Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2012) 1–10

Procedia Computer Science

International Conference on Computational Science, ICCS 2012, WEPA

Engineering parallel sorting for the Intel SCC

Nicolas Melot[a], Christoph Kessler[a], Kenan Avdic[a], Patrick Cichowski[b], Jörg Keller[b,*]

[a] *Dept. Computer and Information Science (IDA), Linköping University, Sweden*
[b] *Fac. Mathematics and Computer Science, FernUniversität in Hagen, Germany*

Abstract

The Single-Chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. It comprises 48 Intel-x86 cores linked by an on-chip high performance mesh network, as well as four DDR3 memory controllers to access an off-chip main memory. We investigate the adaptation of sorting onto SCC as an algorithm engineering problem. We argue that a combination of pipelined mergesort and sample sort will fit best to SCC's architecture. We also provide a mapping based on integer linear programming to address load balancing and latency considerations. We describe a prototype implementation of our proposal together with preliminary runtime measurements, that indicate the usefulness of this approach. As mergesort can be considered as a representative of the class of streaming applications, the techniques developed here should also apply to the other problems in this class, such as many applications for parallel embedded systems, i.e. MPSoC.

Keywords: Parallel sorting, algorithm engineering, on-chip pipelining, streaming applications

1. Introduction

The Single-Chip Cloud Computer (SCC) experimental processor [1] is a 48-core "concept-vehicle" created by Intel Labs as a platform for many-core software research. Its 48 cores communicate and access main memory through a 2D mesh on-chip network attached to four memory controllers (see Figure 1(a)).

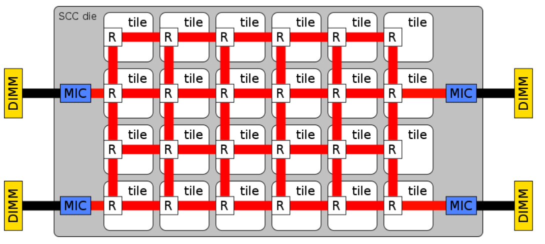
Engineering parallel algorithms for such an architecture involves many considerations, e.g. load balancing, communication patterns, and memory access patterns. Especially for algorithms that mainly transport and modify large amounts of data, so-called streaming applications, the accesses to main memory may represent a performance bottleneck [2], despite the use of caches, because of the limited bandwidth to off-chip main memory. Streaming applications are of interest because they comprise lots of industry applications in embedded systems, e.g. processing of image sequences. Because of throughput requirements, those applications regularly call for parallelization, e.g. with multiprocessor systems-on-chip (MPSoC). The goal for bandwidth optimization, combined with other design targets, leads to approaches such as on-chip pipelining for multicore processors [2].

On-chip pipelining is an algorithmic design pattern for multicore computing that is applicable to pipelineable computations and that allows to trade a higher on-chip communication volume for a reduced amount of off-chip memory accesses, which can improve throughput if the computation is main-memory access bound.

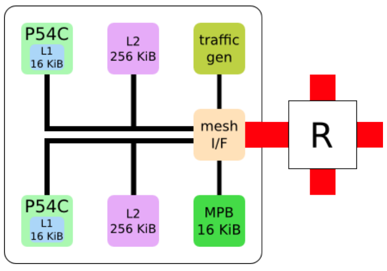
[*] *Email address:* `joerg.keller@fernuni-hagen.de` (Jörg Keller)

We consider sorting of large data sets as a simple and well-researched model for streaming applications. We investigate the implementation of sorting on the SCC as an algorithm engineering problem, and devise a combination of pipelined and non-pipelined mergesort as a good fit for that architecture. Preliminary performance measurements with a prototype implementation indicate the validity of our hypothesis and the applicability of on-chip pipelining.

The remainder of this article is structured as follows. Section 2 introduces the SCC. In Section 3 we present our arguments for the choice of sorting algorithm used, together with an integer linear programming (ILP) approach to optimize the load balancing and latency reduction in memory accesses. In Section 4 we present preliminary performance results of our prototype implementation, and give an outlook to future work in Section 5.



(a) A schematic view of the SCC die. Each box labeled DIMM represents 2 DIMMs.



(b) One of the 24 tiles of the SCC.

Figure 1: A schematic representation of the SCC die and the composition of its tiles.

2. The Single Chip Cloud computer

The SCC provides 48 independent Intel x86 cores, organized in 24 tiles. Figure 1(a) provides a global schematic view of the chip. Tiles are linked together through a 6×4 mesh on-chip network, capable to provide 64GB/s on each of its links. A tile is represented in Figure 1(b). Each tile embeds two cores as well as a common message passing buffer (MPB) of 16KiB (8KiB for each core); the MPB supports direct core-to-core communication.

The cores are IA-32 x86 (P54C) cores which are provided with individual L1 and L2 caches of size 32KiB (16KiB code + 16KiB data) and 256KiB, respectively, but no SIMD instructions. Each link of the mesh network is 16 bytes wide and exhibits a 4 cycles crossing latency, including the routing activity.

The overall system admits a maximum of 64GiB of main memory accessible through four DDR3 memory controllers (MIC). The four memory controllers offer an aggregated peak bandwidth of 21GB/s [3]. Previous work revealed that the bandwidth available to the cores varies widely with the read or write access pattern [4]. Each core is attributed a private domain in this main memory whose size depends on the total memory available (682 MiB in the system used here). Six tiles (12 cores) share one of the four memory controllers to access their private memory. Furthermore, a part of the main memory is shared between all cores; its size can vary up to several hundred megabytes. Note that private memory is cached on cores' L2 cache but caching for shared memory is disabled by default in Intel's framework RCCE. When caching of shared memory is activated, the SCC offers no coherency among cores' caches to the programmer. This coherency must be implemented through software methods, e.g. by flushing caches.

The SCC can be programmed in two ways: a baremetal version for OS development, and using Linux. In the latter setting, each core runs an individual Linux kernel on top of which any Linux program can be loaded. Also, Intel provides the RCCE library which contains MPI-like routines to synchronize cores and allow them to communicate data to each other. RCCE also allows the management of voltage and frequency scaling.

Previous work assessing the SCC indicates a significant performance penalty from the use of shared main memory, compared to using private memory or the on-chip network [5]. That work used non-pipelined parallel mergesort as a case study. The results also indicate that the distance to the memory controller has an influence on round-trip time. Further tests indicate that there is sufficient read bandwidth when accessing memory in a cache-friendly pattern, but that write bandwidth suffers [4]. Also [6] compare programming styles for the Intel SCC and find that using the on-chip message-passing buffers (MPB) provides performance advantages, and that using shared memory has performance problems either due to lack of caching or overhead from software cache coherency.

3. A hybrid parallel mergesort algorithm

In order to engineer a sorting algorithm for a given manycore system, one first has to consider whether the data set to be sorted fits into the on-chip memory (either local memories or caches) or whether the data is stored in the off-chip main memory. We consider the latter situation, especially as the SCC does not possess explicit local memory but only provides small on-chip caches and small buffers for core-to-core communication. While the main memory of SCC in principle can be used as a shared memory, there are two obstacles here. First, the system software only allows a fraction of the main memory to be declared as shared memory, the majority is used as private memory, where each private memory section can only be accessed by one user process. Second, the shared memory is not cached by default. As each core can only have one outstanding memory request, this results in a low bandwidth for shared memory accesses, especially as all cores will access the same memory controller (accesses to the private memory are distributed over the 4 available memory controllers). If caching is enabled for shared memory, there is no hardware coherency i.e. the application would have to take care of that. Therefore, we have concentrated on working with message-passing like algorithms: although each core physically is able to access each part of the external memory, there is no way (at least not on application level) to re-label data in one private memory to belong to the private memory of a different core. Thus a core-to-core communication has to take place to transfer such data.

For sorting algorithms where each core uses its off-chip private memory and communicates with other cores over the on-chip interconnection network, we must consider that the network's bandwidth (64 GByte/s) is much higher than the aggregate bandwidth to the off-chip memory (21 GByte/s). Furthermore, bandwidth to off-chip main memory is strongly dependent on the memory access pattern [4], with access to blocks of consecutive words providing the highest bandwidth. Thus, we arrive at a situation similar to parallel external memory algorithms. This moves our focus to merge sort which is a popular external sorting algorithm. It also has proven more efficient than other algorithms like e.g. bitonic sort used in CellSort on the Cell processor [7]. In order to relieve the external memory, pipelined mergesort has been introduced to forward results from a merge step directly to a another core where the follow-up merge step is to be executed; see e.g. [2] for results on the Cell processor.

Mergesort, based on the divide-and-conquer paradigm, recursively splits an input data into 2 or more chunks until they are small enough to be sorted in a straightforward way. The actual work is done in the combine phase of each divide-and-conquer step where the chunks are then merged together into a larger, sorted sequence. The data flow order among the combine phases implies a *merge tree*, where the nodes represent the merge tasks and edges represent forwarding of sorted subsequences. Hence, the leaves merge the chunks sorted in the base case, each merge task pushes its merged input stream to the parent node in the next higher level, and the root performs the final merge.

On-chip pipelined mergesort organizes this merge tree computation in a pipeline, with all tasks running concurrently on the SCC's 48 cores. Producer and consumer tasks mapped to the same tile communicate through the tile's local memory, those mapped to different tiles through the mesh network. The leaf tasks start the computation by reading data from memory and they forward their intermediate merging results towards their consumer tasks at the next level in the task graph. These tasks can then start the same process, and so on, until all the tasks in the pipeline are active. This schema restricts memory writes to the root node. As the merge operation is done blockwise, follow-up tasks can start as soon as leaf tasks have produced their first block of intermediate results.

As sending and receiving tasks can be mapped to different processors, the sending tasks can restart with new input data while the receiving one can process the data it has just received. Hence, parallelism is achieved through the overlapping execution of successive tasks. As there can be many more tasks than processors, a processor will generally handle several tasks. This introduces an optimization problem for mapping tasks to processors. One difficulty consists of keeping a balanced computational load among the cores, another one in the preference for keeping the distance between sender and receiver small, especially if the memory controller is involved, in order to reduce delays in data transmission. In [2], an ILP-based model for multi-criterion optimized mapping of tasks was presented. In this work, an adapted model is described in Section 3.2.

3.1. Hybrid parallel sorting on the SCC, Overview

Merging complexity (using ordinary sequential merging) is linear in the input size. Hence, each level of the merge tree involves the same computational load. In general, as the root node will allocate a core alone, an m-level mergetree will be mapped to m cores. The high number of 48 cores available on SCC however rules out this direct approach because of the vast size of the merge tree involved. Hence, on-chip pipelined mergesort needs to be adapted by using several smaller mergetrees concurrently. As each tree will allocate the same number of cores, both the number of the trees and the number of cores per tree (which equals the number of levels per tree) should be a divisor of the number of the cores (48). Also, the number of the trees should be a power of two, to allow a balanced final merging phase, and the number of levels in the tree should be as large as possible, to get more work done in this phase, but less than 10, to avoid cores with more than 100 concurrent tasks. As a result, we arranged for 8 trees of 6 levels each. The guidelines above make this arrangement preferable over 4 trees with 12 levels each, or 16 trees with 3 levels each.

Thus, our approach to parallel sorting on SCC as implemented in this work consists of three phases.

- In phase 0, all the leaf tasks of 8 merging trees mapped to 8 disjoint sets of 6 cores, as detailed in Section 3.2, fetch input data from files and locally sort their inputs using qsort.
- Phase 1 runs 8 on-chip-pipelined 6-level merge trees simultaneously. This phase produces 8 sorted subsequences from 8 root nodes. Note that the mapping of the tree leaves for phase 1 influences the runtime of phase 0 as well via the number of cores that carry leaves, so this has to be considered when mapping the tasks to cores.
- Phase 2 merges these 8 subsequences together, using a non-pipelined mergesort where merge nodes are parallelized. We preferred this over using a parallel sample sort with all 48 processors, because of the coordination overhead involved in implementing the latter solution.

3.2. *Optimizing the task mapping by Integer Linear Programming*

For mapping the overall pipelined task graph to the SCC resources, we have developed multiple variants of an integer linear programming (ILP) based method that either optimize (1) for the aggregated overall distance (in hops) between communicating tasks, weighted by their inter-task communication volumes, or (2) for the aggregated overall distance of tasks to their memory controller weighted by the tasks' memory access volumes (and possibly by the direction of the access, to prioritize write access). In addition, the model also balances the computational load per core, and tries to reduce the number of leaf tasks per core, to improve the runtime of phase 0. To consider all goals, we use a linear combination of the respective target functions, which is controlled by weight parameters.

The architecture model part of our ILP model can be configured by the number of tiles per row and column and by the number and distribution of memory controllers. Thus, besides the usage to provide optimal mappings for a concrete application such as mergesort, the model can also be used for computer architecture research. For example, one can compute optimal positions for different numbers of memory controllers, given an application and the number and arrangement of cores for a hypothetic manycore chip.

In order to reduce model complexity by exploiting symmetries in the SCC architecture, we consider the problem of mapping a pipelined merge task *forest* (here, eight parallel 64-to-1 pipelined merge trees) to SCC and split it into four symmetric subproblem instances. Each subproblem is mapped to an SCC quadrant with its memory interface controller. Hence, only one instance, i.e. mapping two 64-to-1 merge trees (thus 126 tasks) to one quadrant consisting of 3×2 tiles of 2 cores each, actually needs to be solved. Two 6-level binary trees are easily modeled by a single 7-level tree where the now artificial root node 1 is assigned a zero workload. We further simplify the problem by mapping tasks to tiles only, not to individual cores. The details of our ILP-based approach are given in an appendix.

As an example, Fig. 2 depicts two mappings of two 6-level binary trees with roots 2 and 3 onto one quadrant of the SCC. Both mappings result from different settings of the weight parameters and are optimal with respect to load balancing. The left mapping is a level-wise mapping, which incurs a small distance to memory for leaf and root tasks but increased core-to-core communication. The right mapping puts each 4-level subtree containing leaves into one core. This minimizes the distances for core-to-core communication but increases the distance to memory for leaf and root tasks. Thus, if goal (2) has a stronger influence in the optimization, the level-wise mapping is to be preferred over the block-wise mapping. If writes to memory would get a higher weight than reads, because the write bandwidth to memory is smaller, the advantage would be even more on the side of the level-wise mapping. In the block-wise mapping however, the leaves are distributed over 8 cores, in contrast to only 2 cores for the level-wise mapping, so that the runtime for phase 0 is improved. Thus, depending on the detailed user preferences, different mappings can be chosen out of a number of pareto-optimal mappings.

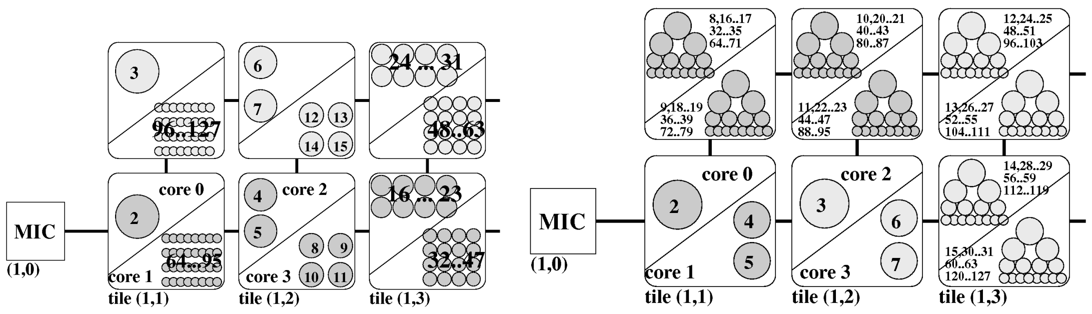


Figure 2: Mappings of two 6-level binary trees onto one SCC quadrant. The co-tasks are not depicted, they are all mapped to the MIC. Left: a level-wise mapping. Right: a block-wise mapping.

Table 1: ILP solver times in seconds for mapping 4, 5 and 6 level binary merge trees to different SCC mesh sizes, using 1 MIC and accounting for maximum communication loads across any mesh edge, in seconds (CPLEX 10.2). A – means that no optimal result was obtained after 10 minutes, but an approximation solution is reported.

3x2 Quadrant, 1 MIC				
k	#var	$\epsilon = 0.5$	$\epsilon = 0.1$	$\epsilon = 0.9$
4	374	0.5	0.1	1.0
5	774	1.8	5.5	0.2
6	1574	53.9	–	111.9

6x4 Quadrant, 1 MIC		
k	#var	$\epsilon = 0.5$
4	1818	–
5	3786	93.6
6	7722	–

ILP solver results. Table 1 shows the runtime requirements of our ILP model for cases with a single MIC, using CPLEX 10.2 as ILP solver. It turns out that when considering one quadrant the problem can be solved to optimality within at most 2 minutes in almost all cases, while considering the whole SCC with 24 tiles leads to considerably higher complexity. However, using a newer ILP solver such as Gurobi might make even such configurations feasible. First tests with Gurobi (also running on a faster machine with more memory) show that, for mapping sufficiently large trees, significant improvements in optimization time are possible, e.g. a 250x speed-up for mapping a 5-level binary tree to a 4x6 mesh, and a 6-level tree mapping which was not feasible with CPLEX could be solved to optimality in 58 seconds with Gurobi.

3.3. Phase 1: on-chip pipelined merge

The first phase of the on-chip pipelined mergesort implemented on SCC is quite similar to the implementation on Cell [2]. The focus here is given to 6 cores of one of the SCC's 4 quadrants, i.e., one octant. All 7 other octants of 6 cores execute in parallel the same actions.

All merging tasks of a 6-level binary merge tree are allocated to 6 cores, using an optimal task mapping computed by the ILP solver (see Section 3.2). The cores that are assigned some of the merge tree leaves locally sort each input buffer stored in their private main memory (phase 0). Once all leaves' buffers are sorted, they are merged together as part of the on-chip pipelined merging procedure. Each merge task is assigned a buffer in the MPB to read data from, and some space in L2 cache to store the merge output before pushing it toward the parent task, with the exception of leaf tasks which take their input directly from the main memory, and of the root task that writes its output to main memory through the L2 cache. Every task in the merging tree executes the following sequence:

1. *Check and merge*
   Checks both input buffers of the task from the core's MPB: if both contain data and if the output buffer in L2 cache has some free space, then the task merges as many elements as possible, i.e. the minimum of the smallest amount of elements available in both input buffers and the amount of elements that the output buffer can take up at the moment.
2. *Check and push*
   Checks the input buffer of this task's parent task; if it can be filled with the sorted data available, then the

parent's input buffer is filled and what is written to the parent is removed from the task memory. The root task writes directly to main memory without any prior verification.

The checks are performed thanks to a *header* preceding every buffer in the MPB. It allows tasks to keep track of the amount of data already sent and received, the amount of data available in this buffer or the amount of free space, as well as where to read input or write output. From this information and the size of the buffer, a task can calculate how much space is available to write, or how much data is available in this buffer. All tasks mapped to the same core time-share it in a Round-Robin fashion, where the time quantum for each task is the time necessary to perform either of the two steps enumerated above.

3.4. Phase 2: Parallel non-pipelined merge

Phase 2 consists of merging the 8 subsequences into one final sorted sequence. For this, we use a parallel mergesort, where each merge node is implemented by several cores to increase performance. For example, in a first round, four merger tasks run concurrently, where each task is implemented by 4 cores. Each core splits one of the two corresponding input blocks into 4 chunks of equal size and separated by 3 pivots. Then the cores look up the positions of those 3 pivots in the other block. Thus there is no need to transfer the pivots between the cores, avoiding overhead from communication and synchronization. Now each core i knows the positions of all pivots in both blocks; thus it can merge the respective ith chunks. From the positions of pivot i in both blocks, it can also compute the position of the merged block in the sorted sequence. While this approach to parallelization does not guarantee balanced work between the cores, the overhead is low which leaves room for imbalance. In the second round, 2 merger tasks run concurrently, each implemented with 8 cores, and in the last round the root merger task is implemented by 16 cores. A higher parallelization lead to longer runtimes because the work imbalance gets too large.

Initially, we also considered a parallel sample sort in shared memory, where each of the 8 subsequences is split into 48 chunks according to 47 pivots, where the pivots are generated as averages of 47 pivots for each subsequence. Then each of the 48 cores could sequentially merge 8 corresponding chunks, one from each of the 8 subsequences. The concatenation of the 48 resulting sequences then forms the final sorted sequence. Details of the parallel sample sort can be found in [8]. While the degree of parallelism is higher with sample sort (48 cores instead of 16), its runtime still was longer than for the non-pipelined mergesort because of the coordination overhead involved that necessitated several synchronizations and scatter/gather communications. Therefore, we did not investigate that variant further.

4. Experimental evaluation

We implemented a prototype of the hybrid sort algorithm for SCC described in Section 3. We ran it on the 48 cores, varying the total input size N from 2^{20} to 2^{25} integers (32 bits each). As our previous work on on-chip pipelined mergesort on Cell [2] exhibits different performance with different available buffer size per core, the sorting algorithm for SCC is run with allocated space in the MPB of 4096, 6144 and 8128 bytes. Each combination of input size and buffer size is run a hundred times to reduce interferences from the underlying operating system, using random data (uniformly distributed unsigned ints) as input. We use the level-wise mapping from the left part of Fig. 2 and the block-wise mapping from the right part of Fig. 2 for the first measurements. Yet, we only include the runtime differences in phase 0 for the latter. The algorithm is monitored so that also sub-timings for the initial sequential sorting phase (phase 0) and the parallel sorting (phases 1 and 2) are available. The timing results are depicted in Table 2. We see that the runtime is growing linearly with the input data size, with a lower pace between 2^{22} and 2^{24} integers. We are still investigating the reasons for this. Also, the difference in buffer size only has an influence for small data size, but is negligible for large data sizes.

The runtime distribution between the phases shows a tendency of phase 0 to grow faster than phase 1, as the runtime in phase 0 grows with $O(N \cdot \log N)$ because of the sorting, while the runtime in phase 1 grows with $O(N)$. In contrast, phase 2 tends to decrease its share on the total runtime from 81% for 2^{20} to 61% for 2^{25} integers (block-wise mapping). This indicates that for larger data sizes, the distribution of work in the parallelized mergers is more evenly.

We compare the performance of phase 1 to the best variants of parallel merge shown in [5], that uses 4 cores to run a 4-level tree and uses 8 cores to run an 8-level tree. As we run our on-chip pipelined merge phase (phase 1) with 8 merging trees, we focus on the runtime of one of them merging an input block 8 times smaller than the global input. Thus we compare phase 1 runtime merging a total of 2^{25} integers (2^{22} per tree) to variants shown in [5] merging 2^{22}

Table 2: Runtime results for different data sizes with maximum buffer size, using both mapping shown in Fig.2 (values expressed in ms).

Data size	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
Total (layer)	481.4	960.8	1842.6	2938.2	5090.4	10308.6
Total (block)	415.6	817.8	1533.0	2279.8	3689.2	7321.0
Phase 0 (layer)	88.0	191.4	413.2	880.8	1875.0	3991.0
Phase 0 (block)	22.2	48.4	103.6	222.4	473.8	1003.4
Phase 1	57.0	114.0	226.0	451.0	902.0	1805.0
Phase 2	336.4	655.4	1203.4	1606.4	2313.4	4512.6

integers. The classic merge implementation in [5] takes 3000ms and 4 cores to run a 4-level tree and 3200ms and 8 cores to run an 8-level tree. Our on-chip pipelined merge takes 1805ms with 6 cores to run a 6 levels tree, yielding a speedup of 1.6 and 1.7, respectively. Thus, our on-chip pipelined merge is more efficient than a classic parallel merge thanks to main memory access optimization despite the unfavorable case. Previous work [4] indicates that the SCC's cores struggle to saturate the main memory, due to old and slow cores coupled to modern, fast memory controllers. In other architectures, on-chip pipelining may yield even better speedup.

Table 3: Scaling the algorithm using one to four quadrants (values expressed in ms).

	1 quadrant	2 quadrants	4 quadrants
Phase 0 (block)	1003	473	222
Phase 1	1805	902	451
Phase 2	956	1221	1606
Total [ms]	3764	2596	2279
Speedup	1	1.5	1.7
Efficiency	1	0.7	0.4

Table 3 details the behavior of our algorithm when scaling it from 1 quadrant (12 cores) to 4 quadrants (48 cores). All variants sort the same input size (2^{23} integers), that is equally distributed among all cores used. Since using 2 or 4 quadrants results in smaller chunks distributed to merging trees, and phases 0 and 1 involve unsynchronized quadrants, it roughly divides the runtime by 2 or 4, respectively, compared to using 1 quadrant. As shown in Table 3, phase 2 does not scale perfectly. This can be explained by the number of input blocks increasing with the number of quadrants, 2 to 8, which leads to more merge steps, and by the fact that our current phase 2 code uses at most 16 cores, even if 48 cores are available. Using more quadrants still results in higher speedup, yet this increasing speedup does not maintain the efficiency at a constant level.

We are not aware of performance results for other sorting algorithms on the SCC, besides our own implementation of non-pipelined mergesort [5]. Thus we compare with sorting algorithms implemented on the Cell processor [9]. The Cell processor, usually employed in pairs, provides 8 processors called SPEs running at 3.2 GHz, where the processors can issue up to 2 instructions per cycle, and the instructions work on 128-bit data (or four 32-bit ints). Thus, considering operations on 32-bit integers as a basic unit, an SPE performs 8 times as many operations per cycle compared to an SCC core. As the SPE operating frequency is 6 times higher than for the SCC cores running at 533 MHz, an SPE is about 48 times more powerful than an SCC core. As a Cell pair comprises 16 SPEs compared to 48 cores in an SCC, a Cell pair is roughly 16 times more powerful than an SCC. CellSort [7], which uses a local sort (corresponds to our phase 0) followed by a bitonic sort (corresponds to our phases 1 and 2), needs 565 ms to sort 32M integers with 16 SPEs, where the time for phase 0 is omitted. As we need about 6317.6 ms for the same data size, we are by a factor of about 1.43 more efficient, considering the factor 16 computed above, although the Cell BE may feature other advantages over the SCC's P54C that are hard to quantify, such as a newer and more efficient architecture.

5. Conclusion and future work

We presented a hybrid sorting algorithm for the SCC processor as an algorithm engineering case study, where we had to combine different algorithmic techniques, knowledge about the processor's memory bandwidth, and an advanced ILP-based multi-objective mapping optimization method to derive an efficient implementation. Preliminary experimental results indicate that the resulting code yields good scaling properties and is competitive with simple off-chip parallel mergesort as well as other high performance parallel sorting algorithms. The algorithm should be portable to other upcoming mesh-based manycore architectures such as e.g. Tilera processors [10] and also to NUMA-based systems such as the AMD Interlagos manycore processor, where the coherent on-chip caches could take over the role of the message-passing buffers. The techniques presented to derive an efficient implementation should be adaptable to other streaming algorithms.

In future work, we plan to extend this study to larger data sets, further algorithms, and to include energy-efficiency aspects into the mapping algorithm. We also plan to use the ILP model for computer architecture research such as optimal placement of memory controllers in on-chip networks.

Acknowledgments

The authors are thankful to Intel for providing the opportunity to experiment with the "concept-vehicle" many-core processor "Single-Chip Cloud Computer".

This research is partly funded by the Swedish Research Council (Vetenskapsrådet), project *Integrated Software Pipelining*, and the CUGS graduate school at Linköping University.

References

[1] 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, 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) (2011) 173–183.

[2] R. Hultén, J. Keller, C. Kessler, Optimized on-chip-pipelined mergesort on the Cell/B.E., in: Proceedings of Euro-Par 2010, Vol. 6272, 2010, pp. 187–198.

[3] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, N. Wilson and, H. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, T. Mattson, A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS, in: Proceedings of IEEE Int. Solid State Circuits Conference (ISSCC 2010), Vol. 978-1-4244-8300-6, 2010, pp. 108–110.

[4] N. Melot, K. Avdic, C. Kessler, J. Keller, Investigation of main memory bandwidth on Intel Single-Chip Cloud Computer, Intel MARC3 Symposium 2011, Ettlingen (2011).

[5] K. Avdic, N. Melot, J. Keller, C. Kessler, Parallel sorting on Intel Single-Chip Cloud Computer, in: Proc. A4MMC workshop on applications for multi- and many-core processors at ISCA-2011, 2011.

[6] C. Clauss, S. Lankes, P. Reble, T. Bemmerl, Evaluation and improvements of programming models for the Intel SCC many-core processor, in: Proc. Int.l Conf. on High Performance Computing and Simulation (HPCS), 2011, pp. 525–532.

[7] B. Gedik, R. Bordawekar, P. S. Yu, Cellsort: High performance sorting on the cell processor, in: Proc. 33rd Int.l Conf. on Very Large Data Bases, 2007, pp. 1286–1207.

[8] K. Avdic, N. Melot, C. Kessler, J. Keller, Pipelined parallel sorting on the Intel SCC, in: Proc. 4th Swedish Workshop on Multi-Core Computing (MCC-2011), 2011.

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

[10] Tilera, Company homepage, www.tilera.com (accessed March 1, 2012).

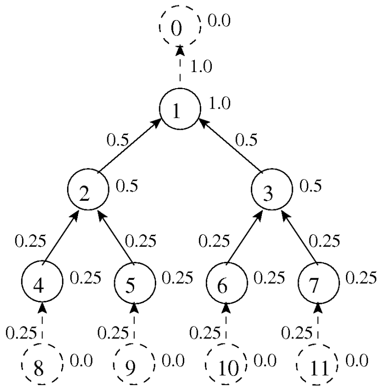
Appendix: ILP Model Details

We model the nodes of a pipelined b-ary merge tree (usually, $b = 2$) with k levels, which has thus $(b^k - 1)/(b - 1)$ merge tasks, as an index set $V = \{1, ..., (b^k - 1)/(b - 1)\}$, where 1 denotes the root merger and where for $v \in V \setminus \{1\}$, parent(v) denotes the merger task to which the task v forwards its output. Hence, the inner nodes (tasks) are $V_{inner} = \{1, ..., (b^{k-1} - 1)/(b - 1)\}$ and $V_{leaves} = V - V_{inner}$ denotes the b^{k-1} leaves. For each leaf task v, we add an artificial *co-task* $v + b^{k-1}$ that models read memory accesses of v. For the root 1, we add a co-task 0 that models its write memory accesses. The overall set of nodes comprising tasks and co-tasks is thus $V_{ext} = \{0, ..., (2 \cdot b^k - b^{k-1} - 1)/(b - 1)\}$, the co-task set is $V_{ext} - V$. See Figure 3(a) for an example with $k = 3$ and $b = 2$.

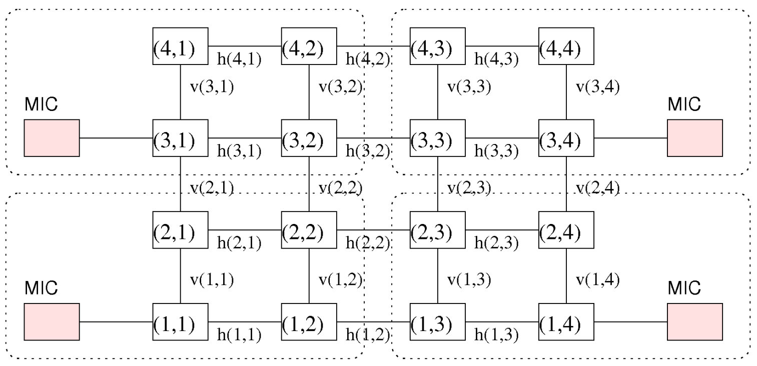
A co-task is to be mapped on the memory controller closest to its corresponding leaf or root task, i.e., on the memory controller of the SCC quadrant where the corresponding task is mapped, while the tasks in V are to be mapped to SCC tiles. The (normalized) computational workload w_v of a merge task $v \in V$ at depth d is $1/b^d$ with $d \in [0; k-1]$. The memory access volume of a co-task $v' \in V_{co}$ is defined by the computational workload of its corresponding task; a co-task has no computational workload.

We model a generic SCC-like architecture as a 2D mesh of tiles with arbitrary extent and arbitrary numbers and positions of memory controllers around the mesh. The SCC mesh is defined by extents $NRows$ and $NCols$, such that $Rows = \{1, ..., NRows\}$ denotes the tiles' row indices and $Cols = \{1, ..., NCols\}$ denotes the tiles' column indices.

For simplicity, the MICs are not modeled by extra tiles adjacent to specific boundary tiles but as part of those boundary tiles, as specified by the (generic) SCC architecture. The additional link to an extra tile would not influence the optimization. The binary parameter *secondMIC* configures if there are one or two MICs per (double) row, i.e., MICs on one or both sides of the SCC. See Fig. 3(b) for an example.



(a) A binary merge-tree pipelined task graph with $k = 3$ levels. The memory access co-tasks for the root merger (write result stream) and for the leaf mergers (read input streams) are shown by dashed circles. The (normalized) computational load is shown next to each task circle, and the communication load next to each edge.



(b) Example of a generic SCC-like architecture with a 4×4 mesh of tiles and with two memory interface controllers (i.e., *secondMIC*= 1) in each second row, defining four quadrants (dotted areas).

Figure 3: Examples of a merge tree and SCC-like architecture.

The mapping to be computed for all tasks and co-tasks will be given by the binary solution variables x where $x_{v,r,c} = 1$ iff node v is placed on tile (r, c), for $v \in V_{ext}$, $r \in Rows$ and $c \in Cols$.

In the mesh, we name each horizontal edge by its left endpoint $(r, c) \to (r, c+1)$ and each vertical edge by its lower endpoint $(r, c) \to (r+1, c)$, see Figure 3(b) for an illustration.

In order to compute communication distances for tree edges, we use auxiliary binary variables $yh_{v,r,r'}$ and $yv_{v,c,c'}$ respectively, for $v \in V_{ext}$, $r, r' \in Rows$ and $c, c' \in Cols$. Here, $yh_{v,r,r'} = 1$ iff task v is placed in row r and task parent(v) is placed in row r'.

We also need to model to which quadrant (memory controller) a tile belongs, which is captured by the auxiliary binary variables $quad_{v,q}$ with $v \in V$ and $q \in Rows$ and a binary variable denoting whether MICs are attached on one or on both outer columns; the standard setting of SCC is one quadrant per half doublerow.

Floating point solution variables *sumDistComm*, *sumDistMem*, *maxCompLoad* and *maxLeaves* are used to denote various optimization goals, namely the weighted sum of communication distances, the weighted sum of memory access distances, the maximum computational load on any tile and the maximum number of leaves per tile, respectively. Then, the objective function of our ILP instance becomes

$$\text{minimize} \qquad \epsilon_1 \cdot maxCompLoad + \epsilon_2 \cdot sumDistMem + \epsilon_3 \cdot sumDistComm + \epsilon_4 \cdot maxLeaves \ ,$$

where $0 \leq \epsilon_i \leq 1$ and $\sum_i \epsilon_i = 1$. By choosing the tuning parameters appropriately between 0 and 1, priority can be given to the various optimization subgoals.

We have the following constraints. *maxCompLoad* is defined by the sum of computational work of all tasks mapped to any tile, and *maxLeaves* is defined by the number of leaves mapped to any tile:

$$\forall r \in Rows, c \in Cols: \ maxCompLoad \geq \sum_{v \in V} w_v \cdot x_{v,r,c}\ , \qquad maxLeaves \geq \sum_{v \in V_{leaves}} x_{v,r,c}$$

Each task node must be mapped to exactly one processor:

$$\forall v \in V_{ext}: \sum_{r \in Rows,\ c \in Cols} x_{v,r,c} = 1$$

where the placement of the co-tasks is additionally constrained: The root's co-task 0 must be fixed to a memory controller in the root's quadrant, where we assume that memory controllers are placed in rows with odd index:

$$\forall r \in \{1,...,NRows/2\}: \sum_{c \in Cols} x_{0,2r,c} \leq 0\ , \qquad \sum_{c \in \{2,...,NCols-secondMIC\}} x_{0,2r-1,c} \leq 0$$

Likewise, a leaf's co-task must be fixed to the memory controller of the quadrant where the leaf is mapped:

$$\forall v \in V_{co},\ r \in \{1,...,NRows/2\}: \sum_{c \in Cols} x_{v,2r,c} \leq 0\ , \qquad \sum_{c \in \{2,...,NCols-secondMIC\}} x_{v,2r-1,c} \leq 0$$

The auxiliary quadrant variables are defined as follows:

$$\forall v \in V,\ mcr \in \{1,...,NRows/2\},\ mcc \in \{0,1\}:$$

$$quad[v, 2mcr-1+mcc] \leq \sum_{\substack{r \in \{2mcr-1,...,2mcr\},\\ c \in \{(mcc\lfloor NCols/2\rfloor+1,...,mcc\lfloor NCols/2\rfloor+NCols/2\}}} x_{v,r,c}$$

$$\forall v \in V: \sum_{mcr \in \{1,...,NRows/2\},\ mcc \in \{0,1\}} quad[v, 2mcr-1+mcc] \geq 1$$

Then, the following constraints force the co-tasks of leaves and the root on the same quadrant as their corresponding tasks:

$$\forall mcr \in \{1,...,NRows/2\},\ mcc \in \{0,1\},\ v \in V_{leaves}, i \in \{1,...,b\}: \ x_{v+b^{k-1},2mcr-1,mcc\cdot(NCols-1)+1} \leq quad_{v,2mcr-1+mcc}$$

$$\forall mcr \in \{1,...,NRows/2\},\ mcc \in \{0,1\}: \ x_{0,2mcr-1,mcc\cdot(NCols-1)+1} \leq quad_{1,2mcr-1+mcc}$$

The communication load on each mesh edge is defined by constraints such as

$$\forall u \in V_{inner},\ i \in \{1,...,b\},\ r1 \in Rows,\ r2 \in Rows: \ yh_{u+b(u-1)+i+1,r1,r2} \geq \sum_{c \in Cols} x_{u+b(u-1)+i+1,r1,c} + \sum_{c \in Cols} x_{u,r2,c} - 1$$

$$\forall u \in V_{inner},\ i \in \{1,...,b\},\ r1 \in Rows,\ r2 \in Rows: \ yh_{u+b(u-1)+i+1,r1,r2} \geq 0$$

for the row segments, and similarly for the column segments.

The weighted sum of communication distances is defined as follows:

$$sumDistComm = \sum_{\substack{u \in V_{inner},\ i \in \{1,...,b\},\\ r1 \in Rows,\ r2 \in Rows}} yh_{u+b(u-1)+i+1,r1,r2} \cdot 0.5|r2 - r1| \cdot w_u + \sum_{\substack{u \in V_{inner},\ i \in B,\\ c1 \in Cols,\ c2 \in Cols}} yv_{u+b(u-1)+i+1,c1,c2} \cdot 0.5|c2 - c1| w_u$$

The weighted sum of memory access distances is defined by

$$sumDistMem = \sum_{\substack{v \in V_{leaves},\\ r \in Rows,\ c \in Cols}} x_{v,r,c}(r + c - 2)w_v + \sum_{\substack{r \in Rows,\\ c \in Cols}} x_{1,r,c}(r + c - 2)$$

We also developed an alternative ILP model that keeps track of the maximum communication load over any edge in the SCC mesh (and also the overall traffic volume on the mesh) rather than latency; unfortunately it is very lengthy and must be omitted due to space limitations. Also, as we found that network link and memory access bandwidth is less critical than latency on SCC [5], we decided to focus on latency optimization in the first hand.