Low-Latency Collectives for the Intel SCC

Adan Kohler, Martin Radetzki
Institute of Computer Architecture and Computer Engineering
University of Stuttgart
Stuttgart, Germany
{kohleran, radetzki}@informatik.uni-stuttgart.de

Philipp Gschwandtner, Thomas Fahringer
Institute of Informatics
University of Innsbruck
Innsbruck, Austria
{philipp, tf}@dps.uibk.ac.at

***Abstract*—Message passing has been adopted as the main programming paradigm for many-core processors with on-chip networks for inter-core communication. To this end, message-passing libraries such as MPI can be used, as they provide well-known interfaces to application developers. Since MPI implementations were originally developed for macroscopic computer networks, the different characteristics of on-chip networks may require rethinking existing solutions. With the example of *Allreduce*, we identify points where collective operations benefit from routines optimized for on-chip networks. The identified issues are then applied to additional collectives including *Broadcast*, *Allgather* and *Alltoall*. The effectiveness of the proposed optimizations is demonstrated on the Single-Chip Cloud Computer (SCC), a many-core research chip created by Intel Labs. Experiments show that collective operations subjected to the identified optimizations are accelerated by factors roughly between 2 to 3 compared to current state of the art implementations. In addition to synthetic benchmarks, we show that the use of the optimized routines accelerates a scientific application by more than 40%.**

***Keywords*-Many-core processors; MPI; Collective operations**

I. Introduction

In the last few years, the most notable advance in microprocessor technology has been the increase in processor cores per chip. While back in 2004 single-core CPUs dominated the desktop market, current workstation and server processors already feature up to 16 cores. This trend of putting more cores onto a single chip is expected to continue, as it helps addressing problems like increasing power consumption or the limited effectiveness of further parallelization efforts on the instruction level. It is projected that future microprocessors could consist of hundreds or even a thousand cores [1]. In current processor designs, cores communicate via shared memory, but since memory bandwidth does not scale as fast as the number of cores, processor vendors are looking for alternatives. One possible solution is to provide cores with their own private memory and let them communicate over fast on-chip networks by exchanging messages. Prominent examples for such architectures include the *Tilera Tile-Gx* [2] and the *Single-Chip Cloud Computer* (SCC) of Intel [3]. These processors resemble distributed memory systems, and as such often use message-passing libraries similar to MPI [4] for communication.

When moving from macroscopic networks to on-chip communication, existing message-passing solutions may require rethinking in multiple ways to achieve high application performance. For example, novel algorithms might be needed to exploit special features provided by the hardware. Second, the consideration of topological information might be used to increase resource utilization. Finally, the different characteristics might favor a certain implementation over alternatives. The low latency of on-chip networks for example allows finer-grained parallelization and enables the scaling of problems to higher core counts compared to macroscopic networks. While the higher latency of off-chip communication masks the overhead of features and convenience functions built into modern message-passing libraries, the latency incurred by this overhead can turn into a limiting factor of communication performance when using on-chip networks. This is of special importance for functions acting as primitives for more complex functions. In MPI, this refers to the point-to-point communication methods on which the collective operations are based.

We investigate some of the formerly stated issues by analyzing the runtime behavior of the collective operations of the Intel SCC's native communication stack. With the example of the *Allreduce* operation, we identify parts inside the stack that exhibit overhead-induced delays and which may benefit from optimizations. With this example, we have developed guidelines for optimizations in order to achieve higher application performance. Measurements show that addressing the identified issues accelerates individual collective operations typically by factors of 2 to 3, and decreases the runtime of a scientific application computing thermodynamic properties by more than 40%.

The remaining paper is organized as follows: Section II gives an overview of the SCC's architecture and highlights the specific parts relevant to this work. Section III briefly evaluates related work. In Section IV, we develop the proposed optimizations in a step-by-step manner. Results are presented in Section V, followed by a conclusion.

II. Architecture of the SCC

The Intel SCC is a homogeneous, non-cache coherent many-core processor consisting of 48 x86 Pentium cores [5]. The cores are paired to 24 so-called tiles and connected to each other via a fast on-chip mesh network. The chip holds four on-chip dual-channel capable DDR3 memory controllers that provide access to a maximum of 64 GB of main memory through this mesh network (our setup is equipped with 32 GB).

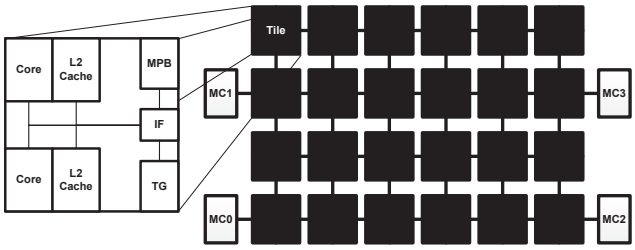


Fig. 1: Simplified hardware architecture of the SCC, displaying the arrangement of the tiles and their content

Fig. 1 shows the general architecture of the chip and one of its tiles. Despite the fact that all cores are on the same die, the SCC can be regarded as a distributed memory machine in its standard configuration. The entire main memory is distributed evenly among the cores and each core is running its own small operating system. The experimental system used for the work in this paper is a standard PC running Ubuntu Linux 10.04. The SCC board is connected via PCI Express, providing easy access to the chip for productive as well as diagnostic or debugging purposes.

A tile of the SCC is composed of two 32 bit Pentium P54C cores, equipped with 16 KB of L1 instruction and data cache as well as 256 KB of L2 cache for each core. The L2 cache policies are pseudo least-recently-used, write-back and non-write-allocate. The architecture of the chip is non-cache coherent, therefore coherency must be ensured in software if required. Furthermore, the tiles also provide interface units that connect them to the mesh network. Since the system allows for a maximum of 64 GB of main memory, lookup tables translate the 32 bit memory addresses of the cores to 36 bit memory addresses of the system.

While shared memory communication among the cores is possible, the main data exchange paradigm of the SCC is message passing communication. To facilitate high-performance communication, the SCC also holds 384 KB of fast on-chip SRAM – 8 KB per core (i.e. 16 KB per tile) – the main purpose of which is to act as message passing buffers (MPBs). Although physically distributed and normally only accessed by communication libraries, the MPBs can be regarded as a single shared memory block that can be equally accessed by all cores using simple memory read/write operations.

III. Related work

The SCC's native communication API is called *RCCE* (pronounced "rocky") [6] and is intended as a lightweight library for writing message-passing-based applications. It features an MPI-like set of functions for both point-to-point and basic collective operations. For point-to-point communication, blocking *send* and *receive* primitives are provided. In addition, RCCE supports the collective operations *Broadcast* and *(All-)Reduce* with very basic implementations. However, due to their simplicity, they do not scale well since the root communicates with the remaining cores in a serial way. In addition, for *Reduce*, the computation of the actual reduction is performed solely by the root. As a consequence, these algorithms do not use the available parallelism and suffer from both high latency and low efficiency.

There have been several approaches to address these limitations. The *iRCCE* library [7] extends RCCE with non-blocking point-to-point primitives and introduces optimized `memcpy` routines that significantly increase the message-passing performance. These optimized routines have also been integrated into later RCCE releases. To improve the efficiency of the collectives, [8] and [9] present tree-based alternatives for *Broadcast* and *(All-)Reduce* that outperform the simple RCCE variants by factors of more than 20x for *Broadcast* and 6x for *Reduce*. *RCKMPI* [10] is a full-featured MPI implementation based on MPICH that uses the SCC's MPBs as an internal communication channel. It implements the complete MPI specification and contains sophisticated algorithms for collective operations. These provide a set of routines for different message sizes and pick the one that performs best at runtime. For short messages sizes, RCKMPI's performance is similar to the previously mentioned isolated tree-based approaches. In case of long message sizes, the specialized routines of RCKMPI even show a performance advantage. Its main drawbacks are the significantly higher memory footprint and runtime overhead compared to RCCE.

Finally, the *RCCE_comm* library [11] provides a solution between a full, function-rich MPI implementation, and the simple, lightweight RCCE library. Second only to RCKMPI, it contains the most complete suite of collective operations currently available for the SCC, including variants for different message sizes. Since it is built on standard RCCE primitives, it extends RCCE by advanced and efficient collective operations without introducing too much overhead in terms of memory and runtime. Initial tests showed that the thermodynamics application scales best and achieves highest performance when linked against the RCCE_comm library. Hence, we have used this library as basis for our optimization efforts.

Our main contribution is the identification of points where common algorithms used for collective operations can be optimized to take full advantage of the characteristics of on-chip networks, here developed for the example of *Allreduce*. These optimizations are then applied to additional collective operations where applicable.

IV. Optimizations

We develop the candidate points for optimization at the example of *Allreduce*, as all presented points can be applied to this primitive. In addition, we describe for each point the collectives to which this optimization can also be applied.

The semantics of *Allreduce* can be described as follows: Let $\left\{v^0, v^1, \ldots, v^{p-1}\right\}$ be a set of input vectors distributed over p cores, vector v^j being located at core j. Every input vector contains n elements, with v_i^j denoting the i-th element of core j's input vector. After the call to *Allreduce*, each core is returned the same result vector w, where w has the same element count as the input vectors and all elements w_i satisfy $w_i = \sum_{j=0}^{p-1} v_i^j$. To put it in a less formal way, an element of the result vector contains the sum of all elements with the

same index from all cores' input vectors. Note that the addition operator applied by the sum is just an example here and can in general be replaced by any associative binary operator.

A. Synchronization

Ideally, cores should be able to work independently in parallel. However, e.g. when communicating, cores will eventually need to synchronize with each other to preserve causality. Since this may require a core to wait on other cores and keep it from computing, synchronization should be used sparsely and only where necessary. This can be an issue, as profiling of the thermodynamic application shows: Here, cores spend up to 50% of their time in the `rcce_wait_until` method, which is used to wait on a flag of a remote core and mainly serves the purpose of synchronizing the sending and receiving cores of a message-passing call. One way to speed up communication is hence to reduce the need for synchronization.

For longer vector sizes, the *Allreduce* of RCCE_comm is implemented as a *ReduceScatter* followed by an *Allgather*. The *ReduceScatter* is particularly susceptible to the synchronization issue, as it is based on the *bucket* or *ring algorithm* [12]. Here, cores iteratively "push" portions of their operand vector along a virtual ring containing all cores (cf. Fig. 2).

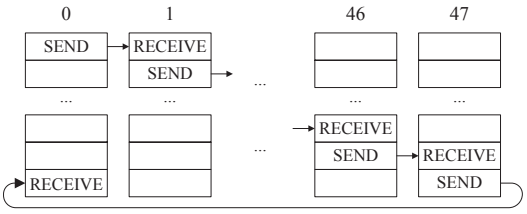


Fig. 2: Ring algorithm for *ReduceScatter*

For the actual data transfer, RCCE's blocking point-to-point primitives are used. These synchronize twice, i.e. the receiver waits for the sender to provide data, and the sender waits until the receiver has picked up the data (cf. Fig. 3). Thus a *send* call cannot return until the matching *receive* is called and vice versa, making the cyclic communication pattern of the ring algorithm prone to deadlocks. RCCE_comm avoids this issue by ordering the *send* and *receive* calls in the *odd-even* pattern: Cores with an odd ID do the *receive* first, followed by *send*, even-numbered cores use the reverse order, as shown in Fig. 4.

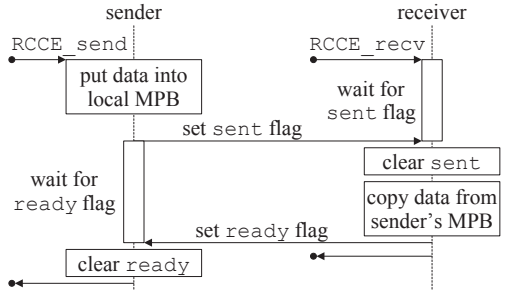


Fig. 3: *RCCE* internals of the *send* and *receive* primitives

This ordering results in excessive synchronization, as nodes are synchronized on both their first and second operations: A node can start its second operation only after all nodes completed the first one, similar to a barrier between both operations. This is illustrated by the bold, dashed lines in Fig. 4. Since the exchanged data blocks have no data dependencies between each other, such barriers are not necessary from the algorithmic perspective. The strict synchronization can be relaxed by the use of non-blocking primitives, as illustrated in Fig. 5. This has two advantages over the "blocking" solution: First, it simplifies the implementation of the ring algorithm, as issuing the *send* and *receive* requests of one round in an arbitrary order does no longer cause deadlocks and renders the odd-even scheme obsolete. Second, since there is no longer an explicit ordering between the first and second operation, cores can concurrently copy data in and out of the MPBs, effectively using the time they formerly spent waiting for doing actual work. Synchronization is now required only once each round by waiting for the *send* and *receive* requests to finish, as indicated by the bold, dashed line in Fig. 5.

As a first optimization step, we built the RCCE_comm library on top of iRCCE and replaced the blocking communication calls by their non-blocking alternatives. Since this kind of synchronization occurs whenever a pair of cores has to exchange a message, this optimization has also been applied to the *Allgather* and *Alltoall* operations. For *Allreduce*, a moderate speedup of ∼25% could be achieved for a single call by this replacement (see Section V for performance figures).

B. Minimizing software overhead

An analysis of the iRCCE primitives showed that they have a significantly lower efficiency, i.e. they have a higher execution time than their blocking counterparts, when the time spent transfering data or waiting for flags is excluded. This is due to the higher management complexity needed for advanced features: iRCCE supports multiple concurrent *isend* and *irecv* requests, the cancellation of pending requests, and the reception of messages from arbitrary cores with arbitrary sizes[1]. Pending requests are kept inside a linked list and hence require dynamic memory operations when issued and after completion.

While these features make message-passing-based programming very comfortable from a user perspective, the additional overhead is counterproductive when the provided functions are used as primitives in communication libraries on higher levels, as e.g. in RCCE_comm. Since most algorithms for collective operations, including the ring algorithm, are organized into rounds where a core exchanges at most one message with another core, the expensive listkeeping can be avoided by allowing only one active send and receive operation at a time. We used this fact to extend RCCE by lightweight non-blocking primitives that support at most one concurrent *send* and *receive*.

[1]Plain RCCE, in contrast, requires the ID of the sender core and the message length to be known "in advance", i.e. they must be passed as parameters to the *receive* call.

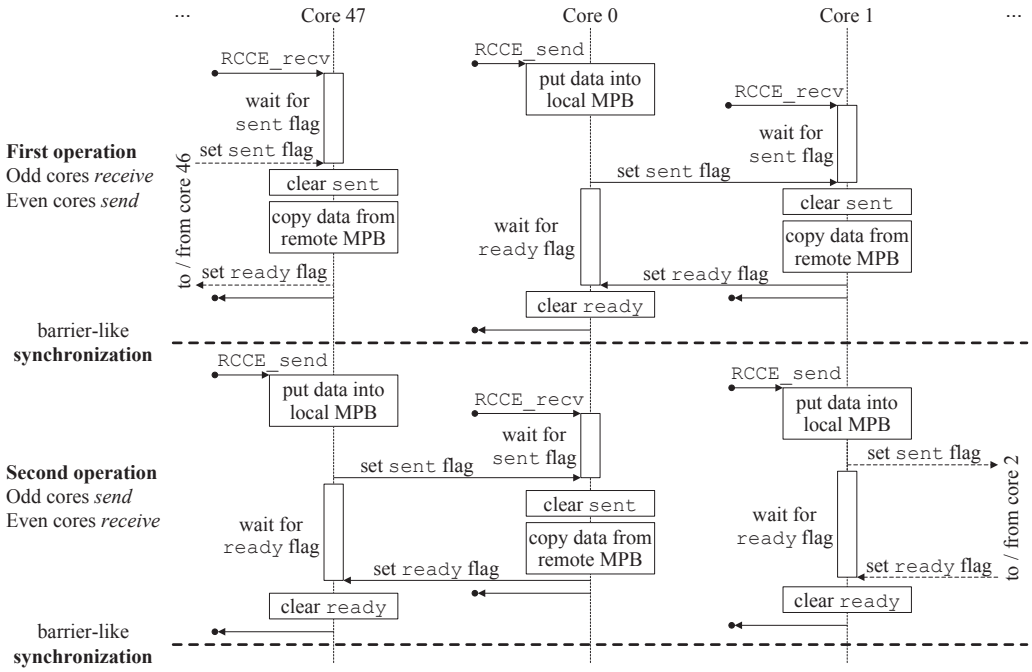


Fig. 4: Cyclic *send* with *odd-even* ordering to avoid deadlocks

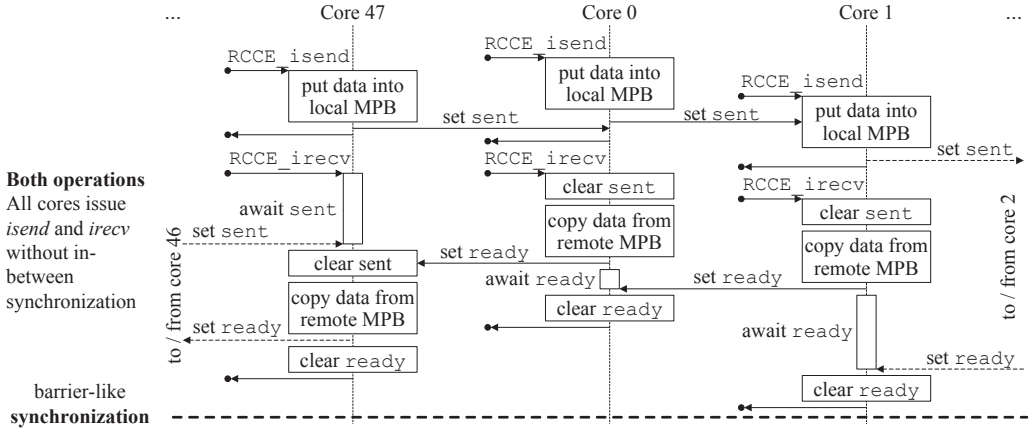


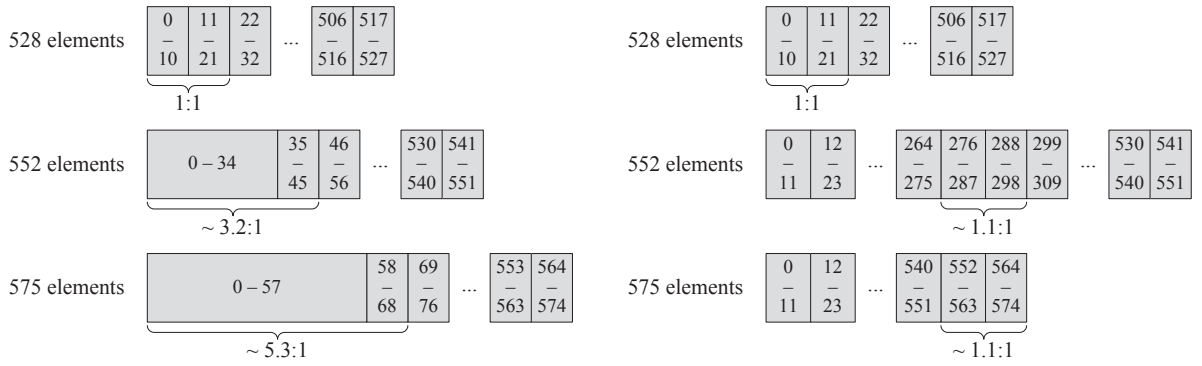
Fig. 5: Cyclic *send* with non-blocking primitives

Again, the communication primitives of the collectives stated in the previous section were replaced with these lightweight functions. First evaluations showed that software overhead is really an issue for low-latency networks, as the lightweight functions further accelerate the *Allreduce* operation by ∼65% compared to the iRCCE-based variant of the previous section.

C. Load balancing

Parallel programs achieve the best performance when the workload is balanced among the cores, so they can work in parallel and do not have to spend time waiting for other cores to finish their workshare. For the ring algorithm of *ReduceScatter*, this means that a round should take about the same time to complete on every core, as cores are synchronized with each other during a round. To have a well-balanced load, cores should communicate and process roughly the same amount of vector elements per round, assuming that they all run at the same frequency and have equal processing power.

As shown in Fig. 2, the ring algorithm splits the operand vectors into a number of blocks that match the number of participating cores and form the basic unit for communication and compution. Each core processes a different block, so all blocks are being processed in parallel. Hence, the load is perfectly balanced when all blocks are of the same size. Unfortunately, it is not always possible to attain perfect balance, since the vector size may not be a multiple of the number of cores. Instrumentation showed that this also happens in the thermodynamic application. In its long-range energy evaluation function, vectors of 276 complex-valued



(a) Standard block sizes (RCCE_comm)

(b) Optimized block sizes

Fig. 6: Block sizes and their ratio for different vector lengths

Fourier coefficients have to be summed over all cores in order to compute the energy term. This is done by an *Allreduce* over the local coefficient vectors, which are treated as vectors of size 552 (a real and an imaginary part per element) for the reduction. Since 552 elements cannot be divided evenly by 48 cores (11.5 elements per core), RCCE_comm defines the general block size as the integer part of the division (i.e. 11), and increases the size of the first block to include the remaining elements.

As shown in Fig. 6a, the remaining elements can be larger than the general block size, thus making the first block significantly larger than the other blocks. While in the best case (Fig. 6a, top) all elements are distributed evenly, the first block can grow to more than five times the size of the remaining blocks in the worst case (Fig. 6a, bottom). For our particular case (Fig. 6a, middle), the first block is more than three times the size of a general block. This slows each round down significantly, as all cores processing blocks with the general block size are idle two thirds of the time, waiting for the core processing the first block to finish.

To improve this issue, we changed the method of splitting the input vector into blocks such that a more equal distribution is achieved. Having n elements per input vector and p cores, we add one additional element to the first $n \mod p$ blocks, as shown in Fig. 6b. In addition to *Allreduce*, this optimization has also been applied to the *Reduce* operation to a single root and to the *Broadcast* implementation, which uses a scatter-gather approach for longer messages. This change in the block definition process reduces the imbalance between larger and smaller blocks from a worst case factor of 5 down to a factor of 1.1. For the example of an *Allreduce* of 552 elements, this achieves an additional speedup of $\sim$28% compared to the version of the previous section.

D. Hardware-specific optimization

As a final step, the hardware at hand can be considered to allow further optimization. In case of the SCC, this puts the focus on the MPBs. The high-level flavor of RCCE (the so-called *non-gory* interface) uses the MPBs exclusively for message-passing and synchronization via flags. Since MPBs

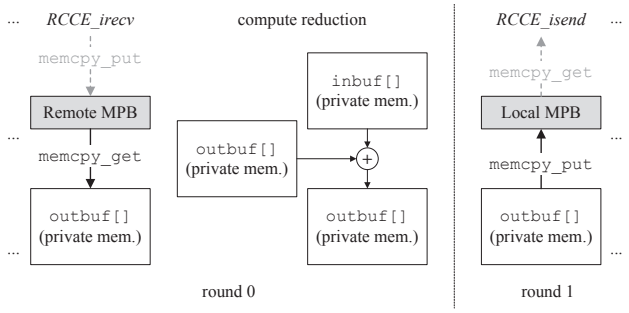


Fig. 7: Low-level actions for *Allreduce* based on *isend/irecv* primitives

can be accessed directly by all cores in the system, application performance can be increased by lifting this restriction. Especially "in-transit" data, which are received and sent with little or no local processing, or data that must be sent frequently to other cores, can benefit from placement inside an MPB.

For *ReduceScatter*, the data blocks of the split input vectors meet these criteria. Starting with the reception of a block via *RCCE_irecv* (cf. Fig. 7, left), a core first copies the data from the MPB of its left neighbor into its own private memory. Each received element is reduced with the corresponding element of the local input vector, the result again being written into private memory (cf. Fig. 7, center). At the start of the next round, this result must be sent to the core's right neighbor in the ring, requiring another copy operation into the local MPB. From here, the right neighbor will read the data and copy it into its private memory, closing the cycle (cf. Fig. 7, right). A data block is thus received, processed and immediately sent out again, fitting the idea of in-transit data.

Consequently, we adapted the ring algorithm to work directly on MPBs whenever possible. The structure is shown in Fig. 8: Instead of copying the intermediate result from the left neighbor's MPB, we feed the reduction operator directly with this MPB's address as the pointer to the first operand, use the local input vector as the second operand, and write the result vector into the local MPB. Then, in the next round, the right neighbor only needs to use this MPB's address as the pointer

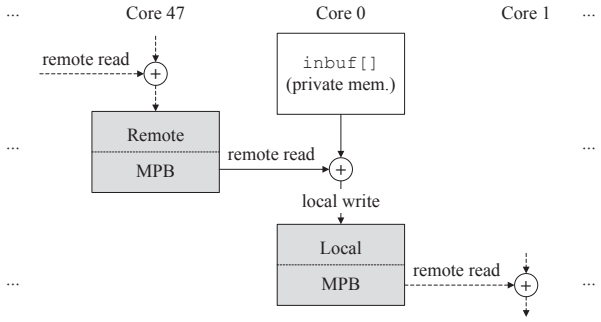


Fig. 8: *Allreduce* working directly on the MPBs

to the first operand, hence completely omitting the need to copy to and from private memory. Since all rounds process one data block per round in parallel, double buffering is used to enable concurrent processing. MPBs are split in half into two buffers, as hinted by the dotted lines in Fig. 8, allowing a core to fill one buffer with the results being computed this round while its right neighbor reads the intermediate results of the previous round from the other buffer. The roles of the buffers are swapped at the beginning of a new round, the same handshaking mechanism via *sent* and *ready* signals used in the non-blocking variant (cf. Section IV-A) ensuring synchronization.

Since MPBs are on-chip SRAM memory, their access time should be comparable to that of on-chip caches, i.e. significantly lower than the access time for off-chip DRAM memory. We expected the MPB algorithm therefore to have a clear performance advantage, as it avoids the frequent copying into and out of MPBs and contains fewer off-chip accesses incurring high latency. However, compared to the version described in the previous section, the MPB-aware implementation achieves only a speedup of 10%. The reason for this rather poor performance is a bug in the SCC's hardware that is triggered when both cores of a tile access the local MPB at the same time. Since the accesses are local, no network packet must be created. When both cores try to access the local MPB simultaneously, the arbiter fails to suppress the packet creation for the request losing the arbitration round[2]. This may cause data corruption or even lead to lockup for the involved core. As a workaround, direct access to the local MPB is prohibited; instead, cores have to send packets to themselves containing the read or write request. Consequently, accesses to the local MPB are slowed down from a latency of 15 core cycles to 45 core cycles plus 8 mesh cycles [13], coming close to the transmission latency required for off-chip memory accesses (40 core cycles + $8d$ mesh cycles, where d is the number of hops between core and memory controller) [5]. As private memory is cacheable, only the first access to a private memory address is actually going off-chip. The latency for this first access is similar to that of an MPB access. All later accesses of the MPB-agnostic algorithm to the same address now refer to the copy inside the cache, effectively masking the off-chip access time. As a consequence, only a very small performance benefit can be gained by working directly on MPBs. However, with the hardware bug resolved, we expect to see significantly higher speedups.

[2]The bug is e.g. mentioned in [13], page 8. A detailed description can be found at http://communities.intel.com/docs/DOC-5405.

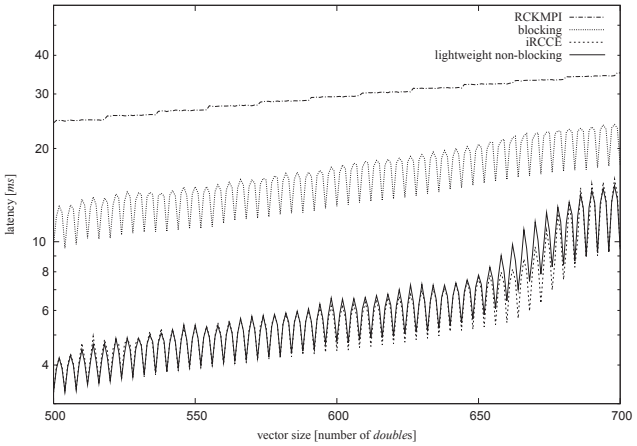
V. Experimental results

We have evaluated the effects of the different proposed optimization steps on the affected collective operations and on an application from the domain of thermodynamics. The SCC was clocked according to the standard preset, meaning the cores run at 533 MHz, whereas the network and the DRAM both run at 800 MHz. For the communication libraries, we used the latest available versions of RCCE (v1.1.0), RCCE_comm and RCKMPI (both at rev. 303 of the public SVN repository).

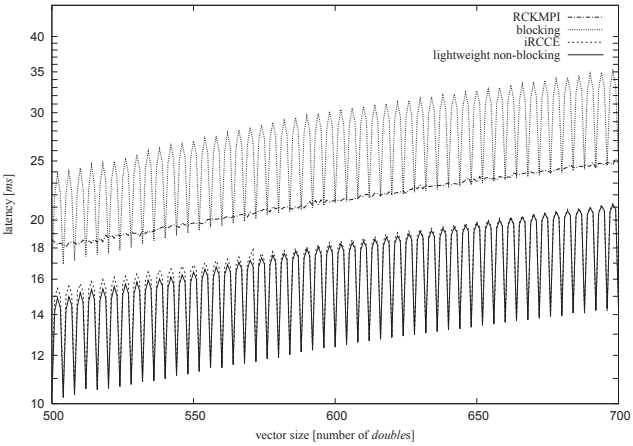
A. Performance of single collective operations

Fig. 9 shows the average measured latency against the vector size for executing a single collective operation on all 48 cores and having different sets of optimizations applied. To produce accurate results, operations were repeated 10000 times for each vector size, and the measured latencies were averaged. The displayed latencies were measured on core 0, but since the statistical variance over all cores is very low, the values are also representative for the other cores. The graph named *blocking* shows the latency for the SCC's native communication stack made of RCCE_comm on top of RCCE without any optimizations. Since we based our optimizations on this implementation, we use it as a reference and specify all speedups relative to its runtimes. Graph *iRCCE* displays the latency after relaxing the synchronization by using iRCCE's non-blocking primitives (cf. Section IV-A). The *lightweight non-blocking* graph shows the latency after replacing iRCCE's complex communication routines with lightweight ones that incur less software overhead (cf. Section IV-B). The *lightweight non-blocking, balanced* graph represents the latency after additionally applying the load balancing optimization (cf. Section IV-C). The *MPB-based Allreduce* graph shows the latency for a communication stack that has all three former optimizations applied (relaxed synchronization, lightweightness, balancedness) and additionally features an *Allreduce* routine optimized to the SCC architecture as described in Section IV-D. Finally, we have provided the latency graph for *RCKMPI* in order to have a comparison against a standard MPI implementation.

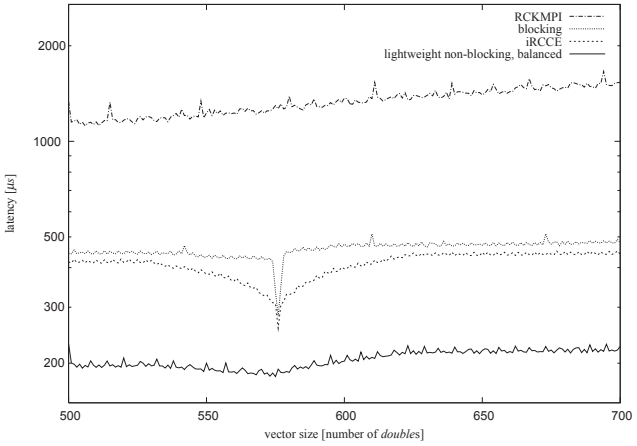
Apart from RCKMPI, virtually all graphs show a "spiky" behavior with a period of 4 elements. As we use *doubles* with a size of 8 bytes each as element type, this matches the size of a core's L1 cache line. Message transfer is implemented inside RCCE by writing cache lines of the local core's L1 into a remote MPB. Since the SCC's cores feature a write combining buffer, only complete cache lines can be transferred across the network, i.e. messages that do not fill a complete line have to be padded and require the transfer of an additional line. This additional transfer is realized by an extra call to the communication function inside RCCE, thereby adding



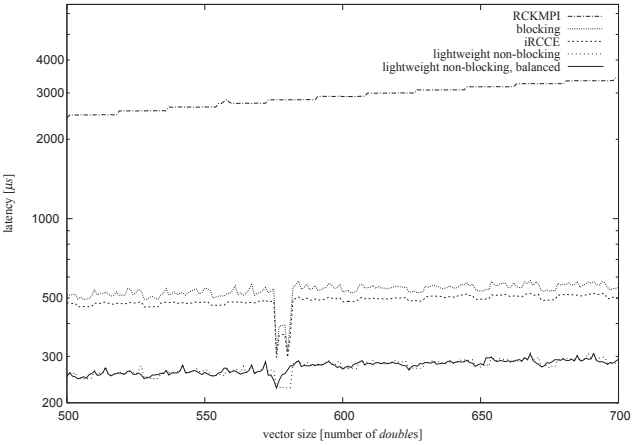
(a) Allgather



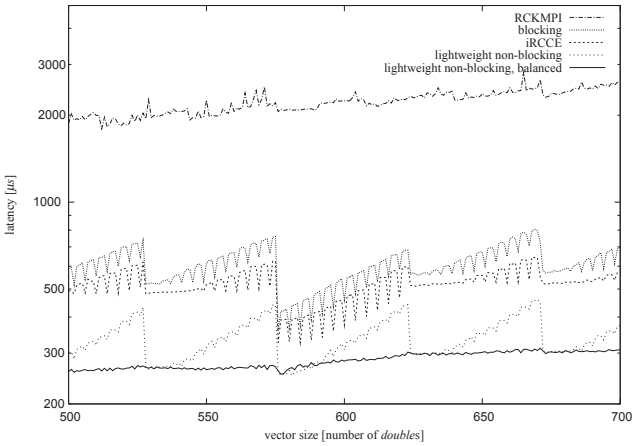
(b) Alltoall



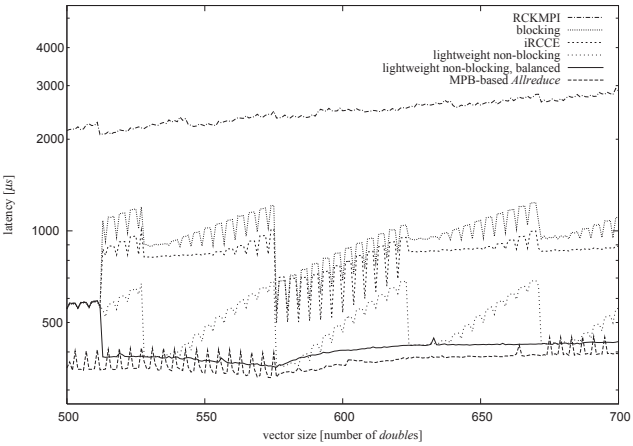
(c) ReduceScatter



(d) Broadcast



(e) Reduce



(f) Allreduce

Fig. 9: Latencies of the optimized collectives

overhead. Thus, vector sizes divisible by 4 (e.g. 600 elements) form the best cases and can be seen at the lower ends of the spikes. In contrast, RCKMPI uses a different transfer scheme that does not involve the extra function call for message sizes not being a multiple of an L1 cache line. As such, its runtime scales much smoother with the vector size. However, since RCKMPI performs significantly worse (about factors 2 to 5) than our baseline of RCCE and RCCE_comm in all cases except *Alltoall*, we will not refer to it in the following discussion.

For the *Allgather* and *Alltoall* operations (Figs. 9a and 9b), the relaxed synchronization results in average speedups of 2.7x and 1.6x, respectively. Interestingly, the choice of non-blocking primitives implementation has little or no effect on performance here. This can be explained by the fact that these operations transfer messages containing all elements of the provided data vectors. Hence, the management overhead of a message transfer is negligible compared to the time actually spent doing communication. For *ReduceScatter* and *Broadcast* (Figs. 9c and 9d), which subdivide the element vector and transmit smaller messages, the lightweight routines show a performance improvement of 1.1x and 1.8x, respectively. At 576 elements, these two routines exhibit a sharp drop in latency for the *blocking* and *iRCCE* libraries. While this behavior can also be observed for the lightweight routines, its impact is much less severe here. Up to this point, we were not able to find a definite cause for this phenomenon, but since it occurs again on a 4-element boundary, we suspect it to be a cache-related effect.

As *Reduce* and *Allreduce* (Figs. 9e and 9f) also subdivide their element vector, these two routines are also accelerated by 1.6x and 1.7x on average when using lightweight non-blocking primitives. In addition, these two operations show the effect of load balancing very clearly, as the latency of non-balanced versions (*blocking*, *iRCCE* and *lightweight non-blocking*) is lowest for vector sizes that are a multiple of 48[‡] and increases linearly for additional elements until the next multiple of 48 is reached. In contrast, performance stays qualitatively on the same level when using the balanced versions. As discussed in Section IV-D, the difference between the lightweight and the MPB-based *Allreduce* implementations is not very significant, but we expect a higher impact of the MPB optimization without the previously mentioned hardware bug.

In summary, all collectives show speedups between approximately 1.6x (*Alltoall*) and 2.8x (*Allgather*) on average. Together with proper load balancing, a maximum of 3.6x is achieved for *Allreduce* at a vector size of 574 elements.

B. Application performance

To show the effect on application performance, we measured the runtime of a thermodynamics application, linked against communication libraries containing the various optimization steps. The application employs statistical mechanics, namely the *Grand canonical Monte Carlo* (GCMC) technique [14], to sample thermodynamic properties like the internal energy or pressure of a gas or fluid under given conditions (e.g. temperature, volume, chemical potential).

[‡]As there are 48 cores, the vector is split into 48 blocks, cf. Section IV-C.

Input: NUM_CYCLES

```
1  en_old = InitialEnergy();
2  for cycle := 1 to NUM_CYCLES do
3      action := PickRandomAction();
4      particle := PickRandomParticle();
5      en_new = en_old − ShortEn(particle) − LongEn();
6      SaveCurrentConfig(particle);
7      DoGCMCMove(action, particle);
8      en_new = en_new + ShortEn(particle) + LongEn();
9      if random() < min(1, e^{−β(en_new − en_old)}) then
10         en_old = en_new ; // accept GCMC move
11     else
12         RestoreConfig(particle) ; // reject move
13     BroadcastUpdate(particle, en_new);
```

Algorithm 1: GCMC main loop

Algorithm 1 shows the simplified structure of the GCMC code. In the main loop, a set of *particles*, i.e. molecules consisting of multiple atoms, is changed at random by moving, rotating, inserting or deleting a particle. Such a modification is called a *GCMC move*. Depending on the change in the total energy, a move is either accepted (line 10) or rejected (line 12). Since evaluating the total energy is computationally the most demanding part of the loop, particles are distributed over the SCC's cores so each core can compute the contribution of its local set of particles in parallel. The total energy is made up of the energy contributions of both *short range* and *long range interaction* between atom pairs. Short range energies are computed in real space, allowing an incremental update of the total energy by subtracting the contribution of the modified particle before the move and adding its new contribution after the move. The long range part, shown in Algorithm 2, is computed in Fourier space and hence cannot be subjected to an incremental update. Instead, a full recalculation considering all atom pairs is required after a move.

This makes long range energy computation the most time consuming part of the application. Profiling results show that up to 60% of the total runtime is spent inside this function. The high complexity is not limited to computation, but also affects communication. While it is sufficient in the short range energy case to sum up the energy contributions of the local particle sets, i.e. one value per core, the summation for the long range energies needs to be done in Fourier space, requiring the transfer of a set of complex-valued Fourier coefficients. The summation is done by a call to *Allreduce* (cf. Algorithm 2, line 14), which hence makes up for a significant part of the application runtime. Consequently, we expected the optimizations to have a notable impact on application performance.

As shown in Fig. 10, these expectations were met as the combined optimizations improve the runtime of the application

Input: $LocalParticles, num\_atoms, atom[\,][\,], vol, KMAX, KMAXVECS$
Output: $energy$

$energy := 0.0$;
$\forall k, p, a: \; F[k][p][a] := 1.0 + 0.0i$;
$\forall k: \; F_{tot}^{local}[k] = 0.0 + 0.0i$;
for $p \in LocalParticles$ **do**
  for $a := 0$ **to** num_atoms **do**
    for $k := 1$ **to** $KMAX$ **do**
      $F[k][p][a] := \cos(2\pi k/vol \cdot atom[p][a])$
        $+i \cdot \sin(2\pi k/vol \cdot atom[p][a])$;
      $F[-k][p][a] := \overline{F[k][p][a]}$;
for $p \in LocalParticles$ **do**
  for $a := 0$ **to** num_atoms **do**
    for $k := 0$ **to** $KMAXVECS - 1$ **do**
      $F_{tot}^{local}[k] :=$
      $F_{tot}^{local}[k] + F[kvec(k)][p][a] \cdot atom\_charge(a)$;
ALLREDUCE $(F_{tot}^{local}, F_{tot}, SUM)$;
for $k := 0$ **to** $KMAXVECS - 1$ **do**
  $energy := energy + coeff(k)/vol \cdot |F_{tot}[k]|^2$;
return $energy$;

Algorithm 2: *LongEn*: compute long range energy

by more than 40% compared to the baseline of standard RCCE and RCCE_comm (*blocking* in Fig. 10). Note that the runtime of RCKMPI is not displayed at scale, as it exceeds the runtime of the baseline by more than a factor of two. In combination with the significant reduction in runtime that goes along with the replacement of iRCCE's complex non-blocking primitives by lightweight routines ($> 17\%$ improvement from *iRCCE* to *lightweight non-blocking*), this demonstrates the importance of lightweight, low-latency collectives for achieving good application performance on architectures with on-chip networks.

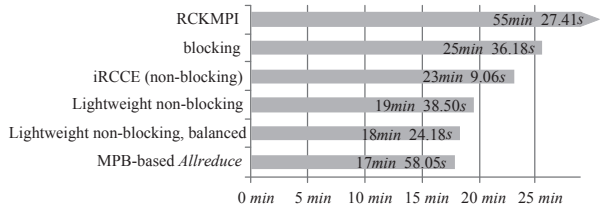


Fig. 10: Application performance

VI. Conclusion

We have investigated several points for improving the efficiency of collective communication operations. The results show that lightweight primitives with low overhead are a significant performance factor for systems with low latency communication and can improve application performance by more than 40%. As low latencies decrease the cost for communication in terms of time, systems with on-chip networks allow finer-grained parallelized algorithms involving a higher communication ratio compared to systems with macroscopic networks. Thus, we believe highly optimized communication operations will become increasingly important for achieving high application performance on many-core processors in future.

Acknowledgment

The authors A. K. and M. R. would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/1) at the University of Stuttgart. This research has also been partially funded by the Austrian Research Promotion Agency under contract No. 834307 (AutoCore) and by the FWF Austrian Science Fund as part of the project TRP 220-N23 "Automatic Portable Performance for Heterogeneous Multi-cores".

References

[1] S. Borkar, "Thousand core chips — a technology perspective," in *Proc. Design Automation Conference (DAC)*, San Diego, CA, USA, June 2007, pp. 746–749.
[2] "TILE-Gx processor family," (2012, Jan. 20). [Online]. Available: http://www.tilera.com/products/TILE-Gx.php
[3] J. Howard *et al.*, "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, January 2011.
[4] "The Message Passing Interface (MPI) standard," (2012, January 20). [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi
[5] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance analysis and benchmarking of the Intel SCC," in *Proc. Conference on Cluster Computing (CLUSTER)*, Austin, TX, USA, September 2011, pp. 139–149.
[6] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on Intel's Single-Chip Cloud Computer processor," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 73–83, January 2011.
[7] C. Clauss, S. Lankes, T. Bemmerl, J. Galowicz, and S. Pickartz, "iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer," RWTH Aachen University, Tech. Rep., November 2011, (2012, January 20). [Online]. Available: http://communities.intel.com/docs/DOC-6003
[8] A. Chandramowlishwaran, R. Vuduc, and K. Madduri, "Performance evaluation of the 48-core Single Chip Cloud Computer," in *Workshop on Manycore and Accelerator-based High-performance Scientific Computing*, Berkeley, CA, USA, January 2011.
[9] H. Al-Khalissi and M. Berekovic, "Performance of RCCE broadcast algorithm in SCC," in *Proc. 3rd Many core Applications Research Community (MARC) Symposium*, Ettlingen, Germany, July 2011, pp. 93–98.
[10] I. A. Comprés Ureña, M. Riepen, and M. Konow, "RCKMPI - lightweight MPI implementation for Intel's Single-chip Cloud Computer (SCC)," in *Proc. European MPI Users' Group Conference on Recent Advances in the Message Passing Interface (EuroMPI)*, Santorini, Greece, September 2011, pp. 208–217.
[11] E. Chan, "RCCE_comm: A collective communication library for the Intel Single-chip Cloud Computer," 2010, (2012, January 20). [Online]. Available: http://communities.intel.com/docs/DOC-5663
[12] M. Barnett, R. Littlefield, D. G. Payne, and R. van de Geijn, "Efficient communication primitives on mesh architectures with hardware routing," in *Proc. Conference on Parallel Processing for Scientific Computing (PPSC)*, Norfolk, VA, USA, March 1993, pp. 943–948.
[13] "The SCC programmer's guide," 2011, (2012, January 20). [Online]. Available: http://communities.intel.com/docs/DOC-5684
[14] D. Adams, "Grand canonical ensemble Monte Carlo for a Lennard-Jones fluid," *Molecular Physics*, vol. 29, no. 1, pp. 307–311, 1975.