Journal of Universal Computer Science, vol. 18, no. 14 (2012), 1987-2023
submitted: 28/3/12, accepted: 27/7/12, appeared: 28/7/12 © J.UCS

Optimized On-Chip-Pipelining for Memory-Intensive Computations on Multi-Core Processors with Explicit Memory Hierarchy

Jörg Keller
(FernUniversität in Hagen, Germany
joerg.keller@fernuni-hagen.de)

Christoph W. Kessler, Rikard Hultén
(Linköpings Universitet, Sweden
chrke@ida.liu.se, rikard.hulten@gmail.com)

Abstract: Limited bandwidth to off-chip main memory tends to be a performance bottleneck in chip multiprocessors, and this will become even more problematic with an increasing number of cores. Especially for streaming computations where the ratio between computational work and memory transfer is low, transforming the program into more memory-efficient code is an important program optimization.

On-chip pipelining reorganizes the computation so that partial results of subtasks are forwarded immediately between the cores over the high-bandwidth internal network, in order to reduce the volume of main memory accesses, and thereby improves the throughput for memory-intensive computations. At the same time, throughput is also constrained by the limited amount of on-chip memory available for buffering forwarded data. By optimizing the mapping of tasks to cores, balancing a trade-off between load balancing, buffer memory consumption, and communication load on the on-chip network, a larger buffer size can be applied, resulting in less DMA communication and scheduling overhead.

In this article, we consider parallel mergesort as a representative memory-intensive application in detail, and focus on the global merging phase, which is dominating the overall sorting time for larger data sets. We work out the technical issues of applying the on-chip pipelining technique, and present several algorithms for optimized mapping of merge trees to the multiprocessor cores. We also demonstrate how some of these algorithms can be used for mapping of other streaming task graphs.

We describe an implementation of pipelined parallel mergesort for the Cell Broadband Engine, which serves as an exemplary target. We evaluate experimentally the influence of buffer sizes and mapping optimizations, and show that optimized on-chip pipelining indeed speeds up, for realistic problem sizes, merging times by up to 70% on QS20 and 143% on PS3 compared to the merge phase of CellSort, which was by now the fastest merge sort implementation on Cell.

Key Words: parallel merge sort, on-chip pipelining, multicore computing, task mapping, streaming computations
Category: C.1.4, D.1.3, D.3.4, F.1.2, F.2.2

1 Introduction

The new generation of multiprocessors-on-chip derives its raw power from parallelism, and explicit parallel programming with platform-specific tuning is needed to turn this power into performance. Many applications use multiprocessors like a dancehall architecture: on-chip local memories, which are typically small, are used like caches, and all cores load and store data from/to the external (off-chip) main memory. The bandwidth to the external memory is typically much smaller than the aggregate bandwidth to the on-chip interconnection network. This limits performance and prevents scalability for an entire class of memory-bound applications. This problem will become more severe as the core count per chip is expected to increase considerably in the foreseeable future. Hence, scalable parallelization on such architectures should prefer direct communication between the worker cores to reduce communication with off-chip main memory.

In this paper, we consider the important domain of memory-intensive computations. Often, these are streaming computations, an important class of applications in embedded computing with wide-spread use in image and signal processing and in embedded control. We investigate the global merging phase of mergesort on Cell as a challenging case study, for the following reasons:

- The ratio of computation to data movement is low.
- The computational load of tasks varies widely (by a factor of 2^k for a binary merge tree with k levels).
- The computational load of a merge task is not fixed but varies over time. The value given is an average over a sufficiently large time intervall.
- Memory consumption is not proportional to computational load.
- Communication always occurs between tasks of different computational load.

These factors complicate the mapping of tasks to cores. In total, pipelining a merge tree is more difficult than task graphs of regular problems such as matrix vector multiplication.

The task graph of the global merging phase consists of a tree of merge tasks that should contain, in the lowest layer, at least as many merger tasks as there are cores available. Previous solutions like CellSort, see [GBY07], and AAsort, see [IMKN07], process the tasks of the merge tree layer-wise bottom-up in serial rounds, distributing the tasks of a layer equally over cores (there is no need to have more than one task per core). Each layer of the tree is then processed in a *dancehall* fashion, where each task operates on (buffered) operand and result arrays residing in off-chip main memory. This organization leads to relatively simple code but puts a high access load on the off-chip-memory interface.

On-chip pipelining reorganizes the overall computation in a pipelined fashion such that intermediate results (i.e., temporary stream packets of sorted elements) are not written back to main memory where they wait for being reloaded in the next layer processing round, but instead are forwarded immediately to a consuming successor task that possibly runs on a different core. This will of course require some buffering in on-chip memory and on-chip communication of intermediate results where producer and consumer task are mapped to different cores, but multi-buffering is necessary anyway in order to overlap computation with (DMA) communication. It also requires that all merger tasks of the algorithm be active simultaneously; usually there are several tasks mapped to a core, which are dynamically scheduled by a user-level round-robin scheduler as data is available for processing.

However, as we would like to guarantee fast context switching on the cores, the limited size of on-chip memory then puts a limit on the number of buffers and tasks that can be mapped to a core, or correspondingly a limit on the size of data packets that can be buffered, which also affects performance. Moreover, the total volume of intermediate data forwarded on-chip should be low and, in particular, must not exceed the capacity of the on-chip interconnection network. Hence, we obtain a constrained optimization problem for mapping the tasks of streaming computations to the cores of the multiprocessor such that the resulting throughput is maximized.

In the following, we will describe optimal and approximative mapping algorithms for optimized on-chip pipelining of merge trees. Theoretically, the required memory bandwidth can be reduced by a factor proportional to the height of the merge tree. But an implementation on the real processor introduces overhead related to dynamic scheduling, buffer management, synchronization and communication delays. We also demonstrate how the optimal mapping algorithms can be used for other streaming applications, and how larger core counts can be handled.

As our exemplary target architecture we use the Cell Broadband Engine, see [CRDI07], with a PowerPC core and 8 parallel worker cores called SPEs, each with a small on-chip local memory (256 KB for both code and data), interconnected by a ring network called the Element Interconnect Bus (EIB). While we present our results for Cell, the techniques seem applicable to upcoming multicore processors as well, as the techniques are not specific for Cell. For example, we do not make any assumptions about the structure of the on-chip interconnection network. Furthermore, newer multicore processors face similar problems. For example, Intel just introduced the prototype of a 48-core multiprocessor, the SCC, see [Cor10] [HDV$^+$11], with four on-chip memory controllers to serve 48 cores, and limited local memory in the form of so-called message passing buffers (MPB). Also, Tilera processors (`www.tilera.com`) have a somewhat similar structure.

We detail an implementation of pipelined mergesort for Cell that actually achieves notable speedup of up to 70% on a QS20 and up to 143% on a PS3 over the best previous implementation. Also, the results support the hypothesis that on-chip pipelining as an algorithmic engineering option is worthwhile in general because simpler applications might profit even more.

The remainder of this article is organized as follows. Section 2 develops the on-chip pipelined merging algorithm. Section 3 presents several algorithms to compute optimal and approximative mappings of merge trees to the Cell SPEs. Section 4 gives a short overview of the Cell processor, as far as needed for this article, and presents technical details of the implementation. Section 5 reports on the experimental results. Further details are available in [Hul10]. Section 6 reviews related work. Section 7 concludes and identifies issues for future work.

2 On-chip pipelined mergesort

Parallel sorting is needed on every modern platform and hence heavily investigated. Several sorting algorithms have been adapted and implemented on Cell BE, which we will use as our exemplary target architecture. The highest performance is achieved by Cellsort, see [GBY07], and AAsort, see [IMKN07]. Both sort data sets that fit into off-chip main memory but not into local memories. Both implementations have similarities.

They 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 pN' that fit into the combined local memories of p cores 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 and as it accounts for the largest share of sorting time for larger input sizes (over 70% for inputs beyond 128 MBytes).

In CellSort, see [GBY07], this phase is realized by a bitonic sort because this avoids data dependent control flow and thus fully exploits the SPE's SIMD architecture. Yet, $O(N \log^2 N)$ memory accesses are needed. With 16 SPEs, the second phase takes 565 ms for 32M integers (128 MByte). When scaling from 2 to 16 SPEs, the speedup gained is only 3.91 for 0.5GByte of data, see [GBY07, Fig. 16].

In AAsort, see [IMKN07], mergesort with 4-to-1-mergers is used in the second phase. The data flow graph of the merge procedures thus forms a fully balanced merge quadtree. The nodes of the tree are executed on the cores layer by layer, starting with the leaf nodes. As each merge procedure on each core reads from main memory and writes to main memory, all N words are read from and written to main memory in each merge round, resulting in $N \log_4(N/(8N')) = O(N \log_4 N)$ data being read from and written to main memory. With 16 SPEs, the second phase takes about 150 ms for 16M key-value pairs, consisting of two integers (128 MByte). When scaling from 1 to 16 SPEs, the speedup gained is about 12, see [IMKN07, Fig. 13].

In [SB09], a global merge is performed as part of indexing on a Cell processor. There, each SPE implements a multiway merge so that a large number of blocks can be reduced in two stages to 8 blocks. The multiway merges are realized via tournament trees. The remaining 8 blocks are merged by a parallel merge implemented on 8 SPEs, where the blocks are split with the help of histogram information so that SPE i merges the i-th parts all blocks, and the resulting blocks can be concatenated. The authors attribute the speed of the global merge largely to the fact that in indexing, successive entries in blocks often contain identical values, so that the winner of the next tournament can be announced immediately. In general however, this will not be true in sorting. The throughput obtained in the global merging phase is 236 Mbyte/s (cf. Table 1 in [SB09]) for 128 bit tokens, i.e. for 128 MByte of data the merge would take about 540 ms.

Note that sorting fewer items of larger size, with constant size of the data set, typically reduces sorting time.

In order to decrease the bandwidth requirements to off-chip main memory and thus increase speedup, we use on-chip pipelining. This means that all merge nodes of all tree levels[1] are active from the beginning, and that results produced by a merge node are forwarded in packets to the consuming merge node directly without usage of main memory as intermediate store. Thus, when each merge node realizes a b-to-1 merge, and those nodes form a k-level complete b-ary tree, then b^k blocks from main memory are merged into one large block written back to main memory. To achieve this with AAsort, where 4-to-1 mergers read from and write to main memory, the amount of data to be transferred to and from main memory would be $k \cdot \log_4(b)$ times higher.

The decision to forward merged data streams in packets enables follow-up merge tasks to start work before predecessor mergers have handled their input streams completely, thus keeping as many merge tasks busy as possible, and allowing pipeline depths independent of the lengths of data streams. Note that already the mergers in the AAsort algorithm, see [IMKN07], must work with buffering and packets.

The requirement to keep all tasks busy is complicated by the fact that the processing of data streams is not completely uniform over all tasks but depends on the data values in the streams. A merger node may consume only data from one input stream for some time, if those data values are much smaller than the data values in the other input streams. Hence, if all input buffers for those streams are filled, and the output buffers of the respective predecessor merge tasks are filled as well, those merge tasks will be stalled. Moreover, after some time the throughput of the merger node under consideration will be reduced to the output rate of the predecessor merger producing the input stream with small

[1] We assume here that the complete tree can be mapped. For a detailed discussion see Subsect. 3.3.

data values, so that follow-up mergers might also be stalled as a consequence. Larger buffers might alleviate this problem, but are not possible if too many tasks are mapped to one core.

Finally, the merger nodes should be distributed over the cores such that two merger nodes that communicate data should be placed onto the same core whenever possible, to reduce communication load on the on-chip interconnection network. As a secondary goal, if they cannot be placed onto the same core, they might be placed such that different parts of the on-chip interconnection network might be used in parallel. However, we do not consider this in the research presented here.

3 Algorithms for Mapping Merge Trees to Multiple Cores

In this section we present several optimal and approximative algorithms that compute mappings with such properties. We begin by formalizing the problem in Section 3.1 and derive lower bounds for the optimization goals in Section 3.2. The height of the merge trees that we consider for on-chip pipelined merging is a tuning parameter of our mapping algorithms. For simplicity of computational load balancing, it could be chosen to match the number of cores used, which is also done in our implementation. In the remainder of this work we focus on this standard case; Section 3.3 discusses in general how we could adapt the mapping techniques to also cover the other cases if needed. Section 3.4 presents an integer linear programming (ILP) model for computing Pareto-optimal mappings, which can, in practice, yield optimal solutions for merge trees of height up to 7. We have chosen ILP over evolutionary approaches because the latter in general do not guarantee to reach an optimum, and also often do not provide advantages in optimization time. For mapping larger trees, we present a divide-and-conquer based approximation algorithm in Section 3.5 and an iterative approximation algorithm in Section 3.6.

3.1 Definitions

We model the target processor as a set $P = \{P_1, \ldots, P_p\}$ of p worker cores connected by an on-chip interconnection network, see Figure 1. The exact type of interconnection (such as multiple ring networks in the case of the Cell processor) is not considered here, as on application level each task can communicate with each other, no matter where they are placed. Thus, in our model, the cores can be considered completely interconnected.

The application is modelled by a k-level balanced b-ary tree $T = (V, E)$ directed towards its root, to be mapped onto the cores. Data flows in the tree from the leaves towards the root, input being fed in at the leaves and output leaving the tree root. Each node (task) v in the tree processes b designated

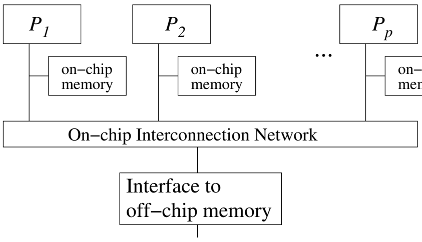

Figure 1: Model of a multi-core processor with explicitly accessible on-chip local memory for each core.

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 the core 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* $\beta(v)$ that a node v will place on the processor it is mapped to may be assumed to be a constant value $c = 1$ in a simplified setting, assuming the node needs a fixed amount for buffering transferred data and for the internal data structures it uses for processing the data. Yet, in order to optimize performance, buffer sizes and data structures may be task-specific, so that the memory load varies from node to node. In particular, the buffer requirements depend on the mapping itself, as inter-core data transfers require extra buffer space compared to intra-core transfers. Hence, we distinguish between two memory load models: The *simplified memory load model* that charges a fixed memory load c per node, and the *mapping-sensitive memory load model* that follows the actual implementation more closely but leads to more complex optimization problem instances.

We construct a mapping $\mu : V \rightarrow P$ of tree nodes to cores. As we consider the cores as symmetric and do not take their interconnection network into account, we can also view μ as a partitioning of the tree nodes into p partitions named P_1 to P_p.

Under this mapping μ, a core P_i has *computational load*[2]

$$C_\mu(P_i) = \sum_{v \in \mu^{-1}(P_i)} \gamma(v),$$

i.e. the sum of the load of all nodes mapped to it, and it has *memory load*

$$M_\mu(P_i) = \sum_{v \in \mu^{-1}(P_i)} \beta(v)$$

which is $c \cdot |\mu^{-1}(P_i)|$ in the simplified setting.

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 cores shall be minimized. This requirement is obvious, because the lower the maximum computational load, the more evenly the load is distributed over the cores. 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 cores shall be minimized. The maximum memory load is roughly proportional to the number of the buffers. As the amount of memory per processor is fixed, the maximum memory load determines the buffer size on this processor. If the buffers are too small, communication performance will suffer.

3. 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 cores, shall be low. Note that so far we do not take into account the length of the communication links on the on-chip interconnection network.

4. As often as possible, sibling nodes (nodes u and v with a common successor w, i.e. where $(u, w) \in E$ and $(v, w) \in E$) should be mapped to the same processor.

   As explained at the end of Sect. 2, a merger should deliver merged data buffers at an actual output rate that does not significantly fall short of the average output rate. A drop in the output rate may be caused by phases of unequal distribution of data in the input sequences, such that a merger processes mainly input data coming from one subtree only, which effectively stalls the other subtree(s). If sibling merger nodes are mapped to the same processor, a stall of one sibling node leaves a larger share of processor time to the busier sibling(s). Thus, a more balanced overall output rate of the siblings can be maintained.

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

While graph partitioning also tries to balance the computational load and the memory load, it does not care for memory load and placement of siblings. Furthermore, the algorithms to construct mappings presented in the sequel provide some guarantees on the results achieved, while graph partitioning algorithms are heuristics without guarantees.

3.2 Lower bounds

First, we provide lower bounds on computational load and memory load:

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 $\lceil c \cdot (b^k - 1)/((b-1)p)\rceil$, under the simplifying assumption that each merge node contributes a fixed memory load $c > 0$. If $p = k$, the maximum memory load is at least $\lceil c \cdot (b^k - b)/((b-1)(k-1))\rceil$.*

Consider the straightforward mapping μ_0 where $k = p$ and where all nodes of level i are mapped onto processor P_i. Obviously, each processor has computational load 1, so that the maximum computational load meets the lower bound. As a further plus, siblings are always mapped to the same processor. However, the communication load is maximized as child and parent are always mapped to different cores. Also, assuming a fixed memory load per node of $c = 1$, the maximum memory load is reached on processor P_p with b^{k-1} nodes of level $k-1$ mapped to that processor. Hence $M^*_{\mu_0} = c \cdot b^{k-1}$ and thus a factor of about $k/2 \leq k(b-1)/b \leq k$ away from the lower bound of Lemma 1. This last restriction is serious because of the inverse relationship between buffer count and packet size, i.e. communication performance.

3.3 Tree size and processor count

The computational load of a k-level tree is $k \cdot \gamma(r) = k$ if the root r has computational load 1, i.e. if the root task gets a core for itself. Then, for $p = k$ each processor will receive computational load 1 in the balanced case. However, in the sorting application, k is defined by the size of the data set, and independently of p. Therefore, we also consider $k > p$ and $k < p$.

If $k > p$, then we may split the tree into subtrees with $k' \leq p$ levels each, and process those subtrees one by one. Each subtree then can be mapped by one of the other two cases.

If $k < p$, then we map the tree onto $p' = k$ *pseudo*-cores, and implement each pseudo-core with p/k cores by evenly distributing the nodes assigned to that pseudo-core. If fewer than p/k nodes are mapped to a core (e.g. if the root is mapped separately), then we use a technique already known (see e.g. [JáJ92]) and mentioned in [IMKN07]: we partition the very large data blocks and perform merges on the partitions in parallel.

Mappings for the case $k = p$ will be treated in detail in the following subsections. However, for larger chip-multiprocessors, e.g. with $p \geq 30$, even the case $k = p$ might lead to problems because the tree gets very large (more than 10^9 nodes for $k = p = 30$ and $b = 2$, i.e. more than 10^7 nodes per core). If this happens, we split the tree into subtrees with $k' \ll k$ levels each, where k' is chosen such that the size of the subtrees is small enough to be mapped. Each of those subtrees is then mapped as in the case $k < p$, and the subtrees are processed one by one as in the case $k > p$.

3.4 Pareto-optimal mapping by integer linear programming

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 cores, where $p = k$.

Our integer linear programming (ILP) formulation (see Figure 2 for an overview) 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 cores, will be given by x:

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

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

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

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

Also, we use an integer variable *maxMemoryLoad* that will indicate the maximum memory load assigned to any core in P, a real-valued variable *commLoad* that will indicate the amount of communication over the on-chip network, and integer variable *nSiblingsOnDiffCores* that will indicate the total number of inner nodes whose children are not all mapped to the same core.

Given:
b-ary k-level merger tree with nodes $V = \{1, 2, ..., (b^k - 1)/(b - 1)\}$ to be mapped to k cores. Moreover, tuning weights $\epsilon_M \geq 0$, $\epsilon_C > 0$, $\epsilon_S > 0$.

Binary variables:
$x_{v,q} = 1$ iff node v mapped to core q.
$z_{u,q} = 1$ iff node $u > 1$ and its parent are mapped to core q.
$y_{u,q} = 1$ iff all children $b(u - 1) + 2, \ldots, b \cdot u + 1$ of node u mapped to core q.

Other variables:
$\mathit{maxMemoryLoad} \geq 0$, $\mathit{nSiblingsOnDiffCores} \geq 0$ (integer); $\mathit{commLoad} \geq 0$ (real).

Objective function:

$$\text{Minimize} \quad \epsilon_M \cdot \mathit{maxMemoryLoad} + \epsilon_C \cdot \mathit{commLoad} + \epsilon_S \cdot \mathit{nSiblingsOnDiffCores} \tag{1}$$

Constraints:

$$\forall v \in V: \quad \sum_{q \in P} x_{v,q} = 1 \qquad\qquad \forall q \in P: \quad \sum_{v \in V} x_{v,q} \cdot \gamma(v) \leq \gamma(r) = 1 \tag{2}$$

$$\forall v \in V_{inner},\ q \in P,\ i \in \{1, ..., b\}: \quad z_{b(v-1)+i+1,q} \leq x_{v,q} \tag{3}$$

$$\forall v \in V_{inner},\ q \in P,\ i \in \{1, ..., b\}: \quad z_{b(v-1)+i+1,q} \leq x_{b(v-1)+i+1,q} \tag{4}$$

$$\mathit{commLoad} = \sum_{v \in V - \{1\}} \tau(v) - \sum_{v \in V_{inner}} \sum_{q \in P} \left(\sum_{1 \leq i \leq b} z_{b(v-1)+i+1,q} \right) \cdot \tau(bv) \tag{5}$$

$$\forall v \in V_{inner},\ q \in P,\ i \in \{1, ..., b\}: \quad y_{v,q} \leq x_{b(v-1)+i+1,q} \tag{6}$$

$$\mathit{nSiblingsOnDiffCores} \ = \ \sum_{v \in V_{inner}} \sum_{q \in P} (1 - y_{v,q}) \tag{7}$$

For the simplified memory load model:

$$\forall q \in P: \quad \sum_{v \in V} x_{v,q} \leq \mathit{maxMemoryLoad} \tag{8}$$

For the mapping-sensitive memory load model:

$$\forall q \in P: \quad \sum_{v \in V} 2x_{v,q} + \sum_{v \in V} (1 - z_{v,q}) \leq \mathit{maxMemoryLoad} \tag{9}$$

Figure 2: Summary of the ILP model as described in Section 3.4.

The following constraints must hold:

Each node must be mapped to exactly one core, and each core can be filled up to 100% with work[3]:[4]

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

The memory load should be balanced. When we use the simplifying assumption of constant memory load $c = 1$ per node, we request:

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

In the mapping-sensitive memory load model, we request instead:

$$\forall q \in P : \quad \sum_{v \in V} 2x_{v,q} + \sum_{v \in V} (1 - z_{v,q}) \leq \mathit{maxMemoryLoad}$$

because in our implementation described in the next section all nodes[5] use one (circular) buffer per input stream, and nodes with successors on a different core require an extra output buffer.

Communication cost occurs whenever an edge is not internal, i.e. its endpoints are mapped to different cores. A straightforward way of expressing which edges are internal would involve terms that are products of two x variables and thus the model would no longer be linear. Instead, we apply a common trick to avoid such products: We use additional slack variables z with the following constraints

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

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:

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

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

[4] Note that the requirement for strict computational load balancing is implicit in the latter equation. Recall that all tasks in a k-level tree together have computational load $k \cdot 100\%$ where $100\% = \gamma(1)$ is the work done by the root merger task and which exactly matches the computational capacity of one of the k cores. Hence, each mapping thus fills each core with exactly $\gamma(1)$ computational work.

[5] The root node is handled in a special way but is, for $k = p > 3$, never the bottleneck for the memory load.

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

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

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

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

Finally, the objective function is:

Minimize $\epsilon_M \cdot \mathit{maxMemoryLoad} + \epsilon_C \cdot \mathit{commLoad} + \epsilon_S \cdot \mathit{nSiblingsOnDiffCores}$

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 *nSiblingsOnDiffCores* as first optimization goal. The formulation above requires that $\epsilon_C > 0$ and $\epsilon_S > 0$.

Table 1: The Pareto-optimal solutions found with ILP for $b = 2$, $k = p = 5,\ 6,\ 7$, using the simplified memory load model ($c = 1$).

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	2.38	1.75	2.63	2.44	1.94	1.88	2.38	2.31	2.0

By varying $\epsilon_M = 1 - \epsilon_C$ and keeping $\epsilon_S \ll \epsilon_C$, two of the Pareto-optimal solutions can be found, namely one with the least possible *maxMemoryLoad* and one with the least possible *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. We use a very small ϵ_S to give the sibling placement optimization the least priority and not interfere with communication optimization. In order to find the remaining Pareto-optimal solutions that may exist in between the two mentioned above, one can use any fixed ratio ϵ_M / ϵ_C, and instead set a given minimum memory load to spend (which is integer) on optimizing for *commLoad* only:

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

We implemented the above ILP model in CPLEX 10.2, see [ILO07], 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 $k = 6$

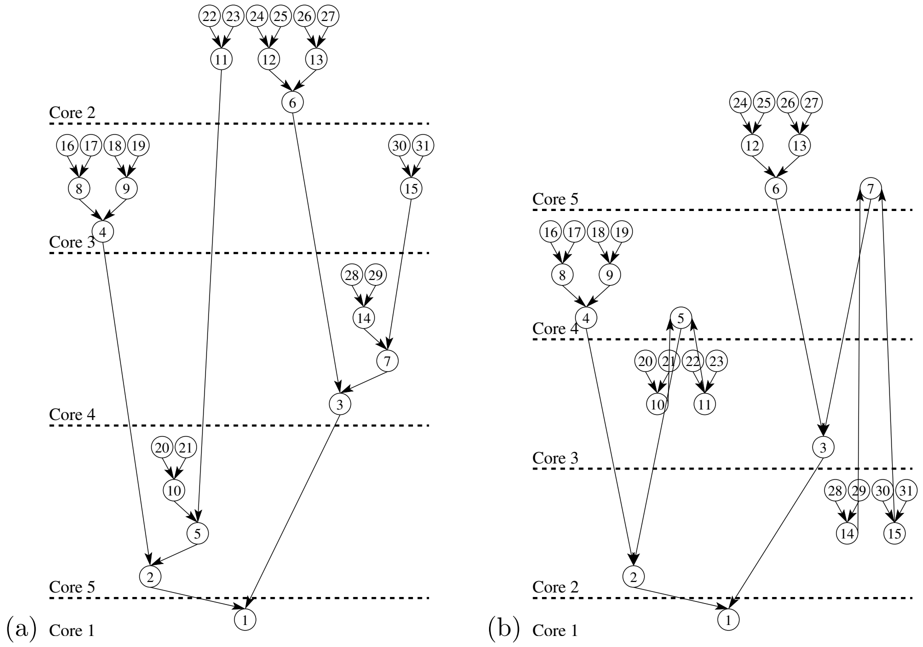

Figure 3: Two Pareto-optimal solutions for mapping a 5-level merge tree onto 5 cores, computed by the ILP solver in Section 3.4: (a) The maximum memory load is 10 communication buffers in the simplified memory load model and 22 in the mapping-sensitive model (as Core4 has 10 nodes with 2 input buffers each, and 2 of these have output buffers for inter-core forwarding) and communication load 1.75 (times the root merger's data output rate); (b) max. memory load 8 (simplified) and 18 (mapping-sensitive), and communication load 2.5. The (expected) computational load is perfectly balanced (1.0 times the root merger's load on each core) in both cases.

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, however including one with *maxMemoryLoad* of 37 (which matches the lower bound) and a *commLoad* of 2.78125, and one with 38 and 2.71875, respectively.

Figure 3 shows the generated tree drawings for two of the solutions for $k = 5$. The mapping computed for $k = 7$ with minimum *commLoad* is visualized in Figure 4.

Note that the only mergetree-specific part of the ILP model is the implicit arrangement of the edges via parents and and children. For taskgraphs of other

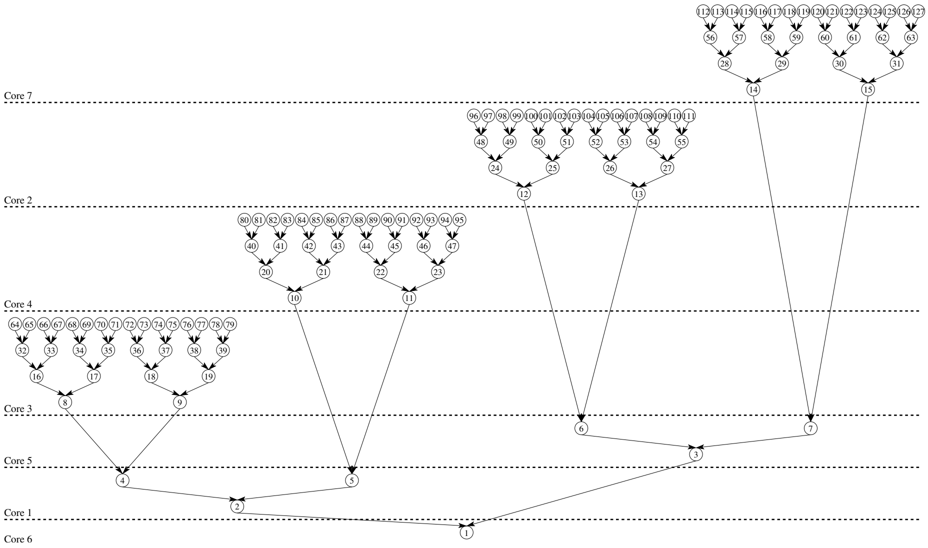

Figure 4: A Pareto-optimal solution for mapping a 7-level tree onto 7 cores with least communication load, computed by ILP (Sect. 3.4). The same mapping was also found by the IT-map heuristic (Sect. 3.6).

streaming applications, similar ILP models can be derived by making the constraints from edges explicit. We demonstrate this in [KK09] by mapping a data-parallel computation from a computational kernel and an FFT computation.

3.5 A divide-and-conquer based approximation algorithm

For larger values of k, where the ILP cannot compute an optimal mapping in an appropriate time, we can use the following divide-and-conquer algorithm (called *DC-map* in the sequel) which we present for $b = 2$.

To construct a mapping μ_2 for a k_1-level binary tree onto k_1 cores, 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 core, and interpret the remaining $k_1 - 1$ cores as two sets of $k_1 - 1$ cores, each with half the computational power. We map a $(k_1 - 1)$-level tree onto each set recursively. Then we sort the cores 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 cores from both lists into one core with full computational power.

By construction, DC-map produces a mapping where each core has an optimal 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 of about $b \cdot k/(k+1)$. 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.

If the mapping used for k_0 levels has a communication load of l, then the resulting mapping μ_2 for k_1 levels has a communication load of $l + k_1 - k_0$, as the root in each step is placed on a separate core.

DC-map does not take special care for the placement of siblings. 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.

We have implemented a prototype version of DC-map, which we evaluate 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 the DC-map prototype with simplified memory load model ($c = 1$)

	$k_0 = 3$					$k_0 = 7$				
k_1	4	5	6	7	8	8	9	10	11	12
$M^*_{\mu_2}$	6	8	15	24	46	42	84	132	236	453
lower b.	5	8	13	21	37	37	64	114	205	373
quotient	1.20	1.00	1.15	1.14	1.24	1.14	1.31	1.16	1.15	1.21
k_1/k_0	1.33	1.66	2.00	2.33	2.66	1.14	1.29	1.43	1.57	1.71

3.6 Iterative Approximation Algorithm for Mapping Merge Trees

In the following, we give an alternative approximation algorithm, *IT-map*, for mapping b-ary merge trees where the maximum memory load is by a factor at most b larger than the lower bound. Thus, for large trees with $k_1 > b \cdot k_0$, this approximation algorithm provides a better bound than the previous one.

The IT-map algorithm constructs a mapping μ_3 in several steps. Let $k_0 = k$ be the number of levels and cores in step 0. In step i, if $k_i \geq 2$, we map $l_i \leq k_i - 1$ of the k_i levels, starting from the leaves, onto a respective number of cores, so that $k_{i+1} = k_i - l_i$ levels and cores remain. If $k_i = 1$, we map the tree root onto the last core, and the mapping is complete. As each level of the tree has

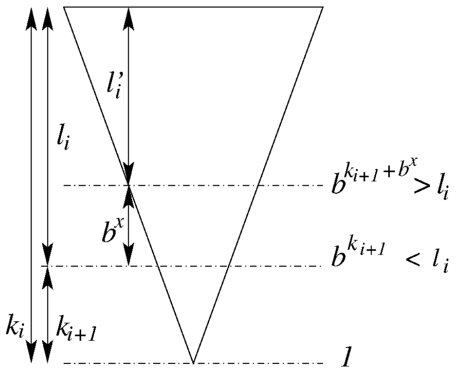

Figure 5: Step i of the construction of mapping μ_3 by the IT-map algorithm.

a computational load of 1, the mapping must be such that each core receives a load of 1 to minimize $C^*_{\mu_3}$.

We choose l_i to be the largest power of b less than or equal to $k_i - 1$. The l_i levels then consist of $b^{k_{i+1}}$ balanced b-ary trees of l_i levels each. If $l_i \leq b^{k_{i+1}}$, then l_i divides $b^{k_{i+1}}$ because it is also a power of b, and we map $b^{k_{i+1}}/l_i$ trees on each of the cores. This balances both maximum computational and maximum memory load.

The case $l_i > b^{k_{i+1}}$ is illustrated in Fig. 5. In this case, we can write $l_i = b^x \cdot b^{k_{i+1}}$, where $x \geq 1$ is an integral number. In this case, we define $l_i' = l_i - b^x$ and first map the l_i' levels starting from the leaves. Those levels consist of $b^{k_{i+1}+b^x}$ balanced b-ary trees of l_i' levels each. As $b^x \geq x$ because of $b \geq 2$ and $x \geq 1$, it follows that $b^{k_{i+1}+b^x} \geq b^{k_{i+1}+x} = l_i$ and that this number is even an integral multiple of l_i because l_i is also a power of b. Thus, we can map the trees of the last l_i' levels evenly onto the l_i cores. For the remaining b^x levels to be mapped in this step, we map those levels starting with the level closest to the root having $b^{k_{i+1}}$ nodes: we map each node onto one core, using $b^{k_{i+1}}$ cores. For the next level having $b \cdot b^{k_{i+1}}$ nodes, we map b nodes on each core, using another $b^{k_{i+1}}$ cores. When we have finished with those b^x levels, we have used $b^x \cdot b^{k_{i+1}} = l_i$ cores. Note that this straightforward placement corresponds to applying mapping μ_0 for the $b^{k_{i+1}}$ trees of $k = b^x$ levels each, with the core capacity scaled down to $b^{-k_{i+1}}$. We might also apply mapping μ_3 recursively to further balance the load.

On each core, we have placed a load of $l_i'/l_i = 1 - b^{-k_{i+1}}$ by mapping l_i' levels, and $b^{-k_{i+1}}$ by mapping the first b^x levels. It follows that the computational load on each core is 1. The maximum memory load is determined in step $i = 0$, because the majority of the nodes is mapped there. In this step $(b^k - b^{k-l_0})/(b-1)$

nodes are mapped onto l_0 cores, so that each core receives a memory load of

$$\frac{b^k - b^{k-l_0}}{(b-1)l_0} < b \cdot \frac{b^k - 1}{(b-1)k}$$

because $l_0 \geq k/b$. Thus, the memory load is larger than the lower bound by a factor less than b. Note that this is not completely exact because the b^x levels — if they are used in the first step — are not mapped with a completely even memory load. However, the imbalance is only very slight, as our simulations will show.

Each step except the last one contributes at least 1 to the communication load, as the edges leaving the l_i levels towards the next level cannot be internal. If $l_i \leq b^{k_{i+1}}$, then all edges within the l_i levels are internal, as complete subtrees are mapped, and thus this step contributes exactly 1 to the communication load. If $l_i > b^{k_{i+1}}$, then the same argument holds for the first l'_i levels. The following level is mapped onto $b^{k_{i+1}}$ cores, so that at a part of the edges entering this level are internal. The remaining levels $l_i - l'_i - 1$ are all mapped onto different cores, so that the edges connecting these levels all contribute to the communication load. If we define $l'_i = l_i$ for the case $l_i \leq b^{k_{i+1}}$, then the communication load cannot be larger than $k - 1 - \sum_i (l'_i - 1) = k - 1 + r - \sum_i l'_i$ with r being the number of steps.

In each step, there are at most two levels (the first one of the b^x and the first one of the l'_i) where siblings are not placed on the same core.

As in each step i the largest power of b less than k_i is chosen as the number l_i of levels mapped, the number r of steps made by the mapping algorithm is one plus the cross sum of $k-1$ in b-ary representation, and thus $r \leq 1 + (b-1) \cdot \log_b(k-1)$.

In Tab. 3, we present the placement results achieved for $b = 2$ and $k = 5, \ldots, 12$. We see that the maximum memory load is oscillating between the lower bound and b times the lower bound, in intervals ending at k being a power of $b = 2$. We also see that the communication load is rather low. In fact, the mappings computed by IT-map for $k = 5$ and $k = 7$ are Pareto-optimal, they correspond to the mappings shown in Figures 3(b) and 4, respectively.

4 Implementation on Cell/B.E.

4.1 Cell/B.E.: Overview and Implications of Limited On-Chip Memory

The Cell/B.E. (Broadband Engine) processor, see [CRDI07], is a heterogeneous multi-core processor consisting of 8 SIMD cores called SPE and a dual-threaded PowerPC core (PPE), which differ in architecture and instruction set. In earlier

Table 3: Results for the iterative approximation algorithm IT-map, based on the simplified memory load model ($c = 1$)

k	5	6	7	8	9	10	11	12
$M^*_{\mu_3}$	8	15	30	60	68	128	255	510
lower bound	8	13	21	37	64	114	205	373
Comm. load	2.5	2	2	3	4.5	3.5	2	3

versions of the Sony PlayStation-3$^{\text{TM}}$ (PS3), up to 6 SPEs of its Cell processor could be used under Linux. On IBMs Cell blade servers such as QS20 and later models, two Cells with a total of 16 SPEs are available. Cell blades are used, for instance, in the nodes of RoadRunner, which was the world's fastest supercomputer in 2008–2009.

While the PPE is a superscalar processor with direct access to off-chip memory via L1 and L2 cache, the SPEs are optimized for doing SIMD-parallel computations at a significantly higher rate and lower power consumption than the PPE. The SPE can issue up to 2 instructions in-order per clock cycle, which puts high requirements on the SPE compiler's code generator. The SPE datapaths and registers are 128 bits wide, and the SPE vector instructions operate on them as on vector registers, holding 2 doubles, 4 floats or ints, 8 shorts or 16 bytes, respectively. For instance, four parallel float comparisons between the corresponding sections of two vector registers can be done in a single instruction. However, branch instructions can slow down data throughput of an SPE because branch misprediction penalty is very high (24 to 26 cycles). The PPE should thus mainly be used for coordinating SPE execution (reduces power consumption), providing OS service and running control intensive code (branch penalty in SPEs).

Each SPE has a small local on-chip memory of 256 KBytes. This *local store* is the only memory that the SPE's processing unit (the SPU) can access directly, and therefore it needs to accommodate both SPU code and data. There is no cache and no virtual memory on the SPE. Access to off-chip memory is only possible by DMA *put* and *get* operations that can communicate blocks of up to 16KB size at a time to and from off-chip main memory. DMA operations are executed asynchronously by the SPE's memory flow controller in parallel with the local SPU; the SPU can initiate a DMA transfer and synchronize with a DMA transfer's completion. DMA transfer is also possible between an SPE and another SPE's local store. The DMA packet size cannot be made arbitrarily small; the absolute minimum is 16 bytes, and in order to be not too inefficient, at least 128 bytes should be shipped at a time. Reasonable packet sizes are a few KB in size.

There is no operating system or runtime system on the SPE except what is linked to the application code in the local store. This is what necessitates user-level scheduling if multiple tasks are to run concurrently on the same SPE.

SPEs, PPE and the memory interface are interconnected by the Element Interconnect Bus (EIB), see [CRDI07]. The EIB is implemented by four unidirectional rings with an aggregate bandwidth of 204 GByte/s (peak). Up to three non-overlapping connections are possible simultaneously on each ring. The bandwidth of each unit on the ring to send data over or receive data from the ring is only 25.6 GB/s. Hence, the off-chip memory tends to become the performance bottleneck if heavily accessed by multiple SPEs.

To allow for overlapping DMA handling of packet forwarding (both off-chip and on-chip) with computation on Cell, an SPE must either process enough streaming tasks, or there should be at least buffer space for 2 input packets per input stream and 2 output packets per output stream of each streaming task to be executed on an SPE. While the SPU is processing operands from one buffer, the other one in the same buffer pair can be simultaneously filled or drained by a DMA operation. Then the two buffers are switched for each operand and result stream for processing the next packet of data. (Multi-buffering extends this concept from 2 to an arbitrary number of buffers per operand array, ordered in a circular queue.) This would amount to at least 6 packet buffers for an ordinary binary streaming operation, which need to be accommodated in the size-limited local store of the SPE. Hence, the size of the local store part used for buffers puts an upper bound on the buffer size and thereby on the size of packets that can be communicated.

As the size of SPE local store is severely limited (256KB for both code and data) and the minimum packet size is the same for all SPEs and throughout the computation, the maximum number of packet buffers of the tasks assigned to any SPE should be as small as possible. Another reason to keep packet size large is the overhead due to switching buffers and user-level runtime scheduling between different computational tasks mapped to the same SPE. Figure 6 shows the sensitivity of the execution time of our pipelined mergesort application (see later) to the buffer size.

4.2 Implementation Details

We have implemented the merge phase for a binary merge tree for the case $k = p$ on the Cell processor. We provide details about peculiar aspects of our implementation.

4.2.1 Merging kernel

SIMD instructions are being used as much as possible in the innermost loops of the merger node. Merging two (quad-word) vectors is completely done with

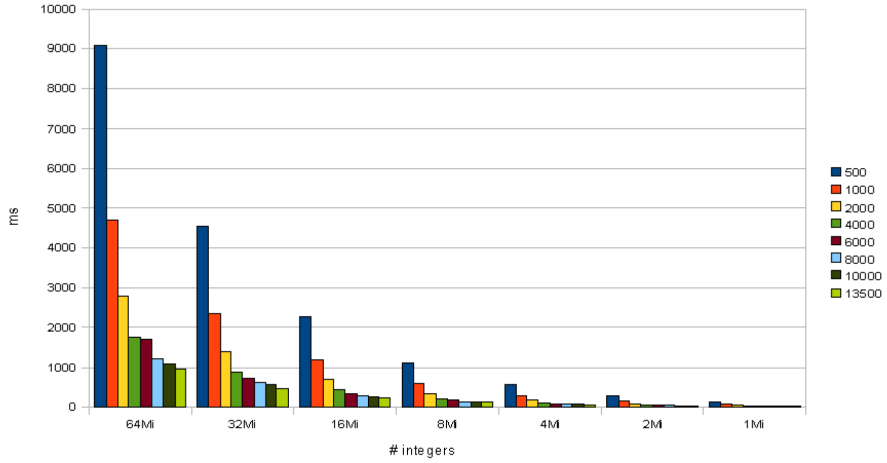

Figure 6: Merge times (here for a 7-level merger tree pipeline, cf. Sect. 4), shown for various input sizes (number of 128bit-vectors per SPE), strongly depend on the buffer size (see right legend) used in multi-buffering.

SIMD instructions as in CellSort, see [GBY07]. In principle, it is possible to use only SIMD instructions in the entire merge loop, but we found that it did not reduce time because the elimination of an if-statement required too many comparisons and forced to duplicate data in different vector positions.

4.2.2 Mapping optimizer

The mapping of merger task nodes to SPEs is read in by the PPE from a text file generated by the mapping optimizer. The PPE generates the task descriptors for each SPE at runtime, so that our code is not constrained to a particular merge-tree, but still optimized to the merge-tree currently used. The mappings are precomputed, either by integer linear optimization for smaller trees, or by approximative solutions for larger trees.

4.2.3 SPE task scheduler

Tasks mapped to the same SPE are scheduled by a user-level scheduler in a round-robin order. A task is ready to run if it has sufficient input and an output buffer is free. A task runs as long as it has both input data and space in the output buffer, and then initiates the transfer of its result packet to its parent node and returns control to the scheduler loop. If there are enough other tasks to run afterwards, DMA time for flushing the output buffer is masked and hence only one output buffer per task is necessary (see below). Tasks that are not data-ready are skipped.

As the root merger is always alone on its SPE, no scheduler is needed there and many buffers are available; its code is optimized for this special case.

4.2.4 Buffer management

Because nodes (except for the root) are scheduled round-robin, the DMA latency can, in general, be masked completely by the execution of other tasks, and hence double-buffering of input or output streams is not necessary. If the latency cannot be masked, the task scheduler skips the tasks using the corresponding buffers (see above) and thus protects the sending buffer from being overwritten. An output stream buffer is only used for tasks whose parents/successors reside on a different SPE. Each SPE has a fixed sized pool of memory for buffers that gets equally shared by the nodes. This means that nodes on less populated SPEs, for instance the root merger that has a single SPE on its own, can get larger buffers.

4.2.5 Communication

Data is pushed upwards the tree, i.e. producers/child nodes initiate cross-SPE data transfer and consumers/parent nodes acknowledge receipt. The (system-global) addresses of buffers in the local store on the opposite side of cross-SPE DMA communications are exchanged between the SPEs in the beginning.

4.2.6 Synchronization

Each buffer is organized as cyclic buffer with a head and a tail pointer. A task only reads from its input buffers, i.e. only updates the tail pointers. A child node only writes to its parent's input buffers, i.e. only writes to the head pointer. The parent task updates the tail pointer of the input buffer for the corresponding child task; the child knows how large the parent's buffer is and how much it has written itself to the parent's input buffer so far, and thus knows how much space is left for writing data into the buffer. This assumption is conservative, i.e. safe. This means that no locks are needed for the synchronization between nodes.

4.2.7 DMA tag management

A SPE can have up to 32 DMA transfers in flight simultaneously and uses tags to distinguish between these when polling the DMA status. If an SPE hosts many leaf tasks reading from main memory, it has many buffers used for remote communication and thus might run out of tags. If that happens, the tag-requesting task gives up, steps back into the task queue and tries to initiate that DMA transfer again when it gets scheduled next.

5 Experimental results

We used a Sony PlayStation-3 (PS3) with IBM Cell SDK 3.0 and an IBM blade server QS20 with SDK 3.1 for the measurements. We evaluated the largest data sets that could fit into RAM on each system, which means up to 32Mi integers on PS3 (6 SPEs, 256 MiB RAM) and up to 128Mi integers on QS20 (16 SPEs, 1GiB RAM). The code was compiled using gcc version 4.1.1 and run on Linux kernel version 2.6.18-128.e15.

A number of blocks equal to the number of leaf nodes in the tree to be tested were filled with random data and sorted. This corresponds to the state of the data after the local sorting phase (phase 1) of CellSort, see [GBY07]. Ideally, each such block would be of the size of the aggregated local storage available for buffering on the processor. CellSort sorts 32Ki (32,768) integers per SPE, blocks would thus be 4 × 128KiB = 512KiB on the PS3 and 16 × 128KiB = 2MiB on the QS20. For example, a 6-level tree has 64 leaf nodes, hence the optimal data size on the QS20 would be 64 × 512KiB = 32MiB. However, other block sizes were used when testing in order to magnify the differences between mappings. Each experiment was repeated 10 times, and the arithmetic mean of the runtimes computed. The standard deviation always was less than 10% of the mean, i.e. rather small.

5.1 On-chip-pipelined merging times

The resulting times with on-chip pipelining for 5-level and 6-level trees on PS3 are shown in Fig. 7 for various ILP-optimized mappings, using the simplified memory load model. Figure 8 gives a more detailed analysis of the SPE execution time for some ILP-optimized mappings on PS3 for an input size of 16Mi integers. The figure reveals that, for such large input sizes, on-chip pipelined mergesort is still memory-bound. Instead, for tiny input sizes such as 1Mi (not shown) the buffer capacity is sufficient to overlap almost all DMA waiting with merging.

For QS20, mappings generated for the mapping-sensitive model with $\epsilon_M =$ 0.01, 0.1, 0.5, 0.9 and 0.99 and $\epsilon_C = 1 - \epsilon_M$ were tested on different data sizes and merger tree sizes from $k = 5$ to $k = 8$, see Figs. 9 and 10 for the results[6]. We see that the choice of the mapping can have a major impact on merging time, as even mappings that are optimal for different optimization goals exhibit timing differences of up to 20%.

The model itself has only limited effect: The best mappings computed with the simplified model (reported in [HKK10]) were slightly better on QS20 for $k = 5$ (by a few percent) and performed for $k = 6$ and $k = 7$ equally well as the best mappings for the mapping-sensitive model. Yet, we observed slightly larger

[6] For $k = 8$, the ILP solver always timed out and the approximative solutions reported are not always valid, see our remarks above.

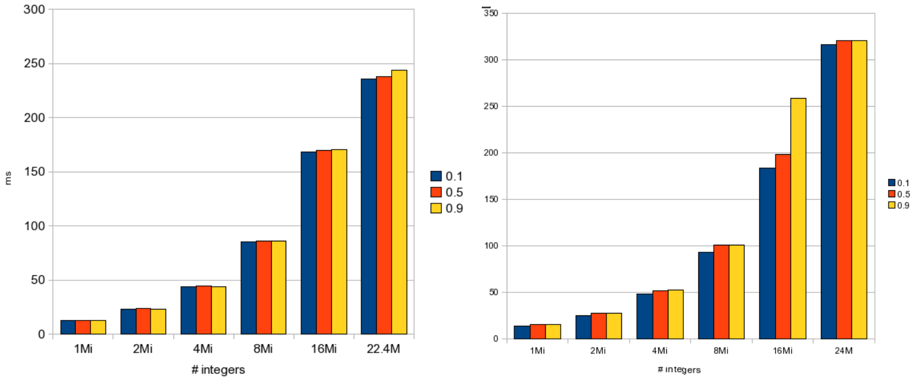

Figure 7: Merge times for $k = 5$ (left) and $k = 6$ (right) for different ILP-optimized mappings (ϵ_M) based on the simplified memory load model, on PS3.

Table 4: Timings for the CellSort global merging phase vs. Optimized on-chip-pipelined merging for global merging of integers on QS20

k	#ints	CellSort Global Merging	On-Chip-Pipelined Merging	Speedup
5	16Mi	219 ms	174 ms	1.26
6	32Mi	565 ms	350 ms	1.61
7	64Mi	1316 ms	772 ms	1.70

variations in performance across the mappings for each k with the simplified model, up to 25%. However, the (valid) approximations obtained at timeout for $k = 8$ were better with the simplified model.

5.2 Results of DC-map

Using the DC-map algorithm from Subsect. 3.5, mappings for trees for $k =$ 8, 7 and 6 were constructed by recursive composition using optimal mappings (computed with the ILP algorithm with $\epsilon_M = 0.5$) as base cases for smaller trees. Fig. 11 shows the result for merging 64Mi integers on QS20.

5.3 Comparison to CellSort

Figure 12 shows the total time for mergesort on QS20, partitioned into the time for the local sorting phase and the global merging phase; it is the latter that we optimize in this work. The time for the merging phase exceeds the time for the local sorting phase from a data size of 8 Mi integers and increasingly dominates for larger data sets.

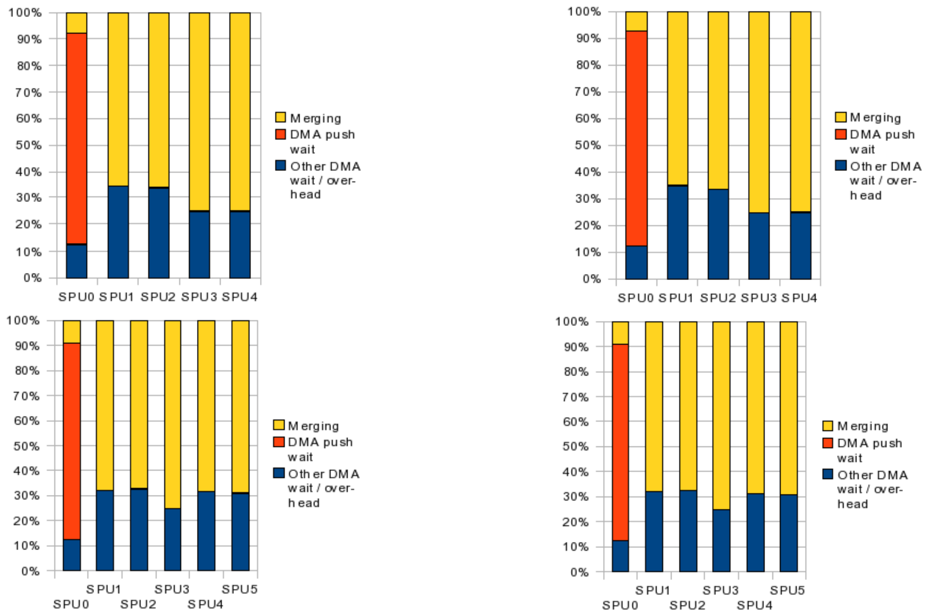

Figure 8: Distribution of merging time (yellow), non-overlapped DMA-push wait time (red) and other non-overlapped DMA wait and overhead time (blue), in percent of each SPE's runtime, of a 5-level tree (upper diagrams) and a 6-level tree (lower diagrams) each with two ILP-generated mappings ($\epsilon_M = 0$ and 0.001), each merging 16M integers on PS3. In all four mappings, SPE0 holds the root merger. Except for tiny problem sizes such as 512K integers (not shown here) where merging always accounts for more than 94% of the time even on SPE0, the application is still memory bound; in particular, SPE0 spends most of its time waiting for the full output buffers to be flushed to the saturated off-chip memory and thus cannot overlap merge time with DMA wait time.

Table 4 shows the direct comparison between the global merging phase of CellSort (which is dominating overall sorting time for large data sets like these) and on-chip-pipelined merging. We achieve significant speedups for on-chip-pipelining in all cases; the best speedup of 70% can be obtained with 7 SPEs (64Mi elements) on QS20, using the mapping with $\epsilon = 0.01$ in Fig. 9; the corresponding speedup figure for the PS3 is 143% at $k = 6$, 8Mi elements. This is due to less communication with off-chip memory.

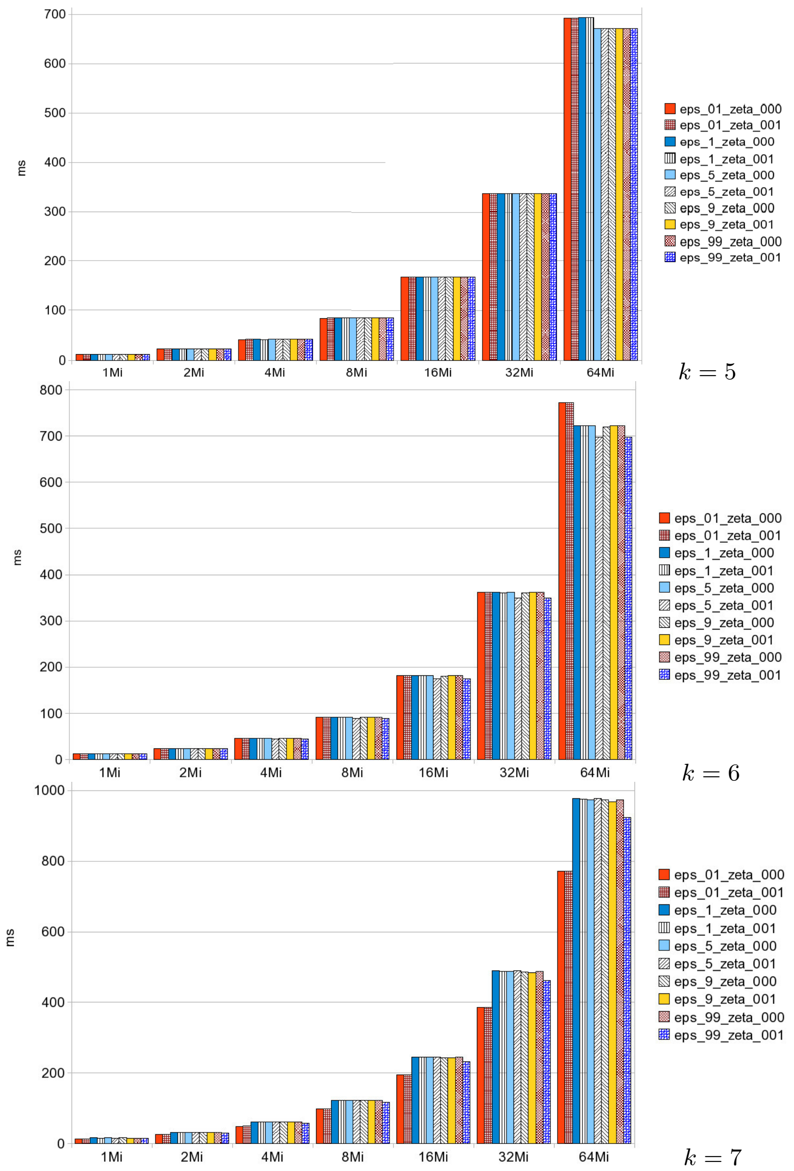

Figure 9: Merge times for $k = 5, 6, 7$ and different input sizes on QS20, using ILP-optimized/approximated mappings (controlled by $\epsilon = \epsilon_M = 1 - \epsilon_C$ between 0.01 and 0.99 and offset ζ for sibling-criterion weight ϵ_S either 0.0 or 0.001) based on the mapping-sensitive memory load model. For $k = 7$, the best mapping ($\epsilon_M = 0.01$) is, apart from symmetries, equal to the one displayed in Figure 4; it is communication-optimal.

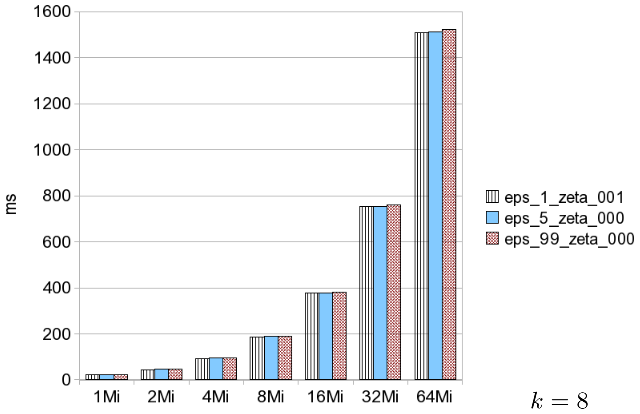

Figure 10: Merge times for $k = 8$ and different input sizes on QS20, using three valid ILP-approximated mappings found for $\epsilon = \epsilon_M = 1 - \epsilon_C$ set to $0.1, 0.5$ and 0.99, and ζ as above, for the mapping-sensitive model.

5.4 Discussion

Different mappings yield some variation in execution times, although there is no clear trend among the ILP generated Pareto-optimal mappings saying whether preference for memory load (ϵ_M) or communication load is more important; one explanation is that, as the executable only occupies a few KB in the local store, there is still some slack in the tolerable buffer sizes and thus in the maximum memory load, as can be seen from Figure 6, making memory load a less critical factor for smaller trees. Also, not all details of the implementation are covered by our models.

In the cases observed, mappings generated by DC-map perform not much worse than ILP-optimized mappings, and can even be better for large trees (here, for $k = 8$) where ILP only can deliver coarse approximations after hitting the timeout. Also, starting from a larger base case tree size k_0 does not always lead to a better mapping.

Interestingly, the best mapping for $k = 7$ on QS20 (Figure 4) as well as the best mapping for $k = 6$ on PS3 are, apart from symmetries, identical to mappings generated by the iterative approximation algorithm IT-map. Both mappings happen to be optimal ones with respect to communication load.

Also with on-chip pipelining, using deeper tree pipelines (to fully utilize more SPEs) is not always beneficial beyond a certain depth k, here $k = 6$ for PS3 and $k = 7$ for QS20, as a too large number of tasks increases the overhead of on-chip pipelining (smaller buffers, scheduling overhead, tag administration, synchronization, communication overhead). The overall pipeline fill/drain overhead is

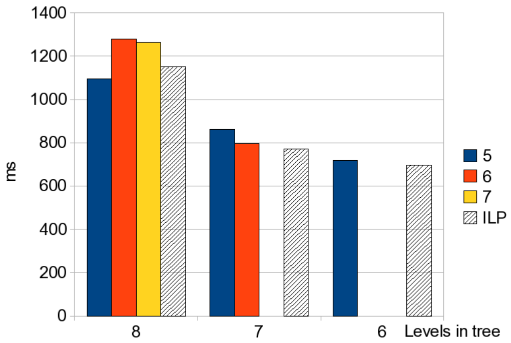

Figure 11: Merge times (64 Mi integers) for trees ($k = 6, 7, 8$) with mappings constructed from ILP-optimized mappings of smaller trees ($k_0 = 5, 6, 7$) using DC-map. As a reference, the best ILP optimized mapping (which is only an approximation for $k = 8$, here one determined with the simplified model was best) is also shown.

more significant for lower workloads but negligible for the larger ones. In general, the above upper limit might be higher e.g. because of larger local memories, although it necessarily exists, as a k-level binary tree has $2^k - 1$ nodes, so one of the k cores must host at least $2^k/k$ tasks, which seems unrealistic for $k \gg 20$. As the ILP approach will take very long runtimes for $k > 10$, the approximation algorithms will cover the range from that point upwards.

From Fig. 6 it is clear that, with optimized mappings, buffer size may be lowered without losing much performance, which frees more space in the local store of the SPEs, e.g. for accommodating the code for all phases of CellSort, saving the time overhead for loading in a different SPE program segment for the merge phase.

6 Related Work

Partitioning and mapping of task graphs is, in general, an NP-complete problem and has been discussed a lot in the literature.

One application area is, as in our case, the parallelization of programs with given dependence graph for execution on a (mostly, shared memory) parallel computer, with the objective to balance the work load of the partitions, minimize the number of partitions (aka. *processor minimization*), and/or minimize the overall weight of all edges cut by the partitioning, as all these are supposed to correspond to expensive shared memory accesses (aka. *bandwidth minimization*).

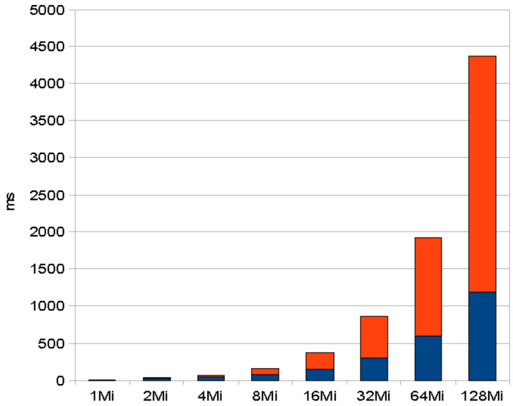

Figure 12: Distribution of overall mergesort times (for 1 to 128 Mi integers) with CellSort on QS20. The lower (blue) part of each column represents the time for the local sorting phase. The upper (red) part is the time for global merging, which we improve by this work.

Another related area is the (spatial) clustering of logic circuits into partitions each matching a maximum chip size constraint, while the communication between partitions must fit an upper limit on the number of pins per chip. Here, one is (as in our case) mainly interested in reducing the accumulated weight of all edges cut between any two adjacent partitions (aka. *bottleneck minimization*).

There is a wealth of literature on mapping and scheduling acyclic task graphs of streaming computations to multiprocessors. Some methods are designed for special topologies, such as linear chains and trees, while others address general task graphs; only few of them deal with multi-criteria optimization

6.1 Mapping of special topologies

For *tree-shaped task graphs*, various partitioning algorithms have been proposed.

[Bok88] considers partitioning of trees for master-slave (there called host-satellite) systems where the partition containing the root is mapped to the master (host) processor while the slaves (satellites) are each assigned exactly one complete subtree that is connected directly to the master partition.

[RJ94a] show that the bandwidth minimization problem is NP-complete even for trees, and give a fast heuristic algorithm for it. In a follow-up paper [RJ94b] give polynomial-time greedy algorithms for bottleneck minimization and processor minimization of tree task graphs.

Most approaches for tree partitioning are for non-pipelined trees and therefore assume that the tree partitions should be connected components (i.e., contiguous subtrees) and exactly one partition be mapped to one processor. This does not apply in our case, where partitions can consist of several disconnected subtrees, so that processors could be better "filled up" to their computational capacity with residual tree fragments if this improves system throughput. Also, in our scenario the b-ary tree is always complete, thus we can exploit symmetry properties when precomputing the mappings.

[LMRT96] consider the problem of mapping a tree that evolves in a search problem onto a distributed memory parallel computer in such a way that computation and communication times both are minimized. They focus on trees that evolve dynamically, i.e. are not known beforehand as in our case. The work associated with each tree node seems to be constant while the computational load in our case depends on the tree level of the node. As the tree is not kept completely, memory load plays a minor role. In contrast, we map a tree to be kept completely in memory. Finally, the trees considered in search problems typically are far from balanced and their degree is irregular, while we consider balanced b-ary trees.

[MLZ99] consider non-pipelined, tree-like task graph structures such as reduction trees, task graphs for parallel prefix computations and *Butterfly graphs*, under the LogP cost model that accounts for transfer latency and limited communication bandwidth in message passing systems. They give polynomial-time algorithms for computing optimal schedules for special cases. However, memory constraints or pipelined versions of these task graphs are not considered.

Melot et al. [MKA$^+$12] use integer linear programming to map forests of 6-level merge trees to the 2D mesh on-chip network of the 48-core Intel Single-Chip Cloud Computer (SCC), where both the network distance between communicating tasks and the network distance between an off-chip memory-accessing task and its closest memory interface are relevant for the quality of a mapping.

6.2 Mapping of general task graphs

In contrast to specific regular topologies such as trees that can be described completely by very few parameters, mapping of general task graphs allows for more irregular structures, which requires an explicit representation and makes it harder to exploit symmetries in the solution of the mapping problem. This is a main difference between the merge tree case study in this article and related approaches discussed in the following. For a generalization of our ILP model by explicit modeling of general pipelined task graphs, such as data flow graphs of data-parallel operations, see [KK09]; in that work we also provide faster heuristics for the general case, such as a divide-and-conquer based heuristic method that uses the ILP model to partition both the task graph and the underlying target platform into several parts that are mapped separately.

The approaches for mapping general task graphs can be roughly divided into two classes: Non-overlapping scheduling and overlapping scheduling. Our approach belongs to the latter.

Non-overlapping scheduling schedules a single execution of the program (and repeats this for further input sets if necessary); it aims at minimizing the makespan (execution time for one input set) of the schedule, which depends strongly on task and communication latencies, while memory constraints are usually a non-issue here. A typical result is that all tasks on a critical path are mapped to the same processing unit. The mapping and scheduling can thus be done by classical list-scheduling based approaches for task graph clustering that attempt to minimize the critical path length for a given number of processors. Usually, partitions are contiguous subgraphs. The problem complexity can be reduced heuristically by a task merging pre-pass that coarsens the task granularity. See [KB06] for a recent survey and comparison.

[SK01] propose a heuristic method for memory-aware assignment and scheduling of a task graph to a bus- or link-connected set of processing units. Tasks' memory requirements for code and data are parameterized. Edges between tasks in different partitions are parameterized by the buffer requirements for sender and receiver. Based on initial estimations for maximum data memory use, this iterative optimization method toggles between two strategies for assignment and scheduling, namely critical path scheduling (which optimizes for the makespan) and scheduling for minimization of memory usage, trying to balance execution time and memory utilization of the resulting solution.

For Cell BE, [BLMR08] propose a constraint programming approach for combined mapping, scheduling and buffer allocation of non-pipelined task graphs to minimize the makespan.

Overlapping scheduling, which is closely related to *software pipelining* and to *systolic parallel algorithms*, see [Kun82], instead overlaps executions for different input sets in time and attempts to maximize the throughput in the steady state, even if the makespan for a single input set may be long. Mapping methods for such pipelined task graphs, especially for signal processing applications in the embedded systems domain, have been described e.g. in [HR93] and [RGB$^+$08]. Our method also belongs to this second category.

[HR93] work on a hierarchical task graph such that task granularity can be refined by expanding function calls or loops into subtasks as appropriate. They provide a heuristic algorithm based on greedy list scheduling for simultaneous pipelining, parallel execution and retiming to maximize throughput. The resulting mapped pipeline is a linear graph where each pipeline stage is assigned one or several processors. Buffer memory requirements are considered only when checking feasibility of a solution, but are not really minimized for. The method only allows contiguous subDAGs to be mapped to a processor.

[RGB$^+$08] decompose the problem into mapping (resource allocation) and scheduling. The mapping problem, which is close to ours, is solved by an integer linear programming formulation, too, and is thus, in general, not constrained to partitions consisting of contiguous subDAGs as in most other methods. Their framework targets MPSoC platforms where the mapped partitions form linear pipelines. Their objective function for mapping optimization is minimizing the communication cost for forwarding intermediate results on the internal bus. Buffer memory requirements are not considered.

[HSH$^+$09] discuss the problem of mapping streaming applications represented as Kahn Process Networks (graphs of streaming tasks communicating via FIFO queues) to general MPSoC platforms, with the Cell processor as one target. Technically, they use a form of light-weight, stack-less user-level threads on the SPEs, similar to our merger tasks (where our case is somewhat simpler as the code is the same for all mergers except the root), and windowed FIFO buffers for inter-SPE communication. As test application they use a MJPEG decoder. Their framework requires the user to specify the buffer sizes and the mapping of tasks to cores manually in a XML file, whereas these are determined automatically in our case, where the mapping is co-optimized with respect to the memory needs.

In our approach, the mapping problem is simplified by not considering Cell's network topology (four one-directional rings, two per direction) for the mapping optimization. This simplification is motivated by Cell's hardware-level abstraction of the on-chip network as a bus, its high bandwidth and dynamic routing mechanism, and the limited possibilities for the Cell programmer to control the placement of SPE tasks to specific SPEs. Adding details about the target topology leads to a more complex model that might no longer be solved by exact methods like ILP for realistic problem sizes. In the literature, heuristics such as genetic algorithms are often employed for mapping problems with more complex target models where e.g. optimized routing of communication for minimum latency or contention needs to be considered. Moreover, features such as voltage and frequency scaling for parts of the chip (such as individual cores) allow for optimization of power consumption as a second optimization goal in mapping.

For instance, Palesi et al. [PHKC09] consider generic network-on-chip (NoC) platforms, defined by a topology graph, to which they map application task graphs, exploiting application-specific information about communicating pairs of tasks and concurrency or non-concurrency of such communications in order to avoid contention at network links or deadlocks in routing. Ascia et al. [ACP06] propose a genetic algorithm to map task graphs to 2D mesh-based NoC systems, to produce Pareto-optimized mappings in the multi-objective trade-off between performance and power consumption. Nedjah et al. [NCd11, NCd12] consider the hardware-software synthesis problem for applications modeled as a task graph where tasks map to IP blocks, which are to be placed on a generic 2D mesh-based

NoC platform such that area and cost constraints are fulfilled, performance is maximized and power consumption is minimized. A genetic algorithm is used to solve this multi-objective constrained optimization problem.

[GR05] discuss how to execute streaming programs, i.e. programs written in a streaming programming language and targeted for streaming architectures, on a common multicore processor. The tasks either access memory or perform a computational kernel and form a task graph, where the edges represent the data dependencies. The mapping discussed is similar to (static or dynamic) scheduling of task graphs with malleable tasks, where access and computation shall be overlapped as far as possible. The malleable tasks, i.e. the computational kernels, are parallelized by multiple threads with similar load.

6.3 Parallel Sorting Algorithms

We discussed *Cell-BE-specific parallel sorting algorithms*, see [IMKN07, GBY07, RB07, SB09], already in Section 2. Here we briefly review further related work on parallel merging.

Our mergesort algorithm, as described above, uses sequential mergers, which are easy to implement and SIMDize and have relatively low overhead, but make the root of the merger tree the performance bottleneck. Hence, the parallel time for mergesort of N elements is $\Theta(N)$ even with $O(\log N)$ cores. If compared with sequential mergesort, the speedup is bounded by $O(\log N)$. For future generations of processors with many more cores, one may consider parallel mergers in the upper levels of the tree to relax the bottleneck. This means that more than one core must be assigned to such a level in the pipeline tree, and these must execute a parallel merge algorithm. The closer the level is to the root (i.e., the more items are to be merged), the more cores must be assigned to it, in order to balance the load over all cores. Parallel merging is based on parallel binary searches for ranking one sorted subsequence into another one, see e.g. [JáJ92]. However, the resulting fully parallel mergesort algorithm does $O(N \log^2 N)$ operations and is thus not work-optimal. This suggests a superlinear cost-up as the output data rate is to be increased. Anyway, the available memory bandwidth will again be the main limiting factor.

[Col88] proposed a work-optimal pipelined parallel mergesort algorithm. His algorithm uses parallel mergers that use a so-called sample, which is a known subsequence of the merged output sequence, as additional knowledge that allows merging in parallel. The larger the sample is, the more processors can be utilized. The key idea is that the *merge-with-sample* algorithm is applied iteratively on each pipeline node in order to gradually assemble more and more precise samples from coarser ones and propagating partial information (samples) upward the tree as early as possible, in order to keep the nodes of several tree levels active at the same time.

Cole's algorithm is very complicated and has very high overhead, such that it is not useful for realistic problem sizes. [Nat90] found by simulation on a CREW PRAM simulator that Cole's algorithm outperforms Batcher's bitonic sort only for problem sizes larger than approximately $3 \cdot 10^{20}$, and it outperforms even a sequential sort routine running on a single PRAM processor only for $N > 10^5$.

7 Conclusion and Future Work

With an implementation of the global merging phase of parallel mergesort as a case study of a memory-intensive computation, we have demonstrated how to lower memory bandwidth requirements in code for chip-multiprocessors like the Cell BE by optimized on-chip pipelining.

We have shown that the performance of on-chip pipelining strongly depends on finding a suitable mapping of the merger tree tasks to cores, and described it as a multi-objective optimization problem. We proposed and discussed three algorithms for solving it in general: an ILP formulation that can provide Pareto-optimal mappings for trees with up to 7 levels within reasonable time, and two fast approximation algorithms, DC-map and IT-map.

With the best mappings, our implementation for Cell obtained speedups of up to 70% on QS20 and 143% on PS3 over the global merging phase of CellSort, which dominates the sorting time for larger input sizes.

On-chip pipelining uses several architectural features that may not be available in all multicore processors. For instance, the possibility to forward data by DMA between individual on-chip memory units is not available on current GPUs where communication is only to and from off-chip global memory. The possibility to lay out buffers in on-chip memory and move data explicitly is not explicitly available on cache-based multicore architectures. Nevertheless, on-chip pipelining will be applicable in upcoming heterogeneous architectures for the DSP and multimedia domain with a design similar to Cell, such as ePUMA, see [Liu09]. Intels 48-core *single-chip cloud computer* [Cor10] [HDV+11], which we use as an experimental platform in ongoing work on on-chip pipelining [MKA+12], supports on-chip forwarding between tiles of two cores, with 16KB buffer space per tile, to save off-chip memory accesses. Also, Tilera processors (`www.tilera.com`) have a somewhat similar structure. Thus, the techniques proposed are not restricted to the Cell processor.

On-chip pipelining is also applicable to other streaming computations such as general data-parallel computations or FFT. In [KK09] we have described optimal and heuristic methods for optimizing mappings for general pipelined task graphs. This could be combined with a generic on-chip pipelining framework for streaming computations on Cell, such as the recent framework in [HSH+09].

The downside of on-chip pipelining is more complex code that is harder to debug. We are currently working on an approach to *generic on-chip pipelining* where, given an arbitrary acyclic pipeline task graph, an (optimized) on-chip-pipelined implementation will be generated for Cell or similar target architectures. This feature may, for instance, extend our BlockLib skeleton programming library for Cell, see [ÅEK08]. An alternative back-end could be the framework of [HSH$^+$09].

So far, our algorithms do not take into account the communication structure (communication distances, contention on links) between cores. All permutations of the cores are considered equivalent with respect to communication performance. However, several studies for Cell (see [KV10, SNBS09]) indicate that the concrete mapping of threads to cores may have a notable influence on the communication performance. For example on the Cell processor, only non-overlapping communications can be placed on the same EIB ring at the same time. Other multicore processors like Intel's SCC with a mesh network also impose restrictions on communication. Hence, in future work we will try to incorporate this aspect into our mapping algorithms.

Acknowledgements

C. Kessler acknowledges partial funding from EU FP7 (project *PEPPHER*, grant #248481, www.peppher.eu), VR (*Integrated Software Pipelining*), SSF (*ePUMA*), SeRC, and CUGS. We thank Nicolas Melot for helping with some timing detail measurements on PS3. We thank Niklas Dahl and his colleagues from IBM Sweden for giving us access to their QS20 blade server.

References

[ACP06] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. A multi-objective genetic approach to mapping problem on network-on-chip. *Journal of Universal Computer Science*, 12(4):370–394, 2006.

[ÅEK08] Markus Ålind, Mattias Eriksson, and Christoph Kessler. Blocklib: A skeleton library for Cell Broadband Engine. In *Proc. ACM Int. Workshop on Multicore Software Engineering (IWMSE'08) at ICSE-2008, Leipzig, Germany*, pages 7–14, New York, NY, USA, May 2008. ACM.

[BLMR08] Luca Benini, Michele Lombardi, Michela Milano, and Martino Ruggiero. A constraint programming approach for allocation and scheduling on the CELL Broadband Engine. In *Proc. 14th Constraint Programming (CP-2008), Sydney*, pages 21–35. Springer LNCS 5202, September 2008.

[Bok88] Shahid H. Bokhari. Partitioning problems in parallel, pipelined and distributed computing. *IEEE Transactions on Computers*, 37(1), January 1988.

[Col88] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[Cor10] Anne-Marie Corley. Intel lifts the hood on its "single-chip cloud computer". IEEE Spectrum Online (9 feb 2010) report from ISSCC-2010, spectrum.ieee.org/ semiconductors/processors/ intel-lifts-the-hood-on-its-singlechip-cloud-computer, February 2010.

[CRDI07] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation—a performance view. *IBM J. Res. Devel.*, 51(5):559–572, Sept. 2007.

[GBY07] Bugra Gedik, Rajesh Bordawekar, and Philip S. Yu. Cellsort: High performance sorting on the cell processor. In *Proc. 33rd Int.l Conf. on Very Large Data Bases*, pages 1286–1207, 2007.

[GR05] Jayanth Gummaraju and Mendel Rosenblum. Stream Programming on General-Purpose Processors. In *Proc. 38th Int. Symp. on Microarchitecture (MICRO 38)*, Barcelona, Spain, November 2005.

[HDV$^+$11] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart. A 48-Core IA-32 message-passing processor in 45nm CMOS using on-die message passing and DVFS for performance and power scaling. *IEEE J. of Solid-State Circuits*, 46(1):173–183, January 2011.

[HKK10] Rikard Hultén, Christoph W. Kessler, and Jörg Keller. Optimized on-chip pipelined merge sort on the Cell/B.E. In P. D'Ambra, M. Guarracino, and D. Talia, editors, *Proc. Euro-Par conference, Part II, LNCS 6272*, pages 187–198. Springer-Verlag, 2010.

[HR93] Phu D. Hoang and Jan M. Rabaey. Scheduling of DSP programs onto multiprocessors for maximum throughput. *IEEE Trans. on Signal Processing*, 41(6):2225–2235, June 1993.

[HSH$^+$09] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele. Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOs. In *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), Grenoble, France*, pages 35–44, October 2009.

[Hul10] Rikard Hultén. Optimized on-chip software pipelining on the Cell BE processor. Master thesis LIU-IDA/LITH-EX-A-10/015-SE, Dept. of Computer and Information Science, Linköping University, Sweden, 2010.

[ILO07] ILOG Inc. Cplex version 10.2. www.ilog.com, 2007.

[IMKN07] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *Proc. 16th Int.l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 189–198. IEEE Computer Society, 2007.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[KB06] Vida Kianzad and Shuvra S. Bhattacharyya. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Trans. on Par. and Distr. Syst.*, 17(7):667–680, July 2006.

[KK09] Christoph W. Kessler and Jörg Keller. Optimized mapping of pipelined task graphs on the Cell BE. In *Proc. 14th Int. Workshop on Compilers for Parallel Computing (CPC-2009), Zürich, Switzerland*, January 2009.

[Kun82] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15:37–46, January 1982.

[KV10] Jörg Keller and Anna L. Varbanescu. Performance impact of task mapping on the Cell BE multicore processor. In *Proc. Int. Symp. Computer Architecture (ISCA 2010), 1st Workshop on Applications for Multi– and Many-Core Processors*, June 2010.

[Liu09] Dake Liu et al. PUMA parallel computing architecture with unique memory access. www.da.isy.liu.se/research/scratchpad/, 2009.

[LMRT96] Reinhard Lüling, Burkhard Monien, Alexander Reinefeld, and Stefan Tschöke. Mapping tree-structured combinatorial optimization problems onto parallel computers. In *Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques*, pages 115–144, London, UK, 1996. Springer-Verlag.

[MKA$^+$12] Nicolas Melot, Christoph Kessler, Kenan Avdic, Patrick Cichowski, and Jörg Keller. Engineering parallel sorting for the Intel SCC. *Procedia Computer*

Science, 9(0):1890 – 1899, 2012. Proc. Int. Conf. on Computational Science (ICCS 2012), WEPA.

[MLZ99] Martin Middendorf, Welf Löwe, and Wolf Zimmermann. Scheduling inverse trees under the communication model of the LogP-machine. *Theoretical Computer Science*, 215:137–168, 1999.

[Nat90] Lasse Natvig. *Evaluating Parallel Algorithms: Theoretical and Practical Aspects.* PhD thesis, Norwegian Institute of Technology, University of Trondheim, Norway, 1990.

[NCd11] Nadia Nedjah, Marcus Vinicius Carvalho da Silva, and Luiza de Macedo Mourelle. Customized computer-aided application mapping on NoC infrastructure using multiobjective optimization. *Journal of Systems Architecture - Embedded Systems Design*, 57(1):79–94, 2011.

[NCd12] Nadia Nedjah, Marcus Vinicius Carvalho da Silva, and Luiza de Macedo Mourelle. Preference-based multi-objective evolutionary algorithms for power-aware application mapping on NoC platforms. *Expert Syst. Appl.*, 39(3):2771–2782, 2012.

[PHKC09] Maurizio Palesi, Rikard Holsmark, Shashi Kumar, and Vincenzo Catania. Application specific routing algorithms for networks on chip. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):316–330, March 2009.

[RB07] N. Ramprasad and Pallav Kumar Baruah. Radix sort on the Cell broadband engine. In *Int.l Conf. High Perf. Comuting (HiPC) – Posters*, 2007.

[RGB$^+$08] Martino Ruggiero, Alessio Guerri, Davide Bertozi, Michaela Milano, and Luca Benini. A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC platforms with communication awareness. *Int. J. of Parallel Programming*, 36(1), February 2008.

[RJ94a] Sibabrata Ray and Ilong Jiang. Improved algorithms for partitioning tree and linear task graphs on shared memory architecture. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 363–370, June 1994.

[RJ94b] Sibabrata Ray and Ilong Jiang. Sequential and parallel algorithms for partitioning tree task graphs on shared memory architecture. In *Proc. International Conference on Parallel Processing, Volume 3*, pages 266–269, August 1994.

[SB09] Daniele P. Scarpazza and Gordon W. Braudaway. Workload characterization and optimization of high-performance text indexing on the Cell Broadband Engine (Cell/B.E.). In *Proc. IEEE Int. Symp. on Workload Characterization (IISWC '09)*, pages 13–23, October 2009.

[SK01] Radoslaw Szymanek and Krzysztof Kuchcinski. A constructive algorithm for memory-aware task assignment and scheduling. In *CODES '01: Proc. 9th int. symposium on Hardware/software codesign*, pages 147–152, New York, NY, USA, 2001. ACM.

[SNBS09] C.D. Sudheer, T. Nagaraju, P.K. Baruah, and Ashok Srinivasan. Optimizing assignment of threads to SPEs on the Cell BE processor. In *Proc. 10th Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC at IPDPS 2009)*, pages 1–8, May 2009.