HAL
open science

Data Sharing Mechanisms for Parallel Graph Algorithms on the Intel SCC

Randolf Rotta, Thomas Prescher, Jana Traue, Jörg Nolte

To cite this version:

Randolf Rotta, Thomas Prescher, Jana Traue, Jörg Nolte. Data Sharing Mechanisms for Parallel Graph Algorithms on the Intel SCC. The 6th Many-core Applications Research Community (MARC) Symposium, Jul 2012, Toulouse, France. pp.13-18. hal-00718993

HAL Id: hal-00718993

https://hal.science/hal-00718993v1

Submitted on 18 Jul 2012

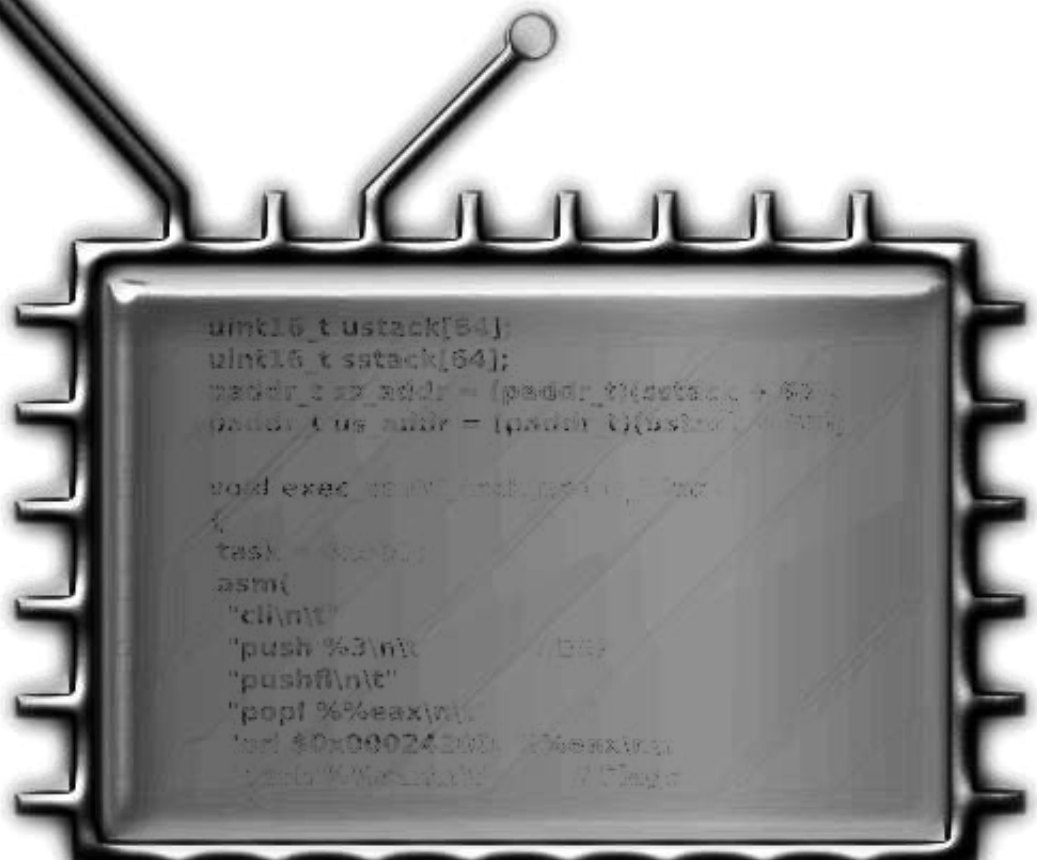
HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium

http://sites.onera.fr/scc/marconera2012

July 19th–20th 2012



ISBN
978-2-7257-0016-8

ONERA

THE FRENCH AEROSPACE LAB

 ISBN: 978-2-7257-0016-8

Data Sharing Mechanisms for Parallel Graph Algorithms on the Intel SCC

Randolf Rotta, Thomas Prescher, Jana Traue, Jörg Nolte
{rrotta,tpresche,jtraue,jon}@informatik.tu-cottbus.de

***Abstract*—On many-core processors that do not provide hardware cache coherence, using shared memory in parallel computations is challenging. Reverting to pure message passing would avoid consistency issues, but replicating large shared datasets by messages is less efficient than accessing them directly through shared memory. The TACO-MESH framework provides lightweight remote method calls and shared objects with software-managed consistency. This paper presents experience from porting a graph partitioning algorithm to the framework. A performance evaluation on the experimental Intel SCC processor, which has no hardware cache coherence, shows that parallelization can be efficient despite the overhead of software-level consistency management.**

***Index Terms*—many-core, shared memory, cache coherence, graph partitioning**

I. INTRODUCTION

The number of cores in current many-core architectures is increasing while the performance of most cores decreases in favor of smaller cores [1]. Technically, many-cores are hybrids combining aspects of distributed as well as shared memory systems [2], because internal networks connect the cores to exchange messages between them and connect to memory modules that provide direct access to shared memory.

Many architectures employ data and instruction caches distributed over the network to reduce access latencies and memory traffic by exploiting locality. On cache coherent architectures (e.g. Intel MIC [3], and Tilera [4]), the caches implement a coherence protocol in hardware. However, coherence protocols face significant scaling issues compared to message passing [5], [6], [7], [8]. The SCC processor is an experimental *concept vehicle* created by Intel Labs as a platform for many-core software research. All of its cores have private caches, but these do, deliberately, not provide hardware-level coherence [9]. Instead, message passing between cores can be used to implement software-level coherence. At the other extreme end are architectures without caches that instead use large shared on-chip memories and a huge number of simpler cores (e.g. IBM Cyclops64 [10], Adapteva [11], and some stream processors in GPUs).

The SCC and other architectures without hardware cache coherence can be treated like distributed systems to work around this limitation. However, passing large messages inside shared memory systems is inefficient, because composing and receiving large messages evicts large portions of the sender's and receiver's caches. Finally, the message data is just copied from and to main memory and will also compete for space in shared caches (e.g. on Intel MIC). Consequently, parallel programming models and frameworks for many-cores should avoid large unnecessary data copies. Instead, software layers that manage the cache coherence for actual shared data should be integrated into the programming models.

TACO [12] provides a partitioned global address space, remote method invocations, and collective operations; it features a highly efficient messaging backend on the SCC. [13]. On top of that, MESH [14], a framework for memory-efficient sharing, introduces direct access to shared data and a consistency layer for shared objects while using TACO for coordination.

This paper presents experience gained from porting central parts of a complex graph partitioning software for modularity clustering [15] to the SCC. The next section introduces the graph partitioning problem, the local search algorithm, and the employed graph data structures. We combined the graph data structures with the MESH consistency layer and extended MESH with SCC-specific software-level cache coherence, which are described in Section III. Section IV discusses results obtained from micro-benchmarks and the parallelized graph partitioning algorithm. Finally, we discuss related work and provide concluding remarks.

II. MODULARITY GRAPH PARTITIONING

Graph partitioning can be applied in the analysis of social, biological and technical networks and is, in this context, also known as *graph clustering*. For example, persons in social networks can be modelled by graph vertices, edges connect pairs of related persons, and edge weights quantify how often both persons interacted in the past. Graph clustering is searching groups of highly related vertices and, in our particular application, a partitioning of the vertices with dense connections within groups and sparse connections in between.

The *modularity* by Newman and Girvan [16] is a popular quality measure that directs the search for interesting vertex partitions. It is based on the difference between the fraction of *observed* within-group edges and the *expected* fraction. The expected fraction is based on a stochastic model where the end-vertices of edges are chosen at random, and the probability that an end-vertex of an edge attaches to a particular vertex is proportional to the vertex weight [17]. Compared to other clustering quality measures, the modularity is still easy to calculate. Nevertheless, in contrast to the more traditional quality measures for load-balancing, modularity has data dependencies that disallow some well-known performance optimizations. Thus, the modularity is a good representative for a broader class of graph partitioning problems.

 ISBN: 978-2-7257-0016-8

```
1: find best v, D, ΔQ_{v,D} over all v ∈ V
   by collective operation over all workers
2: if ΔQ_{v,D} ≤ 0 then exit loop
   // move vertex v from partition C to D:
3: if D is empty then create new partition D
4: set current partition of v to D
5: update partition weights w(C) and w(D)
6: increment partition size of D
7: decrement partition size of C
8: if size of C is 0 then delete partition C
```

Figure 1. A step of the globally greedy vertex moving algorithm.

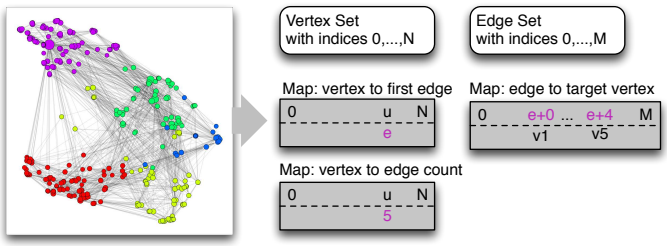


Figure 2. A graph with N vertices, M edges, and three mappings that connect edges and vertices. Highlighted in the mappings is the representation of 5 edges that start in the vertex u and connect to the vertices $v_1 \ldots v_5$.

The next subsection introduces a simple parallel graph partitioning algorithm that will be used for the evaluation of the software-level consistency management. The second subsection describes the data structures used by this algorithm.

A. Globally Greedy Vertex Moving

The problem of finding a clustering with maximum modularity for a given graph is NP-hard [18], and existing exact algorithms are usable only up to a few hundred vertices. In practice, modularity is almost exclusively optimized with heuristic algorithms, and experimental results indicate that relatively simple algorithms based on local search can be highly efficient and effective (e.g. [15]).

One of the simplest local search algorithms is *globally greedy vertex moving* as summarized in Figure 1: In each step, the modularity improvement $\Delta Q_{v,D}$ of moving vertex v to partition D is computed for all pairs of vertices and partitions (line 1) and the globally best move is applied by modifying the partitioning data (lines 3–8). This is repeated until the best move does not increase the modularity (line 2). Faster algorithms exist that are similarly effective but, unfortunately, also more complex. We chose the simplest algorithm to focus on the consistency management of the shared data. Our parallel algorithm uses a set of worker cores to parallelize the computation of ΔQ by dividing the vertex set into equally sized subsets. In each step, the workers compute the improvements of their vertices and return v, D, and $\Delta Q_{v,D}$ of their best move. Then, the master worker selects and applies the best of these moves.

Moving a vertex $v \in V$ from its current partition $C \subseteq V$ to another partition $D \subseteq V$ increases the modularity by

$$\begin{aligned}\Delta Q_{v,D} &= 2\frac{f(v,D) - f(v,C\backslash v)}{f(V,V)} \\ &- 2\frac{w(v)w(D) - w(v)w(C\backslash v)}{w(V)^2} ,\end{aligned}$$

where $f(A,B)$ is the sum of edge weights between two vertex sets and $w(A)$ is the sum of vertex weights in the vertex set. From an algorithmic perspective, this means that the modularity of a partitioning can be quickly updated after each move without recomputing it from scratch. The partition weights $w(C)$ and $w(D)$ of source and destination partition can be updated after each vertex move by using the weight $w(v)$ of the moved vertex.

Moreover, the search space is restricted, because moving a vertex to a non-adjacent partition ($f(v,D) = 0$) never increases the modularity more than moving it to a new, previously empty partition ($w(v)w(D) = 0$). However, the edge weights $f(v,D)$ must be recomputed in each step because storing and updating them is less efficient. At each vertex v, the algorithm scans over its adjacent vertices u and increments $f(v,\mathcal{C}(u))$ by $f(v,u)$, where $\mathcal{C}(u)$ is the partition containing vertex u. A sparse mapping from partitions to accumulated edge weights is used to store $f(v,\mathcal{C}(u))$ and is initialized with zero weights. Internally, each worker uses an own mapping instance for $f(v,D)$, but all workers share the graph, weights, and the current partitioning. Altogether, finding the globally best move requires a constant time per edge and applying a move costs constant time.

B. Data Structures for Graph Partitioning

The original graph partitioning software [15] had to deal with many different algorithms that stored different data about vertices, edges and partitions internally. To handle this diversity, the concept of *Index Spaces* and *Mappings* was introduced to separate navigation through structures from the algorithm's internal data. In general terms, Index spaces represent a collection of indices and methods to navigate over these indices, while mappings are key-value stores that use the indices as keys. This separation allows algorithms to reuse existing spaces and mappings and to create own mappings for internal data on top of these spaces.

Figure 2 depicts one of many methods to model a static graph data structure. The graph has two spaces, namely a set of vertices and a set of edges, and three mappings to connect vertices and edges. The actual index values of the vertices and edges are irrelevant to the algorithms because they are passed to the mappings transparently and only there the index is used to retrieve the corresponding data value. In case the mappings are implemented with arrays, these will work most efficiently, when the vertices and edges are numbered consecutively beginning from zero.

Our implementation is based on the class `RangeSpace` as fundamental index space. It represents a continuous set of indicies from zero to an upper bound and provides a forward iterator as well as methods to increase the upper bound. The mappings are implemented with arrays. Each time the size of

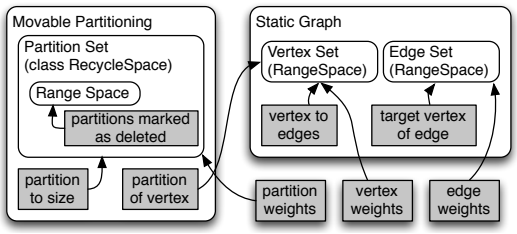


Figure 3. Composition of the graph and partitioning structures. Boxes with round corners are index spaces and all other boxes are mappings. The arrows point to the key space of the mappings.

a space is increased by creating a new index, it is necessary to also resize the arrays of all dependent mappings, which is automated with an observer pattern. In order to minimize the resizing overhead, the maps reserve larger arrays and the space notifies the maps only when the smallest reserve is depleted.

More advanced spaces, for example subsets and graphs, are implemented by using `RangeSpace` and helper mappings. Figure 3 gives an overview of the structures necessary to represent the graphs, vertex partitions, and weights that are used by the graph partitioning algorithm. During the search step, none of these data structures will be modified. To apply the best move, the set of partitions, the mapping from vertices to their current partition, and the partition weights of the source and destination partition will be modified.

III. SOFTWARE-LEVEL CACHE COHERENCE

The MESH framework [14] provides basic facilities for shared memory and shared objects. Its implementation uses the TACO framework [12] to coordinate memory allocation and consistency events. The first subsection describes how we supply MESH with shared memory on the Intel SCC. The second subsection introduces software-level consistency for shared objects and discusses three SCC-specific implementations. Finally, the interactions between shared objects, index spaces, and mappings are discussed in the last subsection.

A. Shared Memory on the Intel SCC

The memory management of MESH is based on separate allocators for shared and core-private physical memory pages and a global allocator for logical pages with aligned addresses over all cores. While most of these allocators are platform independent, a SCC-specific allocator for shared physical memory is necessary. The SCC consists of 32-bit Pentium cores and, thus, each core can address only 4GB of memory. A translation table (LUT) between each core and the on-chip network translates from physical to system addresses that provide a larger address space and contain the network destination (e.g. select one of the four memory controllers). The LUT has 256 entries of 16MB blocks and around 40 entries provide 640MB of private memory for each core.

To acquire direct access to shared memory, it is necessary to make some of each core's private memory accessible to all cores by remapping unused entries in all LUTs. Intel's POPSHM kernel extension provides information about private memory that can be used for this purpose. We tested which LUT entries can be used for the remapping. This approach makes it possible to share 2.5GB of memory between all cores.

The SCC variant of the shared page allocator also provides means to map pages in cached (DCM), non-cached (NCM) and write-through (MPBT+WT) mode. Pages in DCM mode use the core's L1 and L2 caches and require manual coherence management. In contrast, the WT mode uses just in the L1 caches and SCC's write combine buffer to collect writes to a line before sending the modifications to the main memory.

B. Shared Objects with Consistency Management

Special constructors are used to create shared objects. These return a *sharing pointer* that contains the address of the actual object and the address of its *consistency controller* objects. The object is allocated in shared memory but the controllers are allocated in core-private memory at each core. Sharing pointers can be passed between cores using remote method calls or shared memory, because the object resides at the same logical address on all cores and the controller has an instance at the same logical address on each core. Immediate access to shared objects is prohibited. Instead, a temporary *access proxy* has to be created, which triggers consistency events on construction and destruction. The *reader proxies* provide access only to non-modifying (`const`) methods of the shared object, while *writer proxies* allow access to all methods. The consistency events are methods of the consistency controller.

For cache coherence on the SCC it is necessary to invalidate stale data in the caches manually when acquiring read or write access. To ensure that modified data is written back to the main memory before other cores read it, a cache write-back is necessary when releasing write access. Obviously, write-back and invalidation is necessary only when the shared object was actually modified. The next paragraphs present three approaches to implement this coherence management. They support multiple concurrent readers, but only a single non-concurrent writer. Thus, applications have to ensure this concurrency restriction on their own. The structure of our graph partitioning algorithm already guarantees this.

The *Broadcast* (BC) controller has a *needs-flush* flag on each core. When acquiring access to its shared object and the flag is set, the object's memory range is invalidated in the cache. When releasing write access, the cache is flushed to write back all modifications to the main memory and a TACO collective operation sets *needs-flush* on all other cores. To save time the flushing and the collective operation overlap.

In case flushing is much faster than the broadcast, some time can be saved by starting the broadcast earlier. The *Overlapped Broadcast* (OV) controller achieves this by initiating the broadcast already when write access is acquired. As a side-effect, the broadcasts of several modified shared objects can overlap. However, with this approach additional state data for the pending broadcast has to be stored in the consistency controller and the controller could not skip the broadcast if the object was not actually modified.

A completely different invalidation mechanism similar to [19] is used by the *Timestamp* (TS) controller. Each shared object has a timestamp (generation counter) that is stored in the on-chip SRAM of the SCC and the consistency controller on each core has a copy of the last seen value. When releasing write access, the object is flushed to write back all modifications and the object's counter is incremented.

C. Sharing Index Spaces and Mappings

The graph data is implemented with index spaces and mappings (c.f. Section II-B). However, allocating `RangeSpace` as a shared object is not sufficient because it contains a resizable array of pointers to observers (the dependent mappings). This array is invisible to the consistency controller and would be missed when flushing the caches partially. To solve this, shared objects can inherit from the special class `SharingAware` that instructs the consistency controller to also call type-specific consistency management methods. `RangeSpace` uses these to flush the internal observer array.

The mappings use internal data arrays similar to `RangeSpace`. They are managed using `SharingAware`. Furthermore, all mappings have to register themselves as observer at their key space by providing a sharing pointer to themselves. Because a normal shared object does not know its own consistency controller, the `SharingAware` base provides a method to get such a sharing pointer. SCC's small 32-bit address space makes it expensive or difficult to increase the size of arrays. To overcome this, our mapping implementations split the data array into 4kB chunks and use a small lookup array to address these chunks. This allows to increase the maps by adding chunks without moving data.

Three variants of mappings were implemented: The `ChunkedMapDCM` uses cached shared memory for all chunks and the lookup table. The overhead of flushing the relatively large data chunks is avoided by the `ChunkedMapNCM` variant, which still caches the lookup table but not the data chunks. The third variant, `ChunkedMapWT`, caches the data chunks only on the L1 cache using SCC's MPBT+WT memory mode. Any communication by message passing will automatically invalidate cached chunk data and force the write combine buffer to write the modified data back to the main memory. The latter two implementations specialize the writer access proxy to allow write access to the mapping data without triggering unnecessary consistency events.

Composition, that is shared objects using other shared objects, yields an interesting situation: Creating a usual access proxy to the composed object and then calling its methods will be inefficient because each call would have to acquire and release access to the other shared objects, triggering a multitude of consistency management events. To avoid this, we specialize the access proxies of such objects and move the methods from the object into the access proxy. For example, the graph reader access proxy provides the usual methods to navigate through the graph, but it acquires read access to all necessary components just once when the proxy is constructed.

Table I
ACCESS LATENCIES TO SCC'S MEMORY (IN CYCLES).

	on-chip SRAM		off-chip DRAM		
	NCM	MPBT	NCM	MPBT+WT	DCM
w	5	29	11	35	98
g	51–84	78–108	106–137	130–161	107–123
r	58–91	67–97	117–145	123–153	150–174

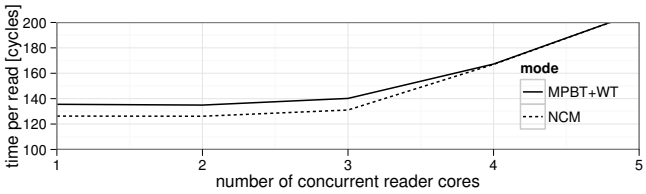


Figure 4. Approximate read overhead as observed by a core with concurrent read accesses to the same controller.

IV. PERFORMANCE EVALUATION ON THE INTEL SCC

The first experiments as presented in the next subsection concerned the impact of the underlying hardware. The second subsection presents and discusses scalability results obtained with the globally greedy vertex moving algorithm on a small and a large graph. All measurements are based on the clock configuration with 800MHz cores, 1600MHz on-chip network, and 1066MHz DDR memory.

A. Shared Memory Performance

Table I summarizes measurements of the memory access latencies following the WGR cost model [13]. The *write overhead* w is the time to issue a write, the *write gap* g is the time until the next write can be issued and the *read overhead* r is the time it takes to fetch data from the memory. For NCM the latencies where measured with scalar values (byte, short, int), while for MPBT, MPBT+WT, and DCM whole cache lines were used. For DCM write performance measurement the lines were read before writing to them because the SCC's caches have no allocate-on-write. Access to the off-chip memory takes around two times longer than to the on-chip memory.

To congest a memory controller, an increasing number of cores read concurrently from the controller. Figure 4 shows the impact on the read latency and congestion is visible above three concurrent readers. Up to three cores do not interfere with each other and, above that, the read overhead increases linearly. Thus, when spreading the data evenly over all four memory controllers, at least 12 cores are necessary to utilize the memory bandwidth. In comparison, the on-chip SRAM can handle more than 15 cores before congestion is visible [13].

The SCC has no hardware mechanism to flush the L2 cache. Thus, a system call has to be used to evict lines by reading other data. The costs of flushing unmodified and modified data are shown in Figure 5. Up to 128 lines the costs increase linearly with 8 000 cycles for one line and $300N + 7\,500$ for N lines. Flushing a 4kB page (128 lines) takes around 47 100 cycles and the entire cache needs 582 000 cycles. Writing back a modified line requires around 80 additional cycles.

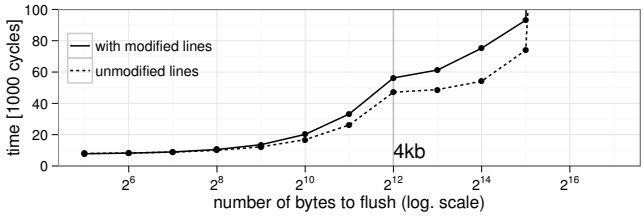


Figure 5. Costs of manually flushing unmodified lines from the L2 cache.

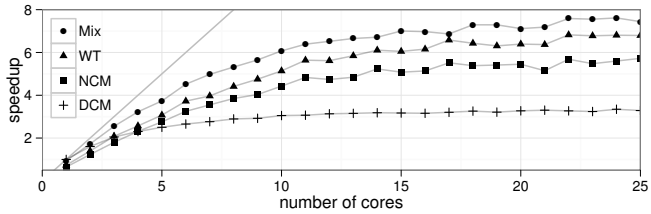


Figure 6. Speedup with the WorldImport1999 graph relative to the single-core DCM-BC variant.

For comparison, a TACO collective operation over all cores of the SCC usually completes within 8 000 cycles [13]. Thus, this cache flushing always takes much longer than the invalidation broadcasts and, therefore, the differences between the three consistency controllers of Section III-B are negligible.

B. Globally Greedy Vertex Moving

For a varying number of cores, the algorithm was applied on a relatively small graph that represents world trade relations in the year 1999. The graph has just 66 vertices and 4290 edges [20]. The local search took 64 steps and we measured the overall time of these steps without the program initialization and reading the data file. The measurements considered the three map implementations DCM, NCM, and WT from Section III-C and a *Mix* variant, which uses DCM static data (e.g. vertices, edges, weights) and WT for maps that are modified by vertex moves (e.g. partitions, partition weights).

To compare the measurements, Figure 6 shows the speedup factors relative to the DCM variant on a single core. Only the results with the Broadcast consistency controller are shown, because the other two controllers were similar. Using only DCM maps, the local search scales really poorly. This is probably caused by the huge cache flushing overhead at the master and the worker cores. With NCM maps better speedups are achieved because almost all of the cache flushing is eliminated. However, this variant does not use any caching and indeed seems to utilize the memory bandwidth when reaching 12 cores. The WT maps improve on this because they use the L1 caches and achieves speedup 6 with 13–14 cores. The *Mix* variant produced the best results with speedup 7 at 15 cores. More importantly, for small core counts the speedup was consistently better than with the other variants.

Amdahl's law provides an upper bound to the achievable speedup. It is based on the parallel runtime $T(P) = T_s + T_p/P$ and the speedup $T(1)/T(P)$, where T_s is the sequential work,

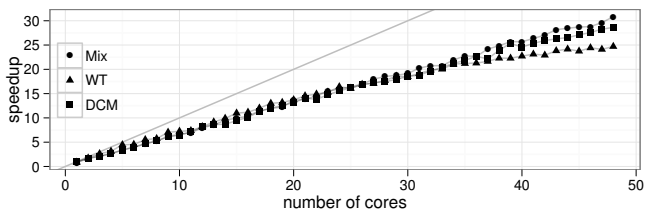


Figure 7. Speedup with the Patents graph. All speedup factors are relative to single-core DCM-BC.

e.g. applying the best vertex move, and T_p/P is the parallel work with P cores, e.g. evaluating the modularity gains of all vertices. However, overheads impact the speedup, leading to

$$T(P) = T_s + T_p/P + T_{wb} + T_{inv} + \alpha P + \beta \log_2(P) \ .$$

After each vertex move, the master worker has to write back all modifications to the main memory, which costs T_{wb} in total. Then, all workers invalidate these memory ranges in their caches with overhead T_{inv}. Because this flushing is done in parallel, it can be attributed once to the sequential costs. The workers might perform some work that is effectively sequentialized by memory bottlenecks. This can be modeled by αP because this sequential work increases with the number of workers. Finally, the workers are invoked in each step by using a multicast tree, which introduces a logarithmically growing coordination overhead $\beta \log_2(P)$.

Fitting linear models on the measurements indicated that the coordination overhead $\beta \log_2(P)$ is negligible, while αP is necessary to explain the low slope at low core counts. The NCM, WT and Mix variants eliminate $T_{wb} + T_{inv}$, and WT and Mix reduce αP through caching. To improve the speedup bound, it is necessary to reduce αP by increasing the memory bandwidth or reducing the cache misses. The latter requires caching of more data by using DCM maps, which is only efficient if the flushing overheads $T_{wb} + T_{inv}$ can be reduced.

With 66 vertices, the world trade graph is quite small, e.g. each worker is responsible for just 5 vertices when using 12 workers. Thus, we repeated some of the measurements with the much larger NBER U.S. patent citations graph, which has 240 547 vertices [21]. Figure 7 compares the runtime of the first 500 steps. Here, the task size was large enough to dominate even the flushing overheads of the DCM variant.

V. RELATED WORK

Two other shared memory mechanisms are available on the SCC. POPSHM provides a simple put/get copy-based interface to access shared memory. It does not remap unused LUT entries but uses a few as read/write buffer in NCM mode. The SMC library supports allocation of shared pages, changing the access modes, and provides release consistency with configurable consistency domains. Both libraries do not use remote method invocation mechanisms, which limits their flexibility. In contrast, we implemented the shared memory management and consistency control together with the application on top of a common messaging subsystem.

Implementing scaleable cache coherence protocols is still challenging because directories grow with the cache size and the number of cores [5]. Some try to balance parameters to reduce the directory size while also keeping the coherence traffic low [22]. In comparison, we omitted the directories and used broadcasts at very coarse granularity by exploiting the algorithm's structure. Other projects exploit typical sharing patterns to compress the directories [23], [24]. Software-level coherence is an interesting alternative because it can incorporate knowledge about the application at design time and can have better performance than hardware coherence in some cases [7]. A promising mixture of both approaches is the Cohesion memory model of Kelm et al. [6].

VI. CONCLUSIONS

We presented a framework for software-level cache coherence on the SCC. A simple parallel graph partitioning algorithm was used to evaluate the impact of the software-level cache coherence and the speedups achievable through parallelization. The experiments showed that considerable speedups are possible depending on the problem size despite non-negligible cache flushing overheads. The results could be improved by better hardware support for manual cache control. The presented framework would benefit from a write-back and a write-back-invalidate instruction on logical address ranges. On architectures with a shared cache level, the write-backs and invalidations would be necessary only on the private levels. Porting the data structures from an existing application was mostly straightforward, but the composed structures had to be changed considerably to interact efficiently with the consistency framework. From a software engineering perspective this might actually not be a drawback, because moving operations on shared data into the access proxies also decouples independent types of operations. For example, the graph initialization methods were implemented in a separate access proxy.

ACKNOWLEDGMENTS

We thank Intel for the access to the SCC and MARC program. In particular, we thank Michiel W. van Tol (University of Amsterdam), Werner Haas (Intel Research Braunschweig), and Jan-Arne Sobania (HPI Potsdam) for implementing the software-based L2 cache flushing. This research would not have been possible otherwise.

REFERENCES

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, May 2011.

[2] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011), Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, Istanbul, Turkey, July 2011.

[3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, pp. 18:1–18:15, 2008.

[4] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, 2007.

[5] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, vol. 23, no. 6, pp. 49–58, Jun. 1990.

[6] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: a hybrid memory model for accelerators," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10, 2010, pp. 429–440.

[7] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon, "Comparison of hardware and software cache coherence schemes," in *Proceedings of the 18th annual international symposium on Computer architecture*, ser. ISCA '91, 1991, pp. 298–308.

[8] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44.

[9] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha, "A Case for Software Managed Coherence in Many-core Processors," Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10, 2010.

[10] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. R. Gao, "A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 64–64.

[11] A. Olofsson, "A 1024-core 70 GFLOP/W Floating Point Manycore Microprocessor," Poster on 15th Workshop on High Performance Embedded Computing HPEC2011, 2011.

[12] J. Nolte, Y. Ishikawa, and M. Sato, "TACO – Prototyping High-Level Object-Oriented Programming Constructs by Means of Template Based Programming Techniques," *ACM Sigplan, Special Section, Intriguing Technology from OOPSLA*, vol. 36, no. 12, December 2001.

[13] R. Rotta, T. Prescher, J. Traue, and J. Nolte, "In-memory communication mechanisms for many-cores – experiences with the Intel SCC," in *TACC-Intel Highly Parallel Computing Symposium (TI-HPCS)*, 2012.

[14] T. Prescher, R. Rotta, and J. Nolte, "Flexible sharing and replication mechanisms for hybrid memory architectures," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam*, vol. 55, 2012, pp. 67–72.

[15] R. Rotta and A. Noack, "Multilevel local search algorithms for modularity clustering," *J. Exp. Algorithmics*, vol. 16, pp. 2.3:2.1–2.3:2.27, Jul. 2011.

[16] M. E. J. Newman, "Analysis of weighted networks," *Physical Review E*, vol. 70, p. 056131, 2004.

[17] ——, "Finding community structure in networks using the eigenvectors of matrices," *Physical Review E*, vol. 74, p. 036104, 2006.

[18] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 2, pp. 172–188, 2008.

[19] S. L. Min and J.-L. Baer, "Design and analysis of a scalable cache coherence scheme based on clocks and timestamps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 1, pp. 25–44, Jan. 1992.

[20] A. Noack, "Example graphs from the LinLogLayout tool," http://www-sst.informatik.tu-cottbus.de/~an/GD/, 2008.

[21] V. Batagelj and A. Mrvar, "Pajek datasets," http://vlado.fmf.uni-lj.si/pub/networks/data/patents/Patents.htm, 2006.

[22] A. Gupta, W.-D. Weber, and T. C. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Proceedings of the 1990 International Conference on Parallel Processing (ICCP)*, vol. 1: Architectur, 1990, pp. 312–321.

[23] H. Zhao, A. Shriraman, and S. Dwarkadas, "Space: sharing pattern-based directory coherence for multicore scalability," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10, 2010, pp. 135–146.

[24] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 423–434.