using up to 16 SPEs. Based on the data size, 1, 2, 4, 8, or 16 SPEs are used. The difference between the first three approaches, named bitonic/bitonic, quick/bitonic, and shell/bitonic, is the sorting kernel employed. The fourth approach is quick sort on the PPE. The fifth and sixth approaches are quick sorts on two Intel processors, a 3.2Ghz Xeon and a 3Ghz Pentium 4. The last three approaches are single-threaded sorts using a single core.

Figures 10 and 11 plots the time it takes to perform the sort with the 6 different approaches, for integers and floats respectively. After the number of items hit 32K, the number of SPEs used is doubled as the number of items sorted is doubled. Time to sort using bitonic/bitonic only increases 63% when going from sorting 32K items with a single SPE up to sorting 512K items (16 times 32K) with 16 SPEs. This number is particularly important. Let us represent the time it takes to perform a j-k-swap during a local sort as $\alpha{\cdot}N$. Then the time it takes to perform the bitonic sort with $N = 32\text{K}$ is given by $A = (\alpha{\cdot}32\text{K}){\cdot}\lg 32\text{K}{\cdot}(\lg 32\text{K}{+}1)/2 = 3932160{\cdot}\alpha$. Now let us assume that in an ideal case we have no overhead due to cross-SPE communication when performing in-core bitonic sort. Then the time it takes to sort 512K items using 16 SPEs with a no overhead assumption is given by $B = (1/16){\cdot}(\alpha{\cdot}512\text{K}){\cdot}\lg 512\text{K}{\cdot}(\lg 512\text{K}{+}1)/2 = 6225920{\cdot}\alpha$. Note that B is only 58% higher than A. This shows that the communication overhead of the actual in-core bitonic sort is as little as 3% for 16 SPEs (we have $(163-158)/168 = 0.03$). This is due to the high cross-SPE interconnect bandwidth and the use of double buffering to hide the transfer delays.

Figures 10 and 11 also show that the bitonic sorting kernel is significantly faster than other alternatives. Bitonic sort with quick sort kernel takes 2.4 times and shell sort kernel takes 4 times the sorting time with bitonic kernel.

The performance of sorting on Cell compared to the Intel processors is better observed from the Figures 12 and 13, which plot the sorting time relative to that of bitonic sort with bitonic kernel. We observe that when sorting 2MB of items, the closest competitor to 16 SPE bitonic sort is the quick sort on the 3.2Ghz Xeon, which takes 13.5 times longer when sorting integers and 21 times longer when sorting floats. Quick sort on the 3Ghz Pentium 4 takes 18 times longer when sorting integers and 31 times longer when sorting floats. Quick sort on the PPE (clocked at 3.2Ghz) performs the worst, taking 23 times longer when sorting integers and 50 times longer when sorting floats.

6.3 Distributed Out-of-core Sort

We now present results from out-of-core sort. 16 SPE bitonic sort (bitonic merge with SIMDized bitonic kernel) is compared against quick sort on the PPE as well as on the Intel 3.2Ghz Xeon (including dual-core using OpenMP) and Intel 3.0Ghz Pentium 4 processors. We sort up to 128M items, which corresponds to 1/2GB of data (maximum available memory on the Cell machine was 1GB).

Table 3: Out-of-core sort performance (in secs)

# items	16 SPEs bitonic	3.2GHz Xeon quick	3.2GHz Xeon quick 2-core	PPE quick
1M	0.0098	0.1813	0.098589	0.4333
2M	0.0234	0.3794	0.205728	0.9072
4M	0.0569	0.7941	0.429499	1.9574
8M	0.1372	1.6704	0.895168	4.0746
16M	0.3172	3.4673	1.863354	8.4577
32M	0.7461	7.1751	3.863495	18.3882
64M	1.7703	14.8731	7.946356	38.7473
128M	4.0991	30.0481	16.165578	79.9971

Table 3 lists the time it takes to perform the out-of-core sort for different number of items. Let us analyze these numbers in more detail. In an ideal case, where the out-of-core sort can be performed without any communication overhead, the time to takes to sort N with P processors, relative to sorting m items with a single processor is given by: $\beta = \frac{(N/P)\cdot\lg N\cdot(\lg N+1)/2}{m\cdot\lg m\cdot(\lg m+1)/2}$. For $N = 1\text{M}$, we have $\beta = 3.92$, whereas the observed β, denoted as β_o based on the results from Table 3 is 3.5. Then the overhead of out-of-core sort for $N = 1\text{M}$ items is $\frac{\beta_o-\beta}{\beta_o\cdot10^{-2}}\% = 12\%$. Recall that the overhead of the in-core sort for half the number of items was 3%. When we look at β for 128M, we see that the overhead becomes 50.82%. In other words, when sorting large numbers of items using out-of-core sort, the communication overhead becomes a significant component of the overall sort time. In fact when sorting 128K items, during the double buffered out-of-core j-k-swap sub-steps, the last issued DMA commands never complete before the time current buffer is completely processed. the time the current buffer is completely processed. This attests to the fact that the out-of-core sort becomes memory I/O bound as the number of items sorted increases.

Figures 14 and 15 plot the time it takes to perform the sort, relative to that of bitonic sort, for varying number of integer and float items, respectively. An important observation from the figures is that, the performance of 16 SPE bitonic sort degrades with increasing number of items, when compared against the alternative approaches. However, the

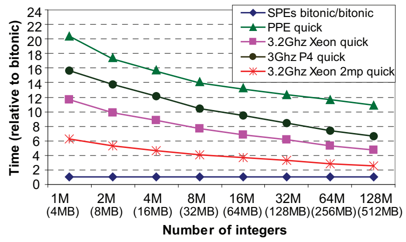

Figure 14: Out-of-core sort, integers

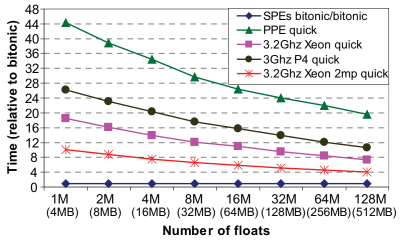

Figure 15: Out-of-core sort, floats

closest single-core approach (quick sort on the Xeon) still takes 4.8 times and 7.3 times longer to sort 0.5GB of integers and floats, respectively. Considering that roughly half of the sorting time is spent for communication (data transfers and synchronization) when sorting 128M items with bitonic sort, the closest single-core approach would be 14.6 times slower compared to bitonic sort with no communication cost assumption. This is still smaller than the 21 times slower result we got for in-core sort when using all 16 SPEs. In other words, even with no communication overhead there is still a reduction in the relative performance of bitonic sort compared to quick sort for increasing number of items. This can be attributed to the higher asymptotic complexity of bitonic sort, compared to that of quick sort. Yet, sorting 0.5GB of floats using 16 SPEs is still 4 times faster compared to parallel quick sort on a dual-core 3.2GHz Intel Xeon.

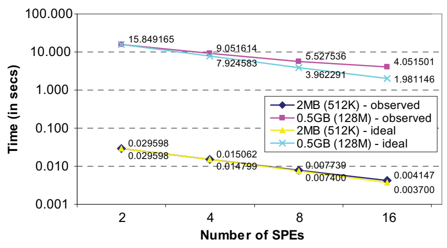

Figure 16: Scalability with number of SPEs

6.4 Scalability

Figure 16 studies the scalability with respect to number of processors, with two different data sizes, namely 2MB and 0.5GB. Notice that both x and y axises are in logarithmic scale. Figure 16 also plots the ideal scalability scenarios for both data sizes, where doubling the number of processors cuts the sort time by one half. We observe almost perfect scalability for 2MB data, in which case 16 SPE sort is in-core. For 0.5GB data, all sorts are out-of-core and the scalability is not as good. As the number of SPEs increase, the scalability of out-of-core sort decreases. Note that for 0.5GB of data, moving from 8 SPEs to 16 SPEs only bring a speed-up of 1.375, whereas going from 4 SPEs to 8 SPEs brings a speed-up of 1.64, and from 2 SPEs to 4 SPEs brings 1.75. This difference in scalability between in-core and out-of-core sort is again due to memory bandwidth bound nature of out-of-core sort. Increasing the number of SPEs increases the contention to access the main memory.

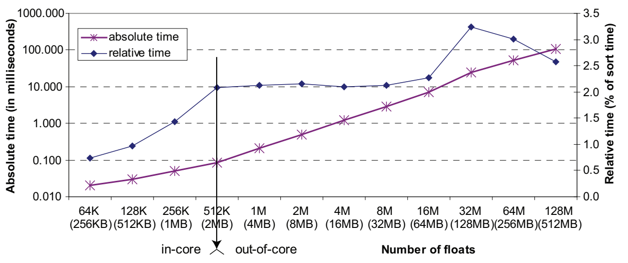

Figure 17: Synchronization cost

6.5 Synchronization Cost

Figure 17 plots the average time (in milliseconds) an SPE waits for synchronization as a function of the number of floats sorted, using the left y-axis (in logarithmic scale). The relative time with respect to the total sort time is also plotted, using the right y-axis. Two different trends are observed. First, synchronization time relative to sort time is mostly stable for the out-of-core sort. Second, for in-core sort, the synchronization cost increases with the number of items sorted. This is due to the increasing number of processors used for the barriers. Note that the average time an SPE spends waiting for synchronization is almost always less than 3% of the sort time.

6.6 Discussions

A number of recent works (see Section 7) on GPU (graphics processing unit) based sorting algorithms have reported impressive performance results compared to CPU-based sorts on high-end processors. Since we do not have access to code from these works, it is not possible to present an apples-to-apples comparison of CellSort with GPU-based sorts. However, here we would mention a brief comparison against the recent AbiSort algorithm [10], using the results reported on a GeForce 7800 system. Since the authors of [10] report results on sorting float items with attached pointers, we doubled our performance results for comparison purposes (our results are from key-only sorts). This is a very conservative comparison, since sorting key-value pairs instead of keys-only is likely to result in a smaller increase in sort time compared to doubling the sort time. However, even with this assumption CellSort with 16 SPEs is significantly superior to AbiSort on GeForce 7800. Concretely, it takes 135ms to sort 1M ⟨float, pointer⟩ pairs with AbiSort, whereas the conservative estimate for CellSort is only 19.5ms, which is around 1/7th of AbiSort's sort time.

In summary, high performance of CellSort stems from three major factors.

1. With proper SIMDization, sorting on an SPE can be made to perform as good as or even better than sorting on a similarly clocked single-core high-end processor, bringing 8 times more computational capacity per Cell processor. Without SIMDization and other low level optimizations such as loop unrolling and branch avoidance, the scalar performance of the SPEs is poor.

2. The high bandwidth provided by the Cell processor for cross-SPE data transfers, together with the use of asynchronous DMAs and optimal SPE communication patters outlined in this paper, enables us to implement distributed bitonic merge with minimal overhead. As a result, in-core sort has close to linear scalability with increasing number of processors.

3. Bitonic merge lends itself not only to an efficient SIMD implementation, but also to an efficient distributed implementation. However, as a side-effect, it has sub-optimal complexity. This limits performance when the data size significantly exceeds the in-core limit. Considering that processors with hundreds of cores with more on-chip memory are in our horizon, this is less of an issue for the future.

7. RELATED WORK

Sorting algorithms have been studied extensively since the inception of the computer science discipline. [16] presents a comprehensive survey of the classic sorting algorithms. Sorting is a still considered a fundamental operation in many applications domains, e.g., database systems [9] and computer graphics [22].

Bitonic sort [3, 6] was designed for a comparator-based abstract sorting network. Bitonic sort's fixed computational pattern and underlying abstract execution model makes it suitable for parallel execution. Over the years, a variety of parallel sorting algorithms have been proposed for different parallel architectures and programming models [1]. Earliest parallel sorting algorithms were devised using the SIMD programming model for the vector multiprocessors like the Cray-YMP [27] or massively data-parallel multicomputers such as the CM-2 [5]. These multicomputer sorting algorithms optimized the fixed computational patterns of the Bitonic and Radix sort [16] using SIMD instructions. Alternatively, the sorting algorithms devised for distributed-memory machines and clusters employed variants of the parallel merge techniques [1]. Current generations of processors, e.g., Pentium 4 and its successors, provide hardware and software support for parallel programming on shared memory using threads (i.e., via hyper-threading and OpenMP primitives). These capabilities can be effectively used for implementing a parallel merge-based sort [20]. The key difference between these approaches and our implementation is that our implementation uses a combination of the distributed-memory, shared-memory, and data-parallel algorithms.

In addition to the Cell, most current computer architectures support instructions for short-vector (e.g., 128 bit) SIMD parallelism, e.g., AltiVec on PowerPC and SSE3 for Intel [25, 28]. Recent generations of GPUs, such as the nVIDIA 8800, enable data-parallel implementations of general purpose computations using 2-dimensional texture representation of input data [22, 19]. In particular, GPUs have been shown to suitable for accelerating the bitonic sort-based algorithms [23, 14, 8, 7, 11]. Conceptually, the GPU-based data-parallel implementations are closer to the multicomputer approaches as both implement data-parallel operations over large datasets. The GPU-based implementations of bitonic sort differ from our approach in many ways. Specifically, GPU-based algorithms donot stripmine computations due to lack of available memory and they donot distribute work over multiple computational units. However, unlike Cell, the GPU-based algorithms are affected by cache memory latencies and need to be optimized for memory efficiency (e.g., using appropriate data layout and tiling) [8]. Finally, the any GPU-based sorting implementation involves non-trivial mapping of the basic datatypes into pixels in the GPU's texture memory [7].

The GPUTeraSort implementation used Radix sort in their sorting kernel. We have also investigated Radix sort and its variant, the Postman's sort [24] for the SPE sorting kernel. We have found that both Radix and Postman's sort involve extensive scalar updates using indirection arrays that are difficult to SIMDize and thus, degrade the overall performance.

The multi-stage approach used in our implementation is similar to the one used in parallel disk-based sorts [1] (in our case, the external storage is the off-chip main-memory, not disk). For example, AlphaSort [18], a shared-memory based parallel external sort, uses quicksort as the sequential sorting kernel and a replacement-selection tree to merge the sorted subsets.

8. CONCLUSIONS

In this paper we described a distributed implementation of the bitonic sort tailored for the Cell processor. Large number of items, that do not fit into the total space provided by the local stores of the participating SPEs, are sorted using a three-tiered approach. A SIMDized bitonic sorting kernel is used to sort items locally in the local stores of the SPEs, a distributed in-core bitonic merge is used to merge local store resident local sort results, and a distributed out-of-core bitonic merge is used to merge the results of a number of main memory resident in-core sort results. Our study shows that the SIMDized bitonic sort is the most effective sorting kernel for the local sort part and the in-core sort on the Cell processor scales well with the increasing number of participating SPEs. The in-core bitonic sort is not memory I/O bound and can significantly surpass the performance of quick sort on similarly clocked Intel processors. On the other hand, the out-of-core bitonic sort on Cell becomes memory I/O bound and its performance degrades with increasing number of items sorted. The non-optimal computational complexity of the bitonic sort also contributes to this degradation. However, distributed out-of-core bitonic sort still compares favorable against other alternatives on competing processors with similar clock speeds.

9. REFERENCES

[1] S. G. Akl. *Parallel Sorting Algorithms.* Academic Press Inc., 1990.

[2] D. Bader, V. Agarwal, and K. Madduri. On the design and analysis of irregular algorithms on the Cell processor: A case study on list ranking. In *Proc. of IEEE IPDPS*, 2007.

[3] K. E. Batcher. Sorting networks and their applications. In *Proc. of the Spring Joint Computer Conference*, 1968.

[4] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos. RAxML-Cell: Parallel phylogenetic tree inference on the Cell Broadband Engine. In *Proc. of IEEE IPDPS*, 2007.

[5] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zegha. A comparison of sorting algorithms for the connection machine CM-2. In *Proc. of ACM SPAA*, 1991.

[6] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *Journal of the ACM*, 36(4):738–757, October 1989.

[7] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proc. of ACM SIGMOD*, 2006.

[8] N. K. Govindaraju, N. Raghuvanshi, M. Hanson, D. Tuft, and D. Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, Department of Computer Science, University of North Carolina, 2005.

[9] G. Graefe. Implementing sorting in database systems. *ACM Computing Surveys*, 38(3), 2006.

[10] A. Greß and G. Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. Technical Report IfI-06-11, TU Clausthal, 2006.

[11] A. Greß and G. Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proc. of IEEE IPDPS*, 2006.

[12] IBM. Cell Broadband Engine Architecture. Technical Report Version 1.0, IBM Systems and Technology Group, 2005.

[13] Intel. IXP2400 network processor hardware reference manual. Technical report, Intel Corporation, May 2003.

[14] P. Kipfer and R. Westermann. *Improved GPU Sorting*, chapter 46. Addison Wesley, 2005.

[15] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor interconnection network: Built for speed. *IEEE Micro*, 26(3), 2006.

[16] D. E. Knuth. *The Art of Computer Programming, volume 3: Searching and Sorting.* Addison Wesley, 1973.

[17] Mercury Systems Dual Cell-based Blade. www.mc.com/literature/literature_files/Cell_blade-ds.pdf, Feburary 2007.

[18] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A cache-sensitive parallel external sort. *VLDB Journal*, 4, 1995.

[19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26, 2007.

[20] R. Parikh. Accelerating QuickSort on the Intel Pentium-4 processor with Hyper-Threading technology. Intel Developer Network.

[21] F. Petrini, G. Fossum, J. Fernandez, A. L. Varbanescu, M. Kistler, and M. Perrone. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *Proc. of IEEE IPDPS*, 2007.

[22] M. Pharr, editor. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison Wesley, 2005.

[23] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proc. of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (EGGH'03)*, 2003.

[24] R. Ramey. The Postman's sort. *C/C++ Users Journal*, August 1992.

[25] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *Proc. of ACM PLDI*, 2006.

[26] V. Sachdeva, M. Kistler, E. Speight, and T.-H. K. Tzeng. Exploring the viability of the Cell Broadband Engine for Bioinformatics applications. In *Proc. of the IEEE International Workshop on High Performance Computational Biology (HiCOMB'07)*, 2007.

[27] M. Zagha and G. E. Blelloch. Radix Sort for vector multiprocessors. In *Proc. of Supercomputing*, 1991.

[28] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proc. of ACM SIGMOD*, 2002.