Middleware for Many-Cores - Why it is needed and what functionality it should provide

Randolf Rotta, Steffen Büchner, Jörg Nolte
Brandenburg University of Technology
Platz der Deutschen Einheit 1
03046 Cottbus, Germany
{rrotta,sbuechne,jon}@informatik.tu-cottbus.de

Abstract

Today's multi-cores and future many-cores are NUMA architectures with complex cache hierarchies and multiple memory channels. Depending on the topologies of these memory networks we find everything from true data sharing with shared caches to distributed memory architectures which just pretend to be physical shared memory systems. In fact, most many-cores are hybrid systems that exhibit the characteristics of both distributed systems and SMPs. In this paper we argue in favor of middleware platforms for many-cores. We will discuss the needed functionality in contrast to common distributed system middleware and present micro benchmarks on several architectures to substantiate our claims.

1 Introduction

The numbers of cores explode in today's architectures and we might expect single board systems or even single CPUs with core counts above 1000 soon [15]. When we have a look at hardware diagrams we'll find complex cache hierarchies and multi-hop internal networks with highly varying latencies. In fact, even though all commercial architectures like the AMD Interlagos (16 cores), the upcoming Intel Knights Corner (over 50 cores plus hyperthreads) or the Tilera TilePro100 (100 cores) still provide cache coherency, these CPUs are nevertheless distributed hardware systems by nature and merely pretend to be shared memory systems. Programs that do not take the peculiarities of such architectures into account are likely to run into a variety of pitfalls. In particular, naive sharing leads to severe problems which caused the designers of the Corey [3] operating system to give applications control over the sharing semantics of OS services such that the cost of sharing can be avoided when it is not needed.

The designers of Barrelfish [17] followed an even more radical approach: the multikernel architecture of Barrelfish is based on shared nothing semantics even on hardware systems that allow some means of sharing. It turned out that in several scenarios shared accesses through remote procedure call (RPC) mechanisms based on shared memory message passing could yield better performance than direct shared memory accesses [1]. The reasons for this effect can be credited to the applied function shipping approach. RPCs cause a high degree of cache utilisation at the server's core and effectively eliminate lock contention or cache thrashing. The authors of [18] offload certain critical sections and synchronisation mechanisms to dedicated faster cores to speed up parallel computations.

Function shipping approaches can be considered as an effective means for many-core programming and should not been neglected. However, even though these topics have been discussed in the distributed computing community for decades the situation in homogenous many-cores is substantially different from loosely coupled and possibly heterogeneous distributed systems. In this paper we discuss the needed characteristics of middleware layers for many-cores and substantiate our findings with micro benchmarks for a variety of architectures.

2 Cross-Core Method Calls

Most distributed systems are constructed around the notion of distributed objects. The object paradigm combines communication and computation in a single, easy to understand model that allows data to be addressed without caring about the location. The term *distributed* means in the context of many-cores that even though any thread might be able to access objects directly in its address space only the core that is responsible for a specific object will (usually) touch

it directly. All other cores apply cross core invocation (CCI) mechanisms to access shared resources with minimized cache thrashing and lock contention.

CCI mechanisms are semantically similar to remote procedure calls (RPC) or remote method invocation (RMI). However, unlike common RMI mechanisms they primarily have to provide ultra-low latencies. Complex parameter marshalling mechanisms are usually not needed, because fast CCI callbacks and hardware sharing facilities are available on many-cores.

RMI mechanisms for distributed systems were designed for relatively slow networks that connect fast computers. Such scenarios are rather forgiving to software inefficiencies because the network latencies usually dominate overall invocation costs drastically. On many-cores where we find ultra-fast networks but also less powerful cores the situation is inverted and low-overhead mechanisms become a priority to achieve low latency. Thus, middleware platforms that were primarily designed for parallel processing are a good starting point and, in particular, C++ template libraries like ABC++ [14], MTTL [11] or Taco [13] (amongst others) already provide useful invocation mechanisms. Taco was easy to port to many-core processors and, hence, will serve as reference for our discussion. However, these mechanisms are not restricted to C++ and can of couse be implemented differently, e.g. like in X10 [5].

```
01 // create "Account" object @ core where
02 ObjectPtr<Account> obj;
03 obj = allocate<Account>(where)(args);
04
05 // call "transfer" asynchronously
06 obj->apply(m2f(&Account::transfer, 80));
07
08 // synchronous method call
09 int r = obj->invoke(m2f(&Account::balance));
10
11 // deferred synchronous method call
12 Future<int> sync; // future result
13 obj->apply(m2f(&Account::withdraw, 17), sync);
14 ...
15 r = sync; // implicitly wait for result
```

Figure 1: Basic CCI Mechanisms

In Taco each thread can create (fig. 1, line 3) and access instances of arbitrary C++ classes on other cores by means of template based CCI mechanisms. Objects being managed at other cores are referenced by *global object pointers* (fig. 1, line 2) which are small tuples that consist of a core number and a pointer to the object's state. These pointers support polymorphism according to C++ type rules and can be passed as parameters to implement true reference semantics. Since global pointers are much smaller than e.g. serialized proxy objects, most CCI messages fit into a single cache line on most architectures. The state of a frequently accessed object will only reside inside the cache of the responsible core and will *never* move into a cache that is private to another core as long as it is accessed solely through CCIs. Thus we have a deterministic means for cache control which is orthogonal to cache coherence algorithms and will perform correctly even on machines without any hardware cache coherence like the experimental Intel SCC [19]. Parallelism is provided through asynchronous (fig. 1, line 6) and deferred synchronous calls (fig. 1, line 13). In the latter case future objects [10] are used (fig. 1, line 12) to wait implicitly for the arrival of the result from the call (fig. 1, line 15).

Technically, Taco's CCI mechanisms rely on the shipment of function objects (so-called *functors*) that are generated automatically (fig. 1, lines 6, 9, 13). The `m2f()` generator wraps a pointer to a method including its actual parameters into a function object that can be transfered to other cores and applied to compatible objects. Currently, message reception is based on polling only. However, polling can be very cheap (sec. 4). On systems that provide hyperthreads a dedicated thread could perform these actions in parallel. Instructions like the `mwait`-instruction are useful to put a core or hyperthread into sleep mode as long as there is no incoming communication.

For local latency hiding, the execution of the method invocations are interlaced by lightweight, cooperative threads to minimize synchronization costs. Latency hiding techniques are necessary, when CCI mechanisms are synchronous. In an event-driven system that e.g. provides call mechanisms with continuations, latency hiding mechanisms are not necessary and the overhead of (user-level) thread switching could be avoided. However, asynchronous systems can be hard to program. Therefore, other latency hiding means such as pseudo blocking protothreads [7] are suitable alternatives.

In Taco all invocation mechanisms are implemented by means of active messages [20] on top of a communication layer that provides non-blocking point-to-point message passing for short messages (only a few cache lines). This simple communication abstraction enhances portability greatly. However, this approach reaches its limit (sec. 4) on networks with extremely low latencies. The layered structure induces small but not negligible copy costs that should be avoided e.g. with a suitable cross-layer design.

3 Group Abstractions

The basic CCI mechanisms sketched in the previous section allow for simple task-parallel computations but common tasks in many-core processors require more convenient means for parallel coordination. For example, consider a multikernel that has to implement a virtual memory system. Each time, a page needs to be evicted, every core that previously had a mapping for that page has to invalidate the respective entries in its translation lookaside buffer (TLB). Such activities are also necessary in monolithic SMP kernels since TLBs are typically not coherent on most architectures. This example requires managing varying groups of cores and addressing the group members collectively in a scalable manner. Figure 2 sketches such a scenario with Taco's object groups.

```
01 // The group will have a quad tree topology
02 class AddrSpace:
03    public QuadTree<AddrSpace> { ... };
04
05 // find out my ancestor's address space
06 GroupPtr<AddrSpace> all = ...;
07
08 // join my managing instance to the group once
09 all->call(m2f(&AddrSpace::join, myAddrSpace));
10 ...
11 // make a page accessible everywhere
12 all->map(m2f(&AddrSpace::mapIn, page, frame));
13 ...
14 // invalidate all entries synchronously
15 all->step(m2f(&AddrSpace::invalidate, page));
16 ...
17 // determine number of existing mappings
18 int n = all->reduce(
19               m2f(&AddrSpace::isMapped, page),
21               std::plus<int>()
22            );
```

Figure 2: Group-Based CCI Mechanisms

The construction of groups is automated by predefined *topology classes*, which can be inherited by the classes of member objects (fig. 2). They provide a `join()` method (fig. 2, line 9) to add new members dynamically. In most cases groups have a hierarchical tree topology such as a `QuadTree` in the example in figure 2. The root of the tree acts as the group's leader such that a global pointer to the leader represents the entire group. *Collective* method calls are implemented by means of a parallel visitor pattern [8] starting at the root. Each visitor functor first forwards itself to the descendants of the currently visited member by means of asynchronous CCI mechanisms and then performs its operation locally on the currently visited object. Therefore, the execution of the visitor is effectively parallelized, too. Descendants are determined by an iterator provided by the topology class. Thus, topology classes offer the unique opportunity to control the order of execution and degree of parallelism of collective operations with language-level inheritance mechanisms.

The `map()` operation (fig. 2, line 12) initiates asynchronous object parallel computations by applying a *void-functor* to all group members while `step()` (fig. 2, line 15)is executed synchronously. The synchronous `reduce()` operation (fig. 2, line 18-22) first applies the specified *functor* to all group members and then combines all results with an associative and commutative binary *reduce-functor*, e.g. the `plus()` functor (fig. 2, line 21) of the STL. Other operations such as `gather()` and `scatter()` are available, too. All collective operations can be masked or applied to predetermined member selections. The execution is strictly in-order for all operations applied to the same group. Thus, synchronous operations can implicitly act as fence-operations that will complete when all previously started operations have completed.

The group concept discussed here builds on the more simple CCI mechanisms (section 2), which helps to enhance portability and keep the code base concise. The collective operations scale well and provide fairly low latencies (refer to section 4). However, there are some drawbacks. In particular Taco cannot use (future) hardware based multicast mechanisms due to the inherent point-to-point nature. Furthermore, the layered design again causes unnecessary memory copies that are negligible on average distributed systems but not on many-cores. Last but not least the group leader concept in combination with strict in-order functor-propagation eases consistency issues but might become a bottleneck when a large number of cores addresses the same group simultaneously.

4 Micro Benchmarks

We ported Taco to many-cores by replacing the MPI-based communication by messaging over shared memory, and performed micro-benchmarks on the three systems summarized in figure 3. The 32 core Intel Xeon E7 (2.13GHz) and 8 core AMD Opteron (2.3GHz) systems used a protocol that was optimized for the Opteron. On the non-cache coherent 48 core Intel SCC (0.8GHz), its experimental hardware for inter-core message passing was used.

The first benchmark measured the CCI roundtrip time between the nearest and the most distant pair of cores. The messages were up to two cache lines long,

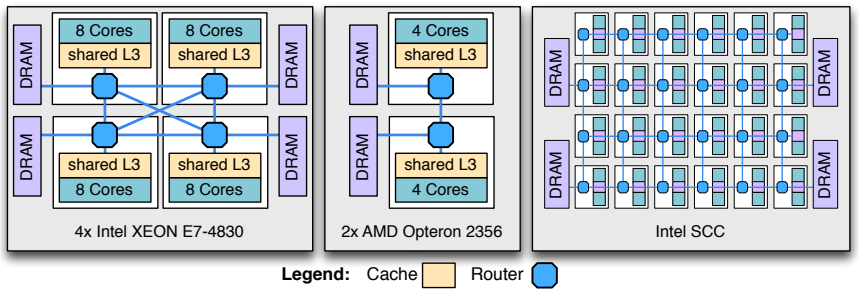

Figure 3: Overview of the network on the three test systems.

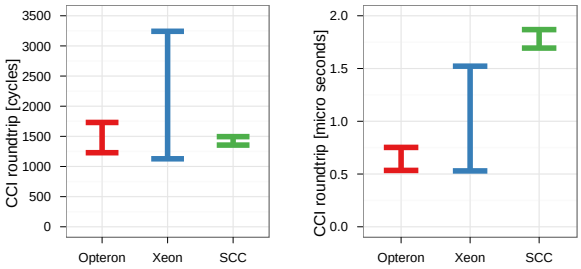

Figure 4: CCI roundtrip times ranging from nearest to farthest core.

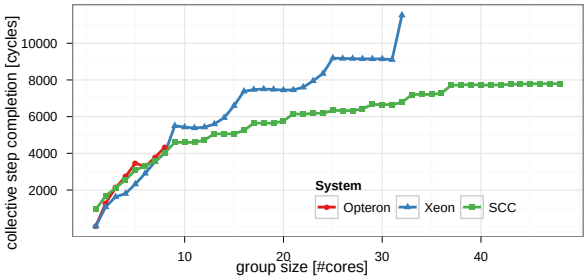

Figure 5: Completion time of collective steps.

which is typical for remote method calls with few arguments. The results are shown in figure 4. For comparison, the CCI roundtrip times were around 3000 cycles in average with MPICH2 on the Xeon. The Opteron and Xeon systems showed a large variation depending on the network distance. While on the Xeon the within-socket communication was slightly faster than on the Opteron, the inter-socket latency seems to be much higher. Up to 50% of the roundtrip time was influenced by the network distance and, thus, the network topology is important on multi-socket systems. Better protocols for the Xeon might decrease the gap. However, then the optimal protocol as well as any efficient data exchange highly depends on the hardware. While middleware-based systems can be adapted by replacing the communication layer, such tuning would not be feasible with arbitrary shared memory code. In contrast, on the SCC just 10% of the roundtrip time was influenced by the distance because SCC's mesh network has very homogeneous link latencies and just a small portion of the memory accesses of the protocol actually go to remote memory. Therefore on future many-core systems, the network topology may be negligible in respect to point-to-point latencies.

The second observation is a huge discrepancy between the raw network latency and the CCI roundtrip times. All systems need >1000 cycles for a roundtrip while the actual payload transfer costs just <400 cycles.[1] This is caused by inevitable memory accesses for notifications and acknowledgements inside the protocols. On cache coherent systems like the Opteron and Xeon this can also involve atomic operations with costly coherence activities. Ideally, a message roundtrip should not cost much more than the actual data transfers and this will require faster notification mechanisms between the cores.

The issue becomes more evident with respect to scalability. Protocols on multi-core systems typically used separate pairwise channels to avoid costly cache conflicts, e.g. [4]. This leads to quadratically growing memory requirements and a linearly growing polling overhead. In contrast, the protocol on the SCC achieves linearly growing memory requirements and constant polling overhead (130 cycles) by using atomic increment counters provided by the hardware to dispatch concurrent senders.

Finally on the SCC, the roundtrip time is strongly influenced by protocol overheads because a third of the roundtrip time (300 cycles) can be attributed to the upper CCI layer.[2] This could be reduced by em-

[1] Opteron and Xeon: <2*200 cycles for remote cache line access [9]. SCC: <2*190 cycles for on-chip memory access.

[2] The sender constructs the message in his L1 cache (<50

ploying optimizations based on static code analysis.

In the second benchmark, the completion time of collective operations over an increasing subset of the cores was measured. An appropriate tree topology was used on each system. Figure 5 presents the results. On the Xeon and SCC systems, the completion time grows logarithmically as can be expected from a tree topology. This is better visible on the SCC, because the inter-socket latency on the Xeon introduces additional costs. By design, the communication overhead per core is constant and, thus, other work can be done while the operation is propagated over the cores. In conclusion, collective operations on many-core systems are scalable and their efficiency depends on the efficiency of basic point-to-point mechanisms.

5 Related Work

Both Corey [3] and Barrelfish [17] are operating system approaches that could benefit greatly from a suitable inter core middleware layer. At least within the Barrelfish system such a layer is mentioned but unfortunately details except about the underlying communication protocol [16] have not been published yet.

Taco itself has been strongly inspired by ABC++ [14] and the MTTL [11]. E.g. the MTTL already provided means for general purpose parallel object oriented programming with global pointers but did neither support polymorphism nor collective operations. Template technology can be successfully applied to prototype parallel programming concepts without the initial need to design new languages before the concepts have been accepted. Thus, many programming concepts found today in Split-C [12], UPC [6], X10 [5] or Cilk [2] can in principle also be achieved with C++ templates. However, the programming language approach generally offers better means for optimization and safety through static analysis. This is not possible in template libraries, since a standard C++ compiler has no notion about parallel constructs and their associated semantics.

Generally, even though we use the term *middleware layer* for Taco, our approach is not suited for programming across security boundaries due to the use of active messages for communication. Taco can in principle be used within a multikernel like Barrelfish, in runtime libraries like for X10, and any parallel services but not to interface these services to clients.

cycles) and after the transmission the receiver copies it into a receive queue in his L1 cache (100 cycles).

6 Conclusion

Current many-cores are internally distributed systems that should at least in part be treated as such to avoid cache thrashing effects and lock contention. Therefore, middleware platforms are required that are able to address this aspect appropriately. Ultra-low latency cross core invocation mechanisms are needed that have a similar overhead as a cache-line roundtrip time on current architectures. However, the experiments with Taco have shown that low latencies can be achieved in principle (compared to cluster architectures) but the final goal is still in far reach. Even though Taco is just a thin (mostly inlined) software layer above the basic communication layer, communication costs do not dominate CCI costs any more. Further achievements could only be reached by giving up the current layered architecture to avoid unnecessary copying of data between layers. Purely event-driven invocation mechanisms in conjunction with protothreads can potentially yield better performance than the currently used cooperative threads. Last but not least certain communication activities like polling for messages and forwarding of messages to other cores during collective operations can be offloaded to dedicated hyperthreads.

Collective operations play an important role to initiate parallel computations and keep replicated data consistent. The basic collective operations of Taco scale well on the mesh network of the Intel SCC and reasonably well on the fully connected cache-coherent systems. For such systems optimized topology classes need to be designed. In-memory communication mechanisms that are based on internal cache-coherence protocols are generally hard to control and need to be adapted carefully to each new architecture. The relatively good performance of the SCC indicates that future many-cores should provide communication means that are clearly independent from hardware coherence mechanisms.

References

[1] 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*, SOSP '09, pages 29–44, 2009.

[2] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal*

of Parallel and Distributed Computing, 37(1):55–69, August 25 1996.

[3] S. Boyd-Wickizier, H. Chen, R. Chen, Y. Mao, M. Frans Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 43–57, 2008.

[4] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud. Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 462–469, 2009.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[6] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 36–47, 2005.

[7] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 29–42, 2006.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA, 1995.

[9] D. Hackenberg, D. Molka, and W.E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 413–422, 2009.

[10] R.H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

[11] Y. Ishikawa. Multiple threads template library. Technical Report TR-96-012, Real World Computing Partnership, 1996.

[12] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 262–273, 1993.

[13] 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*, 36(12), December 2001.

[14] William G. O'Farrell, Frank Ch. Eigler, S. David Pullara, and Gregory V. Wilson. ABC++. In *Parallel Programming using C++*, pages 1–42. MIT Press, 1996.

[15] John D. Owens, William J. Dally, Ron Ho, D. N. (Jay) Jayasimha, Stephen W. Keckler, and Li-Shiuan Peh. Research challenges for on-chip interconnection networks. *IEEE Micro*, 27:96–108, September 2007.

[16] S. Peter, T. Roscoe, and A. Baumann. Barrelfish on the Intel Single-chip Cloud Computer, technical note 005. Technical report, 2010.

[17] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the International Workshop on Managed Many-Core Systems (MMCS'08)*, 2008.

[18] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. *SIGPLAN Not.*, 44:253–264, March 2009.

[19] R.F. van der Wijngaart, T.G. Mattson, and W. Haas. Light-weight communications on Intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45:73–83, February 2011.

[20] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. Technical Report UCB/CSD 92/675, University of California, Berkeley, CA, 1992.