Optimized Pipelined Parallel Merge Sort on the Cell BE

Jörg Keller[1] and Christoph W. Kessler[2]

[1] FernUniversität in Hagen, Dept. of Math. and Computer Science, 58084 Hagen, Germany
[2] Linköpings Universitet, Dept. of Computer and Inf. Science, 58183 Linköping, Sweden

Abstract. Chip multiprocessors designed for streaming applications such as Cell BE offer impressive peak performance but suffer from limited bandwidth to off-chip main memory. As the number of cores is expected to rise further, this bottleneck will become more critical in the coming years. Hence, memory-efficient algorithms are required. As a case study, we investigate parallel sorting on Cell BE as a problem of great importance and as a challenge where the ratio between computation and memory transfer is very low. Our previous work led to a parallel mergesort that reduces memory bandwidth requirements by pipelining between SPEs, but the allocation of SPEs was rather ad-hoc. In our present work, we investigate mappings of merger nodes to SPEs. The mappings are designed to provide optimal trade-offs between load balancing, buffer memory consumption, and communication load on the on-chip bus. We solve this multi-objective optimization problem by deriving an integer linear programming formulation and compute Pareto-optimal solutions for the mapping of merge trees with up to 127 merger nodes. For mapping larger trees, we give a fast divide-and-conquer based approximation algorithm. We evaluate the sorting algorithm resulting from our mappings by a discrete event simulation.

1 Introduction

Multiprocessors-on-chip are about to become the typical processors to be found in desktops, notebooks and clusters. Besides multicores based on x86 architectures, we also find new designs such as the Cell Broadband Engine processor with 8 parallel processors called SPEs and a Power core (see e.g. [1] and the references therein). Currently, explicit parallel programming is necessary to exploit the raw power of these processors. Many applications use the Cell BE like a dancehall architecture, i.e. all SPEs load data from the external memory, and use their small local memories (256 KB for code and data) as explicitly-managed caches. Yet, as the bandwidth to the external memory is the same as each SPE's bandwidth to the element interconnect bus (EIB) [1], the external memory limits performance and prevents scalability. Bandwidth to external memory is a common bottleneck in multiprocessors-on-chip, and the increasing number of cores will intensify the problem [2]. A scalable approach to parallelism on such architectures therefore must use communication between the SPEs to reduce communication with external memory.

Sorting is an important subroutine in applications ranging from computational geometry to bio informatics and data bases. Parallel sorting algorithms on a wealth of architectures have therefore attracted considerable interest continuously for the last decades,

E. César et al. (Eds.): Euro-Par 2008 Workshops, LNCS 5415, pp. 131–140, 2009.

see e.g. [3,4]. As the computation to memory-transfer ratio is quite low in sorting, it presents an interesting case study to develop bandwidth efficient algorithms.

Sorting on the Cell BE presents several challenges. First, the SPEs' local memories are so small that most parallel sorting algorithms must mainly use the external memory, and thus will not be memory-efficient. Algorithms which do not suffer from this problem must also have very simple, data-independent control structures that are able to efficiently use the SPEs' SIMD structure and minimize branching. Sorting algorithms implemented for the Cell BE [5,6] use bitonic sort or merge sort and work in two phases to sort a data set of size n with local memories of size n'. In the first phase, blocks of data of size $8n'$ that fit into the combined local memories of the 8 SPEs are sorted. In the second phase, those sorted blocks of data are combined to a fully sorted data set. We concentrate on the second phase as the majority of memory accesses occurs there. In [5], this phase is realized by a bitonic sort because this avoids data dependent control flow and thus fully exploits the SIMD architecture of the SPEs. Yet, $O(n \log^2 n)$ memory accesses are needed, and the reported speedups are small. In [6], mergesort with 4-to-1-mergers is used in the second phase, where the mergers use bitonic merge locally. The data flow graph of the merge procedures thus forms a fully balanced quad-tree. As each SPE reads from main memory and writes to main memory, all n words are transferred from and to main memory in each round, resulting in $n \log_4(n/(8n')) = O(n \log_4 n)$ data being read from and written to main memory. While this improves the situation, speedup still is limited.

In order to overcome this bottleneck, we propose to run merger nodes belonging to consecutive layers of the merge tree concurrently, so that output from one merger is not written to main memory but sent to the SPE running the follow-up merger node, i.e. we use a form of pipelining. If we can embed k-level b-ary merge trees in this way, we are able to realize parallelized b^k-to-1 merge routines and thus increase the ratio of computation to memory transfer by a factor of $k \cdot \log_4 b$. Yet, this must be done such that all SPEs are kept busy. As in [6], a merger node does not process complete blocks of data before forwarding its result block, but uses fixed sized chunks of the blocks, i.e. a merger node is able to start work as soon as it has one chunk of each of its input blocks, and as soon as it has produced one chunk of the output block, it forwards it to the follow-up node. This form of streaming allows the use of fixed size buffers, holding one chunk each. To overlap data transfer and computation, the merger nodes should use double buffering at least for their inputs, and the buffers should have a reasonable minimum size to allow for efficent data transfer between SPEs.

Both [6] and our approach may benefit from a sample sort [7] preprocessing to reduce the problem to p sorts of cn/p data each, where $c \leq 3$ with high probability, which avoids $\log_4 p$ and $\log_{b^k} p$ rounds, respectively.

Ensuring that our pipeline runs close to the maximum possible speed requires load balancing. If a merger node u must provide an output rate of τ words per time unit, then the mergers u_i, where $1 \leq i \leq b$, feeding its inputs must provide a rate of τ/b words per time unit on average. However, if the values in the output chunk produced by u_i are much larger than those in u_j (see Fig. 1), u will only process values from u_j for some time, so that u_j must produce at a double rate for some time, while u_i will be stalled because of finite buffering between u_i and u. Otherwise the rate of u will reduce.

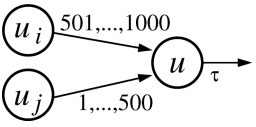

Fig. 1. Load balancing between merger nodes

Finally, the merger nodes should be distributed over the SPEs such that not all communication between merger nodes leads to communication between SPEs, in order not to overload the EIB.

The remainder of this article is organized as follows. In Section 2, we present the mapping problem sketched here in a formal way, give an integer linear programming solution to compute an optimal mapping of a b-ary merge tree onto the SPEs for small and medium-sized merge trees, and present an approximation algorithm based on divide-and-conquer. In Section 3, we discuss how our mapping turns into an efficient sorting algorithm, and we present simulation results. Section 4 concludes.

2 Mapping Trees onto Processors

2.1 Definitions

Given is a set $P = \{P_1, \ldots, P_p\}$ of p processors interconnected by a ring, and a k-level balanced b-ary tree $T = (V, E)$ directed towards its root, to be mapped onto the processors. Information in the tree flows from the leaves towards the root, input being fed in at the leaves and output leaving the tree root. Each node v in the tree processes b designated incoming data streams and combines them into one outgoing data stream of rate $0 < \tau(v) \leq 1$. Hence, the incoming data streams on average will have rate $\tau(v)/b$, if we assume finite buffering within nodes.

The *computational load* $\gamma(v)$ that a node v places on a processor that it is mapped to is proportional to its output rate $\tau(v)$, hence $\gamma(v) = \tau(v)$. The tree root r has a normalized output rate of $\tau(r) = 1$. Thus, each node v on level i of the tree, where $0 \leq i \leq k-1$, has $\tau(v) = b^{-i}$ on average. The computational load and output rate may also be interpreted as node and edge weights, respectively. For $T_l(v)$ being the l-level sub-tree rooted in v, we extend the definitions to $\tau(T_l(v)) = \tau(v)$ and $\gamma(T_l(v)) = \sum_{u \in T_l(v)} \gamma(u)$. Note that $\gamma(T_l(v)) = l \cdot \gamma(v)$, because the accumulated rates of siblings equal the rate of the parent. For nodes u and v not in a common sub-tree, $\tau(\{u, v\}) = \tau(u) + \tau(v)$ and $\gamma(\{u, v\}) = \gamma(u) + \gamma(v)$. In particular, the computational load and output rate of any tree level equals 1. The *memory load* that a node v will place on the processor it is mapped to is a constant value c, because the node needs a fixed amount for buffering transferred data and for the internal data structures it uses for processing the data. For simplicity, one may assume $c = 1$ in the sequel.

We construct a mapping $\mu : V \to P$ of tree nodes to processors. Under this mapping μ, a processor P_i has *computational load*[1] $C_\mu(P_i) = \sum_{v \in \mu^{-1}(P_i)} \tau(v)$, i.e. the sum of

[1] The computational load depends on τ and thus averaged over time.

the load of all nodes mapped to it, and it has *memory load* $M_\mu(P_i) = \sum_{v \in \mu^{-1}(P_i)} c = c \cdot \#\mu^{-1}(P_i)$. The mapping μ shall have the following properties:

1. The maximum computational load $C^*_\mu = \max_{P_i \in P} C_\mu(P_i)$ among the processors shall be minimized. This requirement is obvious, because the lower the maximum computational load, the more evenly the load is distributed over the processors. With a completely balanced load, C^*_μ will be minimized.
2. The maximum memory load $M^*_\mu = \max_{P_i \in P} M_\mu(P_i)$ among the processors shall be minimized. The maximum memory load is proportional to the number of the buffers. As the memory per processor is fixed, the maximum memory load determines the buffer size. If the buffers are too small, communication performance will suffer.
3. As often as possible, sibling nodes shall be mapped to the same processor. We refer to the discussion on load balancing in Sect. 1.
4. The *communication load* $L_\mu = \sum_{(u,v) \in E, \mu(u) \neq \mu(v)} \tau(u)$, i.e. the sum of the edge weights between processors, shall be low.

Lemma 1 (Lower bounds). *In any mapping μ the maximum computational load is at least k/p, and the maximum memory load is at least $\lceil c \cdot (b^k - 1)/((b-1)p) \rceil$.*

We omit the routine proof of Lemma 1. The latter bound can be tightened for the case $p = k$. If no processor is overloaded, the root must be placed on a processor of its own, so that the rest of the tree is mapped onto $p - 1$ processors, leading to $M^*_\mu \geq c((b^k - 1)/(b-1) - 1)/(k-1) = c(b^k - b)/((b-1)(k-1))$.

For larger chip-multiprocessors, e.g. with $p \geq 20$, the assumption $k = p$ might lead to problems because the tree gets very large. In this case, we choose a small k, map the tree onto $p' = k$ *pseudo*-processors, and implement each pseudo-processor with p/k processors by evenly distributing the nodes assigned to that pseudo-processor. If fewer than p/k nodes are mapped to a processor (e.g. if the root is mapped separately), then we use a technique already known [4] and mentioned in [6]: we partition the very large data blocks and perform merges on the partitions in parallel.

2.2 ILP Formulation

In the following, we number the tree nodes in breadth-first order, i.e. the root gets index 1, its children 2, 3 etc., and generally, the ith child of an inner node v gets index $b \cdot (v-1) + i + 1$, for $i = 1, 2, ..., b$. Let $V = \{1, ..., (b^k - 1)/(b-1)\}$ denote the set of tree nodes, $V_{inner} = \{1, ..., (b^{k-1} - 1)/(b-1)\}$ the set of inner nodes, and $P = \{1, ..., p\}$ the set of available SPEs. Our ILP formulation uses three arrays of $O(b^k \cdot p)$ boolean variables, x, y and z. The actual solution, i.e. the mapping of nodes to processors, will be given by x:

$$x_{v,q} = 1 \text{ iff tree node } v \text{ is mapped on processor } q.$$

In order to determine internal edges (where both source and target node are mapped to the same processor) and siblings on the same processor, we need to introduce auxiliary variables z and y:

$$z_{u,q} = 1 \text{ iff non-root node } u > 1 \text{ and its parent are mapped to processor } q.$$

$y_{u,q} = 1$ iff all children $b(u-1)+2, \ldots, b \cdot u + 1$ of inner node u are mapped to proc. q.

Also, we use an integer variable *maxMemoryLoad* that will indicate the maximum memory load assigned to any SPE in P, and integer variable *nSiblingsOnDiffSPEs* that will indicate the total number of inner nodes whose children are all mapped to the same processor. The following constraints must hold:
Each node must be mapped to exactly one processor, and each processor can be filled up to 100% with work[2]:

$$\forall v \in V: \quad \sum_{q \in P} x_{v,q} = 1 \qquad \forall q \in P: \quad \sum_{v \in V} x_{v,q} \cdot \tau(v) \leq 1$$

The memory load should be balanced:

$$\forall q \in P: \quad \sum_{v \in V} x_{v,q} \leq \textit{maxMemoryLoad}$$

Communication cost occurs whenever an edge is not internal, i.e. its endpoints are mapped to different SPEs. To avoid products of two x variables when determining which edges are internal, we use the following constraints and slack variables z:

$$\forall v \in V_{inner},\ q \in P,\ i \in \{1, ..., b\}: \quad \begin{aligned} z_{b(v-1)+i+1,q} &\leq x_{v,q} \\ z_{b(v-1)+i+1,q} &\leq x_{b(v-1)+i+1,q} \end{aligned}$$

and in order to enforce that a $z_{u,q}$ will be 1 wherever it could be, we have to take up the (weighted) sum over all z in the objective function. This means, of course, that only optimal solutions to the ILP are guaranteed to be correct with respect to minimizing memory load and communication cost.

The communication load is the total communication volume over all tree edges minus the volume over the internal edges:

$$\textit{commLoad} = \sum_{v \in V - \{1\}} \tau(v) - \sum_{v \in V_{inner}} \sum_{q \in P} \left(\sum_{1 \leq i \leq b} z_{b(v-1)+i+1,q} \right) \cdot \tau(bv)$$

We apply the same trick to determine $y_{v,q}$:

$$\forall v \in V_{inner},\ q \in P,\ i \in \{1, ..., b\}: \quad y_{v,q} \leq x_{b(v-1)+i+1,q}$$

The total number of nodes whose children are mapped to different processors is then

$$\textit{nSiblingsOnDiffSPEs} = \sum_{v \in V_{inner}} \sum_{q \in P} (1 - y_{v,q})$$

Finally, the objective function is:

$$\text{Minimize} \quad \epsilon_M \cdot \textit{maxMemoryLoad} + \epsilon_C \cdot \textit{commLoad} + \epsilon_S \cdot \textit{nSiblingsOnDiffSPEs}$$

where the positive weight parameters ϵ_M, ϵ_C and ϵ_S can be set appropriately to give preference to minimizing for *maxMemoryLoad*, *commLoad*, or *nSiblingsOnDiffSPEs* as first optimization goal. The formulation above requires that $\epsilon_C > 0$ and $\epsilon_S > 0$.

[2] We focus on the case $k = p$; the general case would need the constraint $\leq k/p$.

Table 1. The Pareto-optimal solutions found with ILP for $b = 2$, $k = p = 5,\ 6,\ 7$

k	5			6				7		
# binary var.s	305			750				1771		
# constraints	341			826				1906		
maxMemoryLoad	8	9	10	13	14	15	20	21	29	30
commLoad	2.5	2.375	1.75	2.625	2.4375	1.9375	1.875	2.375	2.3125	2.0

By choosing the ratio of ϵ_M to ϵ_C, we can only find two extremal Pareto-optimal solutions, one with least possible *maxMemoryLoad* and one with least possible *commLoad*. In order to enforce finding further Pareto-optimal solutions that may exist in between, one can use any fixed ratio ϵ_M/ϵ_C, e.g. at 1, and instead set a given minimum memory load to spend (which is integer) on optimizing for *commLoad* only:

$$\mathit{maxMemoryLoad} \geq \mathit{givenMinMemoryLoad}$$

2.3 ILP Optimization Results

We implemented the above ILP model in CPLEX 10.2 [8], a commercial ILP solver. Table 1 shows all Pareto-optimal solutions that CPLEX found for $b = 2$ and $k = p = 5, 6, 7$. The computations for $k = 5$ and $k = 6$ took just a few seconds each, the time to optimize for $k = 7$ varied between a few seconds and several hours per *givenMinMemoryLoad*. For $k = 8$, with 5088 binary variables and 6369 constraints, CPLEX exceeded the timeout of 24 hours and could only produce approximate solutions, including one with *maxMemoryLoad* of 37 and a *commLoad* of 2.78125, and one with 38 and 2.71875, respectively.

By varying ϵ_M/ϵ_C and keeping $\epsilon_S \ll \epsilon_C$, two of the Pareto-optimal solutions can be found, namely that with best *maxMemoryLoad* and that with best *commLoad*. As the memory load is often one order of magnitude larger than communication load, $\epsilon_C \gg \epsilon_M$ is necessary to spot the communication-optimal one. The remaining Pareto-optimal solutions in between can be found by setting *givenMinMemoryLoad* appropriately. We use a very small ϵ_S, to give the sibling placement optimization the least priority and not interfere with communication optimization. Figure 2 shows the generated tree drawings for two of the solutions for $k = 5$. The tree computed for $k = 7$ with minimum *commLoad* is shown in Figure 3.

2.4 A Divide-and-Conquer Based Approximation Algorithm

For larger values of k, we use the following divide-and-conquer algorithm (called DC-map in the sequel) which we first present for $b = 2$, and then extend to arbitrary b.

To construct a mapping for a k_1-level binary tree onto k_1 processors, we distinguish two cases. If $k_1 \leq k_0$, where k_0 is a constant, we take a precomputed optimal mapping. Currently we use $k_0 = 7$. If $k_1 > k_0$, we place the tree root onto one processor, and interpret the remaining $k_1 - 1$ processors as two sets of $k_1 - 1$ processors, each with half the computational power. We map a $(k_1 - 1)$-level tree onto each set recursively. Then

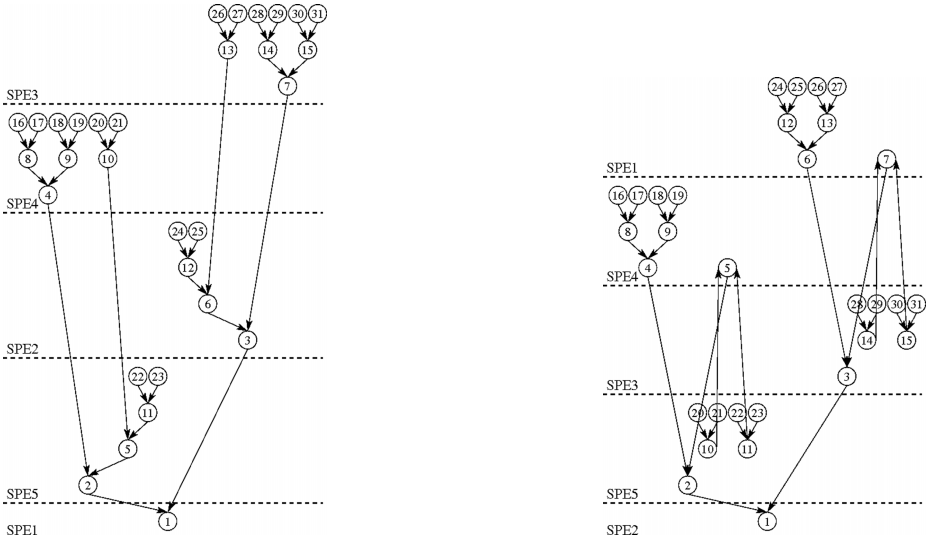

Fig. 2. Two Pareto-optimal solutions for mapping a 5-level tree onto 5 processors, computed by the ILP solver. — Left hand side: max. memory load 10 and communication load 1.75, obtained e.g. for $\epsilon_M = 0.1\epsilon_C$. — Right hand side: max. memory load 8 and communication load 2.5, obtained e.g. for $\epsilon_M = 10\epsilon_C$.

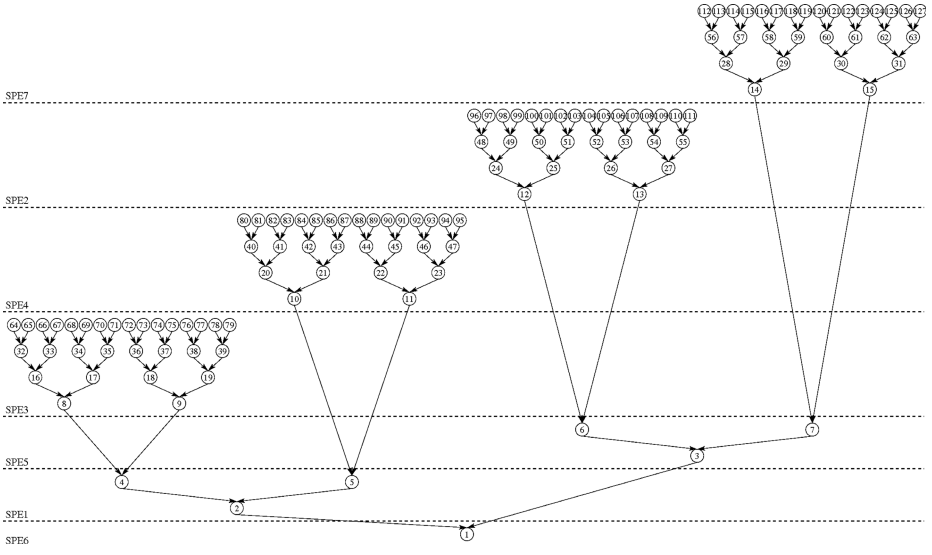

Fig. 3. The Pareto-optimal solution for mapping a 7-level tree onto 7 processors with least communication load, computed by ILP

we sort the processors in each set according to their memory load, one set in ascending order, one set in descending order. Finally we re-combine the i-th processors from both lists into one processor with full computational power.

We illustrate DC-map by an example where we employ an optimal mapping for $k_0 = 5$ (Fig. 2 right hand side), and construct a mapping for $k_1 = 6$. We first place the root of the 6-level tree onto processor 6. The two 5-level trees are mapped onto 5 'half'-processors each with the help of the optimal mapping, with memory loads of $8, 8, 7, 7, 1$. As this list is already sorted in descending order, we sort the copy in ascending order and receive $1, 7, 7, 8, 8$. Combination of the lists results in memory loads of $9, 15, 14, 15, 9$, and thus a maximum memory load of 15, compared to a sharpened lower bound of 13, but still representing a Pareto-optimal solution from Table 1.

To map a tree with $b > 2$, we receive b lists from the recursion step, that we may successively combine into pairs, as in a balanced binary tree with b leaves. Alternatively, we might use some form of linear optimization here.

DC-map mainly sorts lists of increasing length, thus its runtime is $\sum_{k=k_0}^{k_1} O(k \log k)$ = $O(k_1^2 \log k_1)$, which can be considered efficient given that typically $k_1 \ll 10^3$. By construction, DC-map produces a mapping where each processor has a computational load of 1. The maximum memory load may increase by a factor of b when going from k to $k+1$, because b lists are to be combined. In contrast, the lower bound increases by a factor

$$\frac{\frac{b^{k+1}-1}{(b-1)(k+1)}}{\frac{b^k-1}{(b-1)k}} \approx b \cdot \frac{k}{k+1}$$

Thus, if we start with an optimal solution for k_0 and use DC-map to construct a solution for $k_1 > k_0$, the maximum memory load may increase by a factor of $b^{k_1-k_0}$, while the lower bound increases by a factor $b^{k_1-k_0} k_0/k_1$. Thus, we may be away from the optimum maximum memory load by a factor of k_1/k_0.

DC-map does not take special care for the placement of siblings or communication load. Yet, with respect to siblings, the majority of the nodes and thus the siblings is in the levels close to the leaves, which are placed with the help of an optimal mapping. With respect to communication load, we may employ the following additional step. Normally, the two 'half'-processors to be combined into one are from different lists, i.e. they carry nodes from different subtrees that are not connected by edges. Yet, when the two lists are combined, we may interchange pairs of 'half'-processors with identical memory load without disturbing the algorithm. If the 'half'-processors to be interchanged are from different lists, then their partners in the combination are now from the same list, and the nodes they carry may be connected by edges that now become internal.

We have implemented a prototype version of DC-map, albeit without the improvement of communication load. We evaluated the prototype on the basis of optimal solutions for $k_0 = 3$ and $k_0 = 7$. Table 2 depicts the placement results achieved for $k_1 = 3, 4, \ldots, 8$ and $k_1 = 7, \ldots, 12$, respectively. From the numbers it is clear that the algorithm in practice is much closer to the lower bound than by a factor of k_1/k_0.

Table 2. Results for DC-map prototype

	$k_0 = 3$						$k_0 = 7$					
k_1	3	4	5	6	7	8	7	8	9	10	11	12
M_μ^*	3	6	8	15	24	46	21	42	84	132	236	453
lower bound	3	5	8	13	21	37	21	37	64	114	205	373
quotient	1.00	1.20	1.00	1.15	1.14	1.24	1.00	1.14	1.31	1.16	1.15	1.21
k_1/k_0	1.00	1.33	1.66	2.00	2.33	2.66	1.00	1.14	1.29	1.43	1.57	1.71

3 Sorting Algorithm and Performance

In order to test the usefulness of our mappings, e.g. with regard to load balancing, we implemented a discrete event simulation of the second phase of the parallel merge sorting algorithm. As the runtime of each merger node to produce one chunk of output is only dependent on the size of the output buffer, it is considered a constant. As furthermore communication and computation are assumed to be overlapped, we believe the simulation to quite accurately reflect the full algorithm. We chose $b = 2$ because quad-trees lead to high memory load, i.e. to very small buffer sizes, even for small k.

In each step, each SPE runs one merger node that has enough input data until it has produced one chunk, i.e. one output buffer full of data. As buffer size, we use 4 KByte for the output buffer (holding 1,024 32-bit integers), and 2×4 KByte for the input buffers, in order to allow a merger to commence work on its input data, while its input is being simultaneously filled with the output of a previous merger. Each merger mapped to a particular SPE receives a share of the SPE's processing time at least according to its position in the merge tree, i.e. a node at level $i \geq 0$ receives a share of at least 2^{-i}, provided that it has enough input to produce one chunk of output. We use a simple round robin scheduling policy in each SPE, where a merger receives a number of slots in proportion to its share. A merger not ready to run (e.g. insufficient input or full output buffer) is simply left out.

We have investigated three mappings resulting from our mapping algorithm. In the 5-level tree of Fig. 2 (right hand side), we have realized a 32-to-1 merge on 5 SPEs, with the restriction that no more than 8 mergers are to be mapped to one SPE. With 20 KByte of buffering (5 buffers of 4 KByte each) for each merger, this seems to be the upper limit. We used 32 input blocks of 2^{20} sorted integers each. The blocks were filled with randomly chosen integers and then sorted. The pipeline ran with an efficiency of 93%, meaning that in 93% of the time steps, the root merger node could run and produce output. In comparison to [6], our memory bandwidth requirements decreased by a factor of 2.5. Combined with a pipeline efficiency of 93%, we still gain a factor of 1.86.

By way of comparison, we also consider mapping a 4-level tree where leaf merger nodes over 4 SPEs, instead of 2, so that we use 6 SPEs in total. Thus, load balancing should not pose a problem. We have simulated this mapping with 16 input blocks of 2^{20} integers each, chosen as before. In all experiments, the pipeline ran with 100% efficiency as soon as it was filled. As we realize a 16-to-1 merge, we gain a factor of 2 on the memory bandwidth requirements in relation to [6]. Yet, as we need 6 SPEs instead of 4 (which would be the normal case $p = k$), our real improvement is only

$2 \cdot 4/6 = 4/3$ in this case. This mapping would be targeted towards a Cell BE variant with 6 SPEs as used in a Playstation 3.

Finally, we also simulated an 8-level tree on 8 processors. In this simulation, we had to reduce the buffer size to 256 bytes (64 integers), because the maximum memory load is 60 (resulting from DC-map with $k_0 = 2$), so that 300 buffers must be placed into the local memory of one SPE. As typically at most half the memory is available for data because of code size, and there are other data structures must be stored, too, using 75 KBytes for buffers seemed the upper limit. The simulation ran with an efficiency of at least 98% in all simulations. Thus, in comparison to the algorithm running with 4-to-1 mergers on 8 SPEs, i.e. on a complete Cell BE, our algorithm reduces memory bandwidth requirements by a factor of $0.98 \cdot \log_4(256) = 3.92$. Also we see that our algorithm can cope with rather small buffer sizes, as long as computation and communication can be overlapped.

4 Conclusion

We have investigated how to lower memory bandwidth requirements in the Cell BE by pipelining, with sorting being used as our case study. We have formulated the mapping of the merge tree onto the processors as an integer linear optimization problem, and given solutions for small tree sizes. For larger sizes, we presented a divide-and-conquer approximation algorithm. The mapping turns into a sorting algorithm whose performance we have demonstrated by a discrete event simulation. Note that our resulting sorting algorithm is also able to run on multiple Cell processors, as does [6]. At the beginning, there will be many blocks, and hence many b^k-to-1 mergers can be employed. In the end, when nearing the root, we are able to employ parallel mergers similar to the case $p > k$ discussed in Sect. 2.1. Approaches similar to the one presented here may work for other memory-intensive problems as well, such as data-parallel computations. We therefore plan to investigate other applications in the future.

References

1. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation—a performance view. IBM J. Res. Devel. 51(5), 559–572 (2007)
2. Huh, J., Keckler, S.W., Burger, D.: Exploring the design space of future CMPs. In: Proc. Int.l Conf. Parallel Architectures and Compilation Techniques (PACT 2001), pp. 199–210 (2001)
3. Akl, S.G.: Parallel Sorting Algorithms. Academic Press, London (1985)
4. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading (1992)
5. Gedik, B., Bordawekar, R., Yu, P.S.: Cellsort: High performance sorting on the cell processor. In: Proc. 33rd Intl. Conf. on Very Large Data Bases, pp. 1286–1207 (2007)
6. Inoue, H., Moriyama, T., Komatsu, H., Nakatani, T.: AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In: Proc. Int.l Conf. Parallel Architectures and Compilation Techniques (PACT 2007), pp. 189–198 (2007)
7. Shi, H., Schaeffer, J.: Parallel sorting by regular sampling. Journal of Parallel and Distributed Computing 14, 361–372 (1992)
8. ILOG Inc.: Cplex version 10.2 (2007), `http://www.ilog.com`