Parallel sorting on Intel Single-Chip Cloud computer

Kenan Avdic[1], Nicolas Melot[1], Jörg Keller[2], and Christoph Kessler[1]

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

Abstract. After multicore processors, many core architectures are becoming increasingly popular in research activities. The development of multicore and many core processors yields the opportunity to revisit classic sorting algorithms, which are important subroutines of many other algorithms. The Single-Chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. It introduces 48 IA32 cores linked by an on-chip high performance network. This work introduces the evaluation of the performances this chip provides, regarding access to main memory and mesh network throughput. It also gives implementations of several variants of parallel mergesort algorithms, and analyzes the performance of the merging phase. The results presented in this paper motivate to use on-chip pipelined mergesort on SCC, which is an issue of ongoing work.

1 Introduction

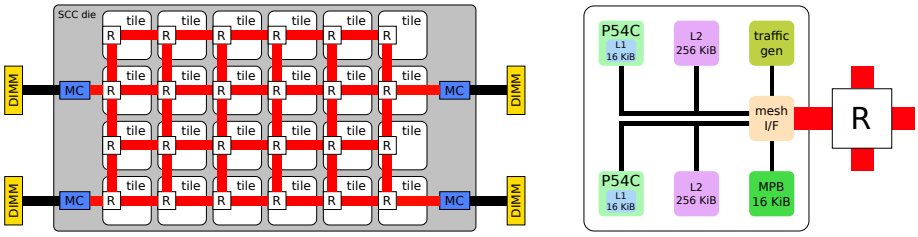
Processor development is on the way toward the integration of more and more cores into a single chip. Multicore processors allow both better performance and energy efficiency [2]. Shortly after the release of the first multicore processors around 2005, research activity now involves *many-core* architectures such as SCC. The Single-Chip Cloud Computer (SCC) experimental processor [3] is a 48-core "concept-vehicle" created by Intel Labs as a platform for many-core software research. Its 48 IA32 cores in a single die are synchronized and communicate through a 2D mesh on-chip network, which also realizes access to the chip-external main memory through four memory controllers. Unlike many smaller scale multicore processors, the SCC does not provide any mechanism of cache consistency for memory shared among the cores.

The raising popularity of such architectures in research yields new considerations over classic algorithms such as sorting algorithms. Sorting algorithms are important subroutines in many algorithms and model the class of algorithms with low compute-communication ratio. Parallelization of sorting algorithms for multi and many-core architecture is an active research area. For example some recent work leverages the SPEs in the Cell Broadband Engine [2] to implement parallel sorting algorithms [4, 6]. However Hultén et al. [5], Keller and Kessler [8] argue that these implementations suffer from a performance bottleneck in off-chip memory access. They propose a pipelined mergesort algorithm relying on Cell's on-chip network capabilities and show in [5] that it achieves a speedup of up to 143% over work from [6] and [4]. This performance improvement comes thanks to a drastic reduction of off-chip memory accesses by forwarding intermediate results from core to core in the pipeline through the on-chip network.

Even if both the SCC and the Cell link their cores through a high-performance internal network, they differ very much in their communication models as well as available memory buffers. Implementing a mergesort algorithm on the SCC could reveal the impact of such differences on actual performance. This paper presents an evaluation of the mappings of the threads to the cores in the chip, and their performance regarding access to main memory or when communicating with each other. It also introduces the implementation, the evaluation and comparison of several variants of the merging phase of parallel mergesort algorithms. Mergesort is relatively easy in its structure; as a memory intensive algorithm, it is a good way for studying the behavior of the processor in case of memory intensive computation. This work does not focus on the development of high performance sorting algorithms for the SCC; rather, the implementations introduced here, in conjunction with the evaluation of on-chip network and main memory, pinpoint where the performance bottlenecks are on SCC and define the direction of further work toward an on-chip pipelined sorting algorithm on this processor. The remainder of this paper is organized as follow: Section 2 gives a quick description of the SCC. Section 3 describes several parallel mergesort algorithm variants making use of SCC's cores' private and shared memory, as well as caching. Section 4 describes the evaluation of the SCC regarding the performance of its on-chip network and main memory access, then it measures the performance of parallel mergesort algorithms. Finally Section 5 discusses the results, introduces the ongoing work and concludes this article.

2 The Single Chip Cloud computer

The SCC is the second generation processor issued from the Intel Tera-scale research program. It provides 48 independent IA32 cores, organized in 24 tiles. Figure 1(a) provides a global view of the organization of the chip. Each tile embeds a pair of cores, and tiles are linked together through a 6×4 mesh on-chip network. A tile is represented in Fig. 1(b): each tile comprises two cores with their cache and a message passing buffer (MPB) of 16KiB (8KiB for each core); it supports direct core-to-core communications. The cores are IA32 (P54C) cores running at 533MHz (they can be accelerated); each of them has its own individual L1 and L2 caches of size 32KiB (16KiB code + 16KiB data) and 256KiB, respectively. Since these cores have been designed before Intel introduced MMX, they provide no SIMD instructions. The mesh network can work up to 2GHz. Each of its link is 16 bits wide and exhibits a 4 cycles latency, including the routing and conveying activity. A group of six tiles share an off-chip DDR3 memory controller that the cores access through the mesh network. The overall system admits a maximum of 64GiB of main memory accessible through 4 DDR3 memory 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). Furthermore, a common small part of the main memory (32 MiB) is shared between all cores; this small amount of shared memory may be increased to several hundred megabytes. Note that private memory is cached on cores' L2 cache but cache support for shared memory is not activated by default in Intel framework RCCE. When activated, no automatic management for cache coherency among all cores is offered to the programmer. This functionality must be provided through a software implementation.



(a) Global organization of the 24 tiles linked by an on-chip mesh network.

(b) A tile in SCC with two cores and their individual memory and MPB.

Fig. 1: Organization of SCC.

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 allows them to communicate data to each other, as well as the management of voltage and frequency scaling. Because of its architecture consisting of a network that links all cores and each core running a Linux kernel, programming on the SCC is very similar to programming parallel algorithms for clusters (for instance) on top of a MPI library.

3 Mergesort algorithms

Parallel sorting algorithms have been investigated since about half a century for all kinds of platforms, sorting being an important subroutine for other applications, and being a good model for non-numeric algorithms with a low computation-to-communication ratio. For an overview cf. [1] or [7]. Mergesort originated as an external sorting algorithm, with data held on the hard disks and merging done in main memory. A similar situation occurs with multiple cores on a chip, connected by an on-chip network, and the main memory being chip-external. Furthermore, mergesort has a simple control structure, is cache-friendly as the blocks to be merged are accessed consecutively from front to end, and is largely insensitive to distribution of values or occurrences of duplicates. For this reason, we chose mergesort for implementation on SCC.

The mergesort algorithm sorts values using a recursive method. The array to be sorted is recursively split into chunks that are trivial to sort then merged together to obtain larger sorted blocks. Figure 2 gives an example of the split/merge phases of the mergesort algorithm, where the split operation is actually trivial. The chunks issued from blocks split can be treated separately and independently. This separation enables parallelization of the algorithm thus leading to a *parallel mergesort algorithm*. When the block has been divided into as many chunks as there are cores available, the chunks are sorted sequentially and then merged pairwise on different cores. Experiments with larger problem sizes show that the merging phase dominates overall sorting time; for

instance, when sorting 32 Mi integers on 32 cores with the fastest of our mergesort implementations, local sorting accounts for less than 10% of the overall time. For this reason, focus will be on the merging phase in the remainder of this paper.

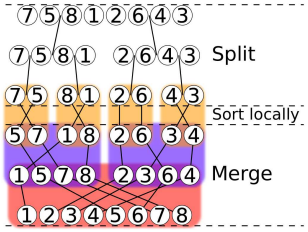


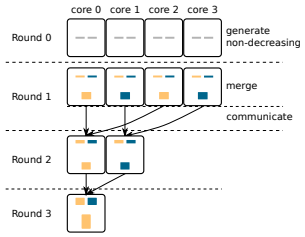
Fig. 2: Data-flow graph of the mergesort algorithm.

The first variant of our parallel mergesort, namely “private memory mergesort algorithm”, is an adaptation to the SCC’s architecture. It uses cores’ private memory to store integer blocks and shared memory to transfer these blocks from a core to another. Each core starts the algorithm by generating two blocks of size $\frac{private\_memory\_size}{4 \cdot number\_of\_cores}$ of pseudo-random non-decreasing values. This size is chosen so that the processor that performs the last recursive merge operation can fit both input and output buffer in its private main memory. Then the algorithm enters a sequence of *rounds* as follows: A core merges its two blocks into a larger one and either sends it to another core or waits for a block to be received. All cores whose ID is lower than or equal to half the number of active cores in this round wait for a block to be received. All other cores send their blocks to a recipient core whose ID is $\lfloor \frac{id}{2} \rfloor$, through shared main memory and then turn idle. At the end of a round, the remaining active core number is divided by two. At the last round, only one core is still active and the algorithm ends when this core finishes the merging of both of its blocks. Figure 3(a) provides an illustration of this algorithm.

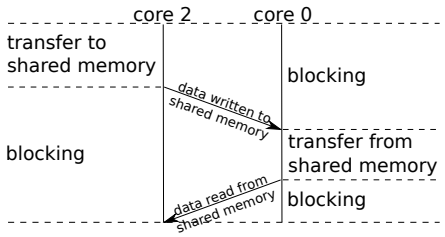
The communication process is shown in Fig. 3(b). Forwarding blocks uses shared memory for buffering: at every round, the shared memory is divided into buffers of size $\frac{2 \cdot total\_shared\_memory}{number\_of\_active\_cores}$. This value maximizes the buffer size for block transfers because half of the active cores have to send a block in a round. Both sending and receiving cores can calculate to which shared main memory buffer they should read or write the block to be transferred, from their ID and active round number. A core sends its block to another core by writing a portion of this block in its dedicated buffer in shared main memory. When this shared memory buffer is full, the sending core sets a flag to the recipient core and waits for it to reply another flag stating it is ready to take more. Both sender and receiver can determine the amount of data to be sent and received from to the total amount of values to be sorted and the number of the active round. Then the process restarts until the whole block has been transferred.

Flags and barriers used for synchronization are sent through the on-chip network. Also in the communication phase, each processor can calculate to what portion of main memory it should read or write. Therefore there is no two cores writing at the same

address in shared memory. For each core writing in shared memory, exactly one other reads the same address after both cores have synchronized using a flag and on-chip network. Finally, the shared memory used for this algorithm is not cached, thus there is no need any shared memory coherency protocol.



(a) Cores merge their blocks into one larger block and then send it to another core to merge it.



(b) One iteration of the block communication process.

Fig. 3: Private memory mergesort algorithm.

A variant of the private memory mergesort algorithm uses the on-die network mesh to communicate blocks between cores. The process of “MPB private memory algorithm” is the same as the one described above but individual core’s MPB memory buffer is used in place of shared memory as transfer buffer. This uses the high speed on-die network and reduces accesses to main memory when it comes to transfer data blocks from one core to another. The other two variants of parallel mergesort are similar to the one described above, except that they work directly on shared memory as storage for input and output blocks. Their implementation here actually relies on SCC’s shared memory. Since they work directly on shared memory, they don’t need any transfer phase in the algorithm. The two shared memory versions of mergesort differ in that one uses L1 and L2 cache and the second one does not use caches at all. Similarly to the private memory algorithm, each core can determine its own chunk in main memory thanks to its ID and the number of the round being run. Thus no coherency protocol is required here. In the case of cached shared memory, the cache is flushed before and after each round in order to make sure that no value cached in a round is considered valid and is read again from the cache in the next round; instead, it is forced to be actually loaded from main memory.

A significant difference between private memory and shared memory variants is that the private memory variant needs a communication phase between all rounds, which requires more or less accesses to main memory, while the shared memory variant does need any transfer phase. This difference gives a performance advantage to the shared memory variants over the private memory based algorithms.

4 Experimental evaluation

We first analyze the influence of the core placement on memory access performance and communication latency and bandwidth, using a smaller part of the algorithm and a microbenchmark, respectively. Then we evaluate the performance of the overall merging algorithm implementations on SCC.

4.1 Access to main memory

In the following, the length of the first merge round is compared in experiments with 2^k cores, where $k = 0,\ldots,5$, and initial block size of 2^{20} integers. This is done for the private memory variant, the uncached shared memory variant, and the cached shared memory variant, as described before. Each setting is run 20 times to reduce influences such as the operating system's scheduler. The averaged time is computed for each core, as well as the minimum, maximum, and average times over all cores. The results are depicted in Fig. 4.

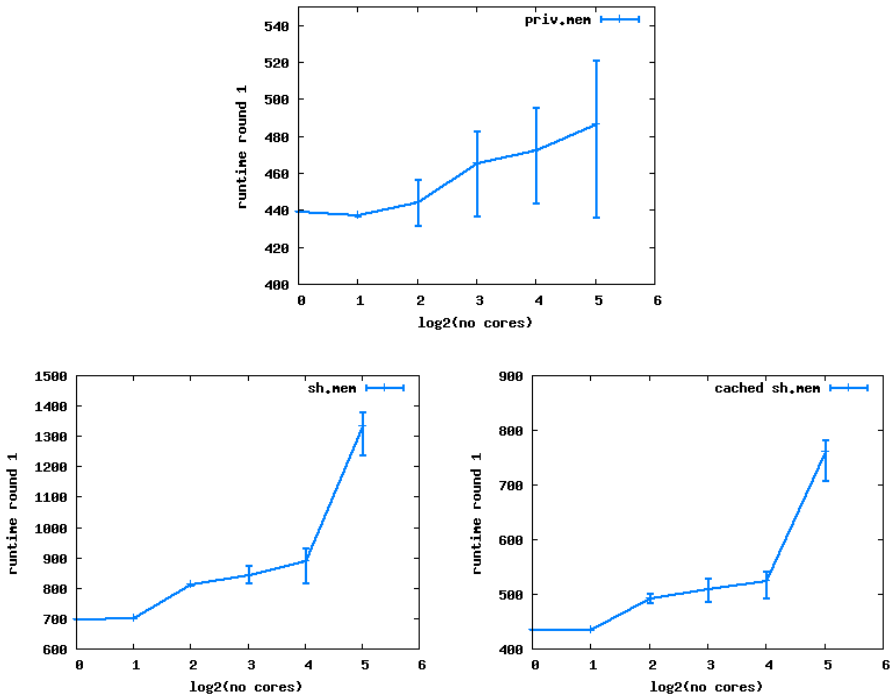


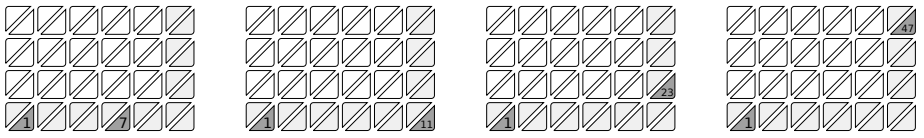
Fig. 4: Runtime differences with increasing core count

The measurements show that placement of cores on the die has a significant impact on performance while accessing main memory or when communicating to each other.

When accessing to main memory, Fig. 6(a) demonstrates that the closer a core is to the relevant memory controller, the faster it is loading or writing data. It shows a linearly growing minimum time bound to complete round 1 of private memory mergesort algorithm. That round is quite an intensive main memory user (private and shared). In each setting, Fig. 4 shows that the runtime increases with the growing number of cores. This indicates contention and/or saturation at the memory controllers, as the code executed is the same in all cases. This effect is very prominent in the shared memory variants. Consequently, on-chip pipelining might help to relieve the memory controllers, especially in applications where the computation-to-communication ratio is low. Figure 4 also shows that the difference between the fastest and the slowest core in a round increases with growing core count, especially for the shared memory variants. This indicates that the distance of a core to the memory controller has a notable influence on the share of the controller's bandwidth that the core gets. Yet, as the distribution of the times is not similar to the distribution of the distances, there must be some other influence as well, which however so far is unknown to us.

4.2 Communication latency and bandwidth

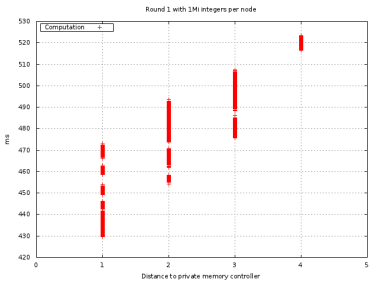
The on-chip network is also of interest, as a future pipelined mergesort implementation would make heavy use of it. The evaluation method proposed here consists in investigating latency by making a core to send several blocks to another core, and monitor the average time these blocks take to be totally transferred. This test is repeated with growing block sizes and with pairs of cores at a growing distance. Figure 5 illustrates how the pairs are chosen to increase the distance to its maximum.



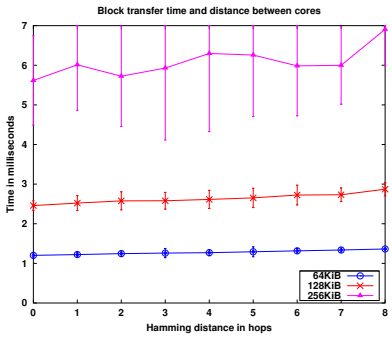
(a) 3 hops between cores.
(b) 5 hops between cores.
(c) 6 hops between cores.
(d) 8 hops between cores.

Fig. 5: Four different mappings of core pairs with a different and growing distance.

Figure 6(b) pinpoints that the communication speed is a linear function of the distance between cores, and the low standard deviation indicates that this speed is stable. However, when the block size reaches 256KiB this linear behavior does not hold anymore and the standard deviation gets much higher. 256KiB is the size of L2 cache and transferring such blocks may generate much more cache misses when storing received values, thus drastically slowing down the communication.



(a) Execution times for round 1 of private memory mergesort algorithm, where nodes manage 1 million integers, depending on the distance from a core to its memory controller.



(b) Average time to transfer 64, 128 and 256 KiB blocks as function of the distance between cores.

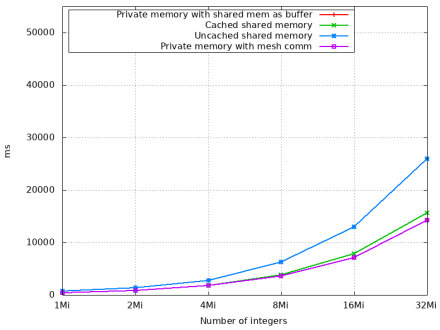
Fig. 6: Impact of the distance from a core to its main memory controller and between two cores.

4.3 Overall mergesort performance

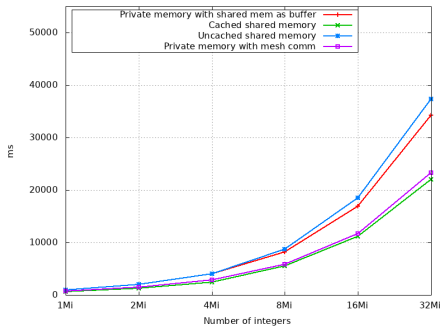
The merging algorithms described in Section 3 are evaluated by monitoring the time they take to sort different amounts of integers. The stopwatch is started as soon as all data is ready and cores are synchronized by a barrier. Then the cores run the algorithm after which the stopwatch is stopped. Every algorithm is run a hundred times with the same amount of randomly generated integers to be sorted and the mean sorting time is plotted in Fig. 7.

The private memory mergesort algorithm requires the blocks to be copied to shared memory, and then to a different private memory. This doubles access to main memory and significantly slows down the private memory mergesort variant. Figures 7(a) and 7(b) show slightly better performance for the private memory variant compared to the cached shared memory variant but this does not scale with number of cores. As soon as more than 2 processors are used, the private memory gets the worst performances of all algorithms implemented. As expected, the private memory variant using the MPB is faster than its equivalent using shared memory to transfer blocks, as it requires less than half the number of main memory access when transferring a block from one core to another. This latter variant competes very well with the cached shared memory algorithm even though it still requires a communication phase between all rounds. The cached shared memory is the fastest of all algorithms studied here.

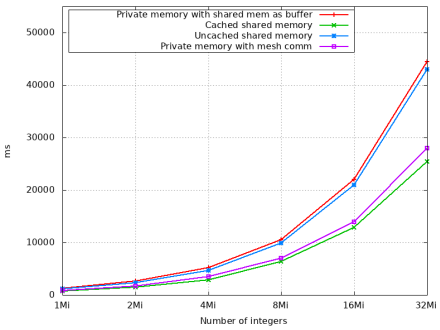
However the good performance of the MBP private memory algorithm shows that transferring data from core to core using MPBs and on-die network brings little penalty



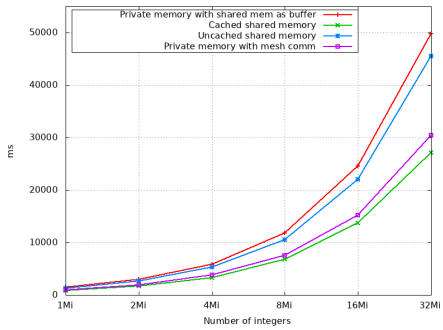
(a) Merging time using 1 processor.



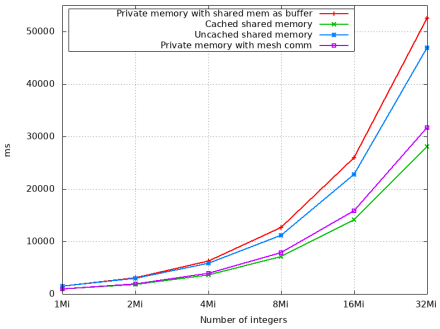
(b) Merging time using 2 processors.



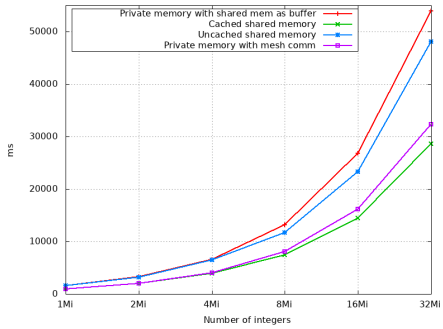
(c) Merging time using 4 processors.



(d) Merging time using 8 processors.



(e) Merging time using 16 processors.



(f) Merging time using 32 processors.

Fig. 7: Performance of the four mergesort variants on SCC, sorting integer blocks with randomly generated data of different size and using different numbers of cores.

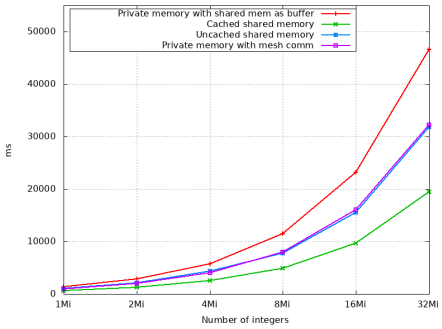


Fig. 8: Performance of the four algorithms merging blocks of constant values, identical among different blocks.

to the mergesort algorithms studied, and much less accesses to main memory. It demonstrates the high capacity of this on-die network, which encourages the implementation of a pipelined mergesort algorithm using this network.

Another setting alters the algorithms to generate blocks of constant, identical value among blocks and merges them using the same four techniques. This different pattern of data to be merge-sorted aims at showing the behavior of the algorithms in an extreme case of input. Constant, identical values in all blocks denote the case of uniform input as well as an input already sorted. The results of this latter setting are shown in Fig. 8. It shows better performance of the mergesort algorithms with an input of already sorted data. However, when compared to Fig. 7(f), the ratio between times of algorithms merging random and constant input blocks is constant and can thus be neglected. Such absence of significant difference is a characteristic of mergesort algorithms, which are insensitive to distribution of values or occurrences of duplicates.

5 Conclusion and ongoing work

Mergesort (and in particular its global merging phase) is an interesting case of a memory-access intensive algorithm. We have implemented four versions of mergesort on SCC, and evaluated their performance. Beyond the mergesort algorithms themselves, our experiments enlighten important factors that influence performance on SCC.

Our results motivate that on-chip pipelining would make sense also for SCC; this is an issue of on-going implementation work. Because the pipeline trades access to main memory for heavier use of the on-chip network, the main memory is accessed only when loading blocks to be sorted and when the final merged block is written back. Also, because off-chip memory access is more expensive than communication through the mesh network, it is expected that the on-chip pipelining technique brings significant speedup over any of the parallel mergesort implementations described here. Its performance may even be improved thanks to awareness of distance between active cores and distance to main memory controllers to generate better task-to-processor mappings.

Acknowledgments

The authors are thankful to Intel for providing the opportunity to experiment with the "concept-vehicle" many cores processor "Single-Chip Cloud computer", and for the reviews provided on an earlier version of this paper. We also thank the A4MMC reviewers for their comments. This research is partly funded by Vetenskapsrådet, project Integrated Software Pipelining.

References

[1] G.S.ja Akl. *Parallel sorting algorithms*. Academic press, 1985.

[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, 2007.

[3] M.A. Corley. Intel lifts the hood on its single chip cloud computer. Report from ISSCC-2010, IEEE Spectrum online, spectrum.ieee.org/semiconductors/processors/intel-lifts-the-hood-on-its-singlechip-cloud-computer, 2010.

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

[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] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-Sort: A new parallel sorting algorithm for multi-core SIMD processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*, 2007.

[7] J. JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., 1992.

[8] J. Keller and C. Kessler. Optimized pipelined parallel merge sort on the Cell BE. In 2^{nd} *international workshop on highly parallel processing on a chip* (*HPPC* – 2008), pages 8–17, 2008.