Pipelined Parallel Sorting on the Intel SCC

Kenan Avdic
Linköping Universitet
Dept. of computer and Inf. science
58183 Linköping, Sweden
kenav350@ida.liu.se

Nicolas Melot,
Christoph Kessler
Linköping Universitet
Dept. of computer and Inf. science
58183 Linköping, Sweden
{melot.nicolas, christoph.kessler}@liu.se

Jörg Keller
FernUniversität in Hagen
Fac. of Math. and Computer Science
58084 Hagen, Germany
joerg.keller@FernUni-Hagen.de

ABSTRACT

The Single-Chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. It comprises 48 Intel-IA32 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.

1. INTRODUCTION

The Single-Chip Cloud Computer (SCC) experimental processor [4] 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)). Implementing parallel algorithms on 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, accesses to main memory may represent a bottleneck [5], despite the use of caches, because of the limited bandwidth to 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 [5]. We consider sorting of large data sets as a simple and well-researched model for streaming applications. We investigate implementation of sorting on the SCC as an algorithm engineering problem, and devise a combination of pipelined mergesort and sample sort as a good fit for that architecture. Preliminary performance measurements with a prototype implementation indicates the validity of our hypotheses.

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 load balancing and latency reduction in memory accesses. In Section 4 we present preliminary performance results of a prototype implementation, and give an outlook to future work in Section 5.

2. THE SINGLE CHIP CLOUD COMPUTER

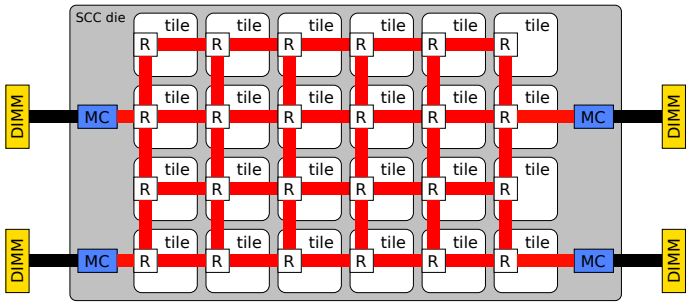
The SCC provides 48 independent Intel-x86 cores, organized in 24 tiles. Figure 1(a) provides a global schematic view of the chip. Tiles are linked together through a 6×4 mesh on-chip network. A tile is represented in Fig.1(b). Each tile embeds two cores as well as a common message passing buffer (MPB) 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-cycle crossing latency, including the routing activity.

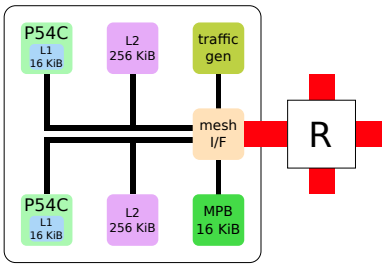
The overall system admits a maximum of 64GiB of main memory accessible through 4 DDR3 memory (MICs) controllers evenly distributed around the mesh. Each core is attributed a private domain in this main memory whose size depends on the total memory available (682 MiB in the system used here). Six tiles (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 is activated, the SCC offers no coherency among cores' caches to the programmer. This coherency must be implemented through software methods, by flushing caches for instance.

The SCC can be programmed in two ways: a baremetal version for OS development, and using Linux. In the latter setting, the cores run 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

MCC 2011, Linköping, Sweden



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



(b) One of the 48 tiles of the SCC.

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

them to communicate data to each other. RCCE also allows the management of voltage and frequency scaling.

Some previous work assessing the SCC reveals a significant performance penalty from the use of shared main memory, compared to using private memory or the on-chip network [1]. It also indicates 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 [6].

3. A HYBRID PARALLEL MERGESORT ALGORITHM

To choose the type of sorting algorithm to be implemented, we took the architectural features of SCC into account. Techniques like parallel quicksort rely on shared memory and thus have a penalty. The restriction to accessing neighboring addresses in private favors merge sort, while the bandwidth restriction on writes favors the implementation of an on-chip pipelined algorithm such as the pipelined mergesort on the Cell [5].

On-chip pipelined mergesort consists in sorting a large sequence of ordered elements thanks to the mergesort algorithm. Mergesort is illustrated in Fig.2; it recursively split an input data into 2 or more chunks, until they are enough small to be trivially sorted. All chunks are then merged together into a larger, sorted sequence. This phase involves a merge tree, where each the node represents a merge task. The leaves take the chunks trivially sorted and push them to the nodes in the lower level, and the root performs the final merge. On-chip pipelined mergesort organizes this process in a pipeline, across SCC's 48 cores through its mesh network. The leaf tasks start the computation and forward their intermediate result toward further tasks in the task graph. These tasks can then start the same process, and so until all the tasks in the pipeline are active. This 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 the first block of intermediate results.

As sending and receiving tasks can be mapped to different processors, the sending task can restart with new input data while the receiving one can process the data it has just received. Thus parallelism is achieved through the overlapping executions of successive tasks. As there can be many more tasks than processors, a processor can handle several tasks. One difficulty of this technique consists in keeping a balanced computational load among the cores, and the necessity to keep the distance between sender and receiver small, especially if the memory controller is involved, to restrict delays in data transmission. In [5], an ILP-based model for multi-criterion optimized mapping of tasks was presented. In this work, an adapted model is described in subSection 3.2.

3.1 Parallel sort on the SCC

In its simplest form, each level of the merge tree would be merged to one core, to achieve a balanced computational load. Thus, in [5], an 8-level merge tree was mapped onto 8 SPEs. Yet, the high number of cores available on SCC (48) excludes this approach because of the vast size of the tree involved. A 48-level tree contains $2^{48}-1$ merging tasks to distribute among all 48 cores, or more than 2^{43} tasks (more than 8 thousand billions) per core on average. Thus, parallel sort as implemented in this work on the SCC consists of three phases. In phase 0, all the leaf tasks of 8 merging trees mapped to 8 disjoint sets of 6 cores, as described in subSection 3.2, fetch input data from files and locally sort their input buffer using qsort. Phase 1 runs 8 on-chip pipelined merge trees simultaneously using the same 8 trees and mapping as Phase 0. This phase produces 8 sorted subsequences from 8 root nodes. Phase 2 merges these 8 subsequences together, using a parallel sample sort with all 48 processors and working through shared main memory.

3.2 Task mapping ILP model

For mapping the overall pipelined task graph to the SCC resources, we have developed multiple integer linear programming (ILP) based solutions that either optimize for the aggregated overall distance (in hops) between communicating tasks, weighted by their inter-task communication volumes, or for the aggregated overall distance of tasks to their memory controller weighted by the tasks' memory access volumes, or a weighted linear combination of both goals. In addition, the model balances the computational load per core.

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.

We split the mapping problem of a symmetric pipelined task graph (here, eight parallel 64-to-1 pipelined merge trees) for SCC into four symmetric subproblem instances. Each subproblem is mapped to an SCC quadrant with its memory interface controller. Hence only one instance, mapping two 64-to-1 merge trees (thus 128 tasks) to one quadrant of 6 tiles, actually needs to be solved. This considerably reduces the problem complexity. We further simplify the problem by mapping tasks to tiles only, not to individual cores.

3.2.1 ILP Model Details

We model a pipelined b-ary merge tree (usually, $b = 2$) with k levels, which has thus $(b^k - 1)/(b-1)$ merge tasks, with $V = \{1, ..., (b^k-1)/(b-1)\}$, where 1 denotes the root merger. 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 $V_{ext} = \{0..(2 * (b^k) - (b^{k-1}) - 1)/(b-1)\}$, the co-task set is $V_{ext} - V$. 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. The communication volume from a task v to its successor $\lfloor v/b \rfloor$ is given by the computational load of v.

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. The MICs are modeled by artificial extra tiles that are adjacent to specific boundary tiles, as specified by the (generic) SCC architecture. 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.

The mapping to be computed for all tasks and co-tasks will be given by the binary solution variables $x_{v,r,c} = 1$ 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 identify each horizontal edge with its left endpoint $(r,c) \to (r,c+1)$ and each vertical edge with its lower endpoint $(r,c) \to (r+1,c)$.

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 double-row.

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

$$\begin{aligned}&\text{minimize}\\ &\varepsilon \cdot maxCompLoad\\ &+ (1-\varepsilon)(1-\zeta) \cdot sumDistComm\\ &+ (1-\varepsilon)\zeta \cdot sumDistMem;\end{aligned}$$

By choosing the tuning parameters ε and ζ appropriately between 0 and 1, priority can be given to the various optimization sub-goals.

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

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

Each task node must be mapped to exactly one processor:

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

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:

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

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

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

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

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

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 on the same quadrant as their corresponding tasks; for the leaves,

$$\forall mcr \in \{1, ..., NRows/2\},\; mcc \in \{0,1\},\; v \in V_{leaves}, i \in \{1, ..., b\}:$$

$$x_{v+b^{k-1}, 2mcr-1, mcc \cdot (NCols-1)+1} \leq quad_{v, 2mcr-1+mcc}$$

and for the root

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

$$x_{0,2mcr-1,mcc\cdot(NCols-1)+1} \leq quad_{1,2mcr-1+mcc}$$

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:

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

$$+sum_{\substack{u \in V_{inner},\ i \in B,\\ c1 \in Cols,\ c2 \in Cols}} yv_{u+b(u-1)+i+1,c1,c2} \cdot 0.5 \cdot |c2-c1| \cdot w_u$$

Finally, 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 have 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 and memory access bandwidth is less critical than latency on SCC, we decided to focus on latency optimization in the first hand.

Two parallel k-level trees are then easily modeled by a single $k+1$ level tree where the now artificial root node 1 is assigned a zero workload and the write co-tasks are connected to the root children (2 etc.) instead.

3.2.2 Experimental 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 considering one quadrant 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.

3x2 Quadrant, 1 MIC				
k	#var	$\varepsilon = 0.5$	$\varepsilon = 0.1$	$\varepsilon = 0.9$
4	374	0.5	0.1	1.0
5	774	1.8	5.5	0.2
6	1574	53.9	–	1:51.9
6x4 Quadrant, 1 MIC				
4	1818	–		
5	3786	1:33.6		
6	7722	–		

Table 1: ILP solver times 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.

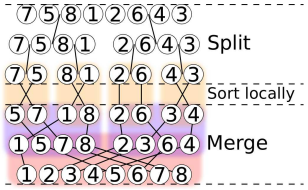


Figure 2: An example of mergesort sorting 8 numbers.

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 the Cell [5]. The focus here is given to 6 cores of one of the SCC's 4 quadrants. All 7 other sets of 6 cores execute in parallel the same actions.

All merging tasks of a 6-levels binary merging tree are allocated to 6 cores, according to an optimal task mapping from the ILP solver (see Section 3.2). The cores who are attributed some of the merging tree's leaves locally sort each input buffers stored in their private main memory. Once all leaves' buffers are sorted, they are merged together as part of the on-chip pipelined merge. To each merging task, a buffer in the respective MPB is attributed to read data from and a 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 the root task that write output to main memory through the L2 cache. Every tasks in the merging tree execute 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 has some free space, then the task merges as much pairs of elements as the minimum between the smallest amount of element available in both inputs buffer and the amount of elements the output buffer can take at the moment.

2. Check and push
   Checks the input buffer of the current task's parent; 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. The root task does not check its output buffer for available space; instead it writes directly to main memory the output of the merger. All tasks mapped to the same core run in a round robin fashion, where the quantum is the necessary time to perform either one or another of the two steps enumerated above.

3.4 Phase 2: parallel sample sort

The second phase consists in merging the 8 subsequences into one final sorted sequence. To this extend a root master is arbitrarily chosen among the 8 roots of the 8 merge trees from the first phase. Each root computes 47 pivots from its sorted subsequence, which divides it into 48 equal chunks. All the 8 roots send their 47 pivots to the root master using the on-chip network. The root master computes 47 global pivots, as the 47 means between all 7 pivots per chunk received from other roots and its own 47 previously calculated pivots. The global pivots are then broadcast back to all roots, and used to cut again the roots' sorted subsequence into 47 chunks, whose lengths are defined by the subsequence's elements and the global pivots received. This process, and the following are depicted in Fig.3. The roots write their subsequence to a buffer in shared main memory (see Fig. 4), at an offset they can calculate from the length of the other chunks to be written. The roots can fetch the length information they need on the root master's MPB. At this point, the buffer in shared main memory contains a sequence of 48 subsequences, where each element in one subsequence is lower than or equals to any element in further subsequences. Every subsequence consists of 8 sorted chunks which, once merged, form the global sorted sequence. All 48 cores get the lengths of all chunks from the main root, so they can calculate the offsets in shared memory from which the 8 chunks begin and finish. From these offsets, sequential merge algorithms can read their input and write their output data. Each of the 48 cores (cores 0 to 3 in Fig.4) runs one sequential merge algorithm simultaneously. After this step the global sequence is sorted.

4. EXPERIMENTAL EVALUATION

We implemented a prototype of the hybrid sort algorithm for SCC described in Section 3. We run it on the 48 cores, varying the total input size from 2^{20} to 2^{24} integers (32 bits each). As a previous work around on-chip pipelined mergesort on the Cell [5] exhibits different performance with available buffer size per core, the sorting algorithm for SCC is run with allocated space on the MPB of 4096, 6144 and 8128 bytes. Each setting in $\{input\_size \times buffer\_size\}$ is run a hundred times to lower side effects from the underlying operating system, and using random data (uniformly distributed unsigned ints) as input. A simple, layered task-to-processor mapping is used, that is, all tasks of one level in the merge tree are mapped to one core. This simple scheme allows the computation load to be balanced perfectly among the cores, but does not minimize data communication and the number of tasks mapped to the same processor. The algorithm is monitored so that timings for the sequential sort (phase 0) and the parallel sort (phases 1 and 2) are available. Results on total times are depicted in Table 2. The first measurements (see Table 2) were obtained using the layer-wise mapping, which is bad for on-chip pipelining, because many producer-consumer tasks pairs need to share the same MPB to forward data. We thus expect that optimized (including ILP optimized) mappings will yield better results. Also, the difference in buffer size only has an influence for small data size, but is negligible for large data sizes.

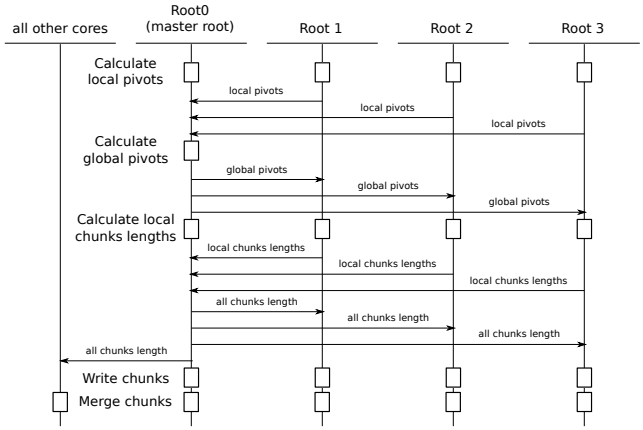


Figure 3: The roots and the master root collectively calculate the chunks' length.

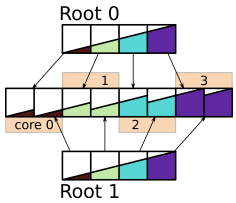


Figure 4: Each root writes its sorted subsequence buffer to shared main memory. For the sake of simplification, this schema assumes a total of 4 cores, 2 of them being the root of a 2-level tree.

The runtime distribution between the phases shows a tendency of phase 0 to grow faster than phase 1, whereas timings for phase 2 tend to decrease from 2^{20} to 2^{25} integers, respectively from 6%, 6% and 87% to 19%, 9% and 71%. This indicates that for bigger amount of data to merge (from 2^{23} integers in our measures), the

Data size	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}
Sequential	10767	23083	49280	104110	220800
Pipelined (total)	1176	2067	4037	6185	10567
Phase 0	58	129	292	653	1481
Phase 1	57	114	226	451	902
Phase 2	781	1561	3120	4418	7036
Speedup	9.1	11.1	12.2	16.8	20.8

Table 2: Runtime in milliseconds for different data sizes with maximum buffer size, using the sequential and pipelined (layer-wise mapping) mergesort.

sample sort in phase 2 gets faster than an on-chip pipelined merge sort, despite the disadvantage of using shared memory.

We have also implemented a sequential mergesort to illustrate the speedup of on-chip pipelined mergesort, compared to its the sequential version. The sequential mergesort recursively splits the array of integers given as input, into smaller chunks that can fit in one third of the cores' L2 cache (256 KiB). Then a sequential quicksort sorts these chunks, and the chunks are merged together again in the merging phase of mergesort. The execution time is measured from the moment when the necessary memory is allocated and input data is fully loaded from files, to when the sorting algorithm finishes, before the sorted array is checked and written to an output file. The performance of this implementation of reference is also depicted in Table 2. Compared to the timing observed with the on-chip pipelined mergesort, we observe a relative speedup growing from 9 with 1 million integers to 20 with 16 millions.

We are not aware of performance results for other sorting algorithms on the SCC. Thus we compare with sorting algorithms implemented on the Cell processor [2]. 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). CellSort [3], 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. While we need about 15,800 ms for the same data size, we have to note that SCC employs cores that run at a lower frequency (factor 4) and with less powerful instructions (about factor 8), so that we achieve a somewhat similar performance with 3 times more cores. Here, we must note however that CellSort is a highly developed code while we provide a prototype.

5. CONCLUSION AND FUTURE WORK

We presented the mapping of a sorting algorithm on the SCC processor as an algorithm engineering study, where we had to combine different algorithmic techniques, knowledge about the processor's memory bandwidth and an elaborate ILP-based load balancing scheme to derive an efficient implementation. Preliminary experimental results indicate that the resulting code is competitive with parallel mergesort investigated in [1] and a relative speedup growing with the input set and up to 20 in our measurements; further work includes the use of an optimal mapping obtained through the load-balancing scheme we describe. The algorithm should be portable to other upcoming manycore architectures, if they have a somehow similar structure, such as e.g. Tilera processors (see `www.tilera.com`).

In future work, we plan to extend this study to further algorithms, and to include energy-efficiency aspects into the mapping algorithm.

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*.

6. REFERENCES

[1] K. Avdic, N. Melot, J. Keller, and 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.

[2] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation - a performance view. *IBM Journal of Research and Development*, 51(5):559–572, Sept 2007.

[3] B. Gedik, R. Bordawekar, and P. Yu. Cellsort: high performance sorting on the Cell processor. In *VLDB '07 Proceedings of the 33rd international conference on Very large data bases*, pages 1286–1207, 2007.

[4] 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, Jan. 2011.

[5] R. Hultén, J. Keller, and C. Kessler. Optimized on-chip-pipelined mergesort on the Cell/B.E. In *Proceedings of Euro-Par 2010*, volume 6272, pages 187–198, 2010.

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