Latency-optimized Collectives for High Performance on Intel's Single-chip Cloud Computer

Adán Kohler and Martin Radetzki
University of Stuttgart, Department of Embedded Systems
Pfaffenwaldring 5b, 70569 Stuttgart, Germany
{kohleran, radetzki}@informatik.uni-stuttgart.de

***Abstract*—The Single-Chip Cloud Computer (SCC) is a research chip of Intel Labs featuring 48 processor cores on a single die. Having no hardware support for cache coherence, the SCC resembles a distributed on-chip system that uses message passing over an on-chip network of switches for inter-core communication. The low latency of on-chip message passing allows algorithms to scale beyond the limits of macroscopic computer clusters. In addition to point-to-point communication, parallel message passing applications often use collective operations that involve all or a set of the available processes. Consequently, collectives should be tuned for low latency, too, in order to achieve fair application performance. To support this, we present optimized versions of collectives that outperform the fastest implementations currently available for the SCC by factors of 1.6x up to 2.5x. For a thermodynamics application, the use of these optimized routines results in a speedup of 42%.**

***Index Terms*—Message passing, MPI, collective communication**

I. Introduction

The Single-Chip Cloud Computer (SCC) is a research chip developed by Intel Labs that features 48 x86-compatible processor cores on a single die. The chip is a tile-based design, where 24 homogeneous tiles, each containing two processor cores, are arranged into a 6×4 mesh. The tiles are connected with each other and off-chip resources over a fast on-chip network. To this end, a tile includes a mesh interface unit and 16 KB of local memory that is primarily intended for message-passing. Since the SCC has no hardware support for cache coherence, it can be seen as a distributed on-chip system. This notion is also supported by the native communication library RCCE [1], which provides MPI-like functions for passing point-to-point messages in a blocking manner. In contrast to a computer cluster, passing a message through an on-chip network involves very low latency, typically in the order of 50 clock cycles [2]. As communication costs decrease in relation to computation, applications can be parallelized at a finer scale and among a higher amount of cores to make use of the chip's available transistors. In addition to point-to-point communication, such applications generally use collective communications like *Broadcast* to distribute initial data to cores, or *Reduce* for collecting intermediate or final results. Consequently, having both low-latency point-to-point and collective operations is important for reaching high application performance.

With the example of an application computing thermodynamic properties, we have developed a set of optimizations for the existing message-passing libraries and evaluted their effect on the latency of individual collective calls, as well as on the performance of the complete application.

II. Previous work

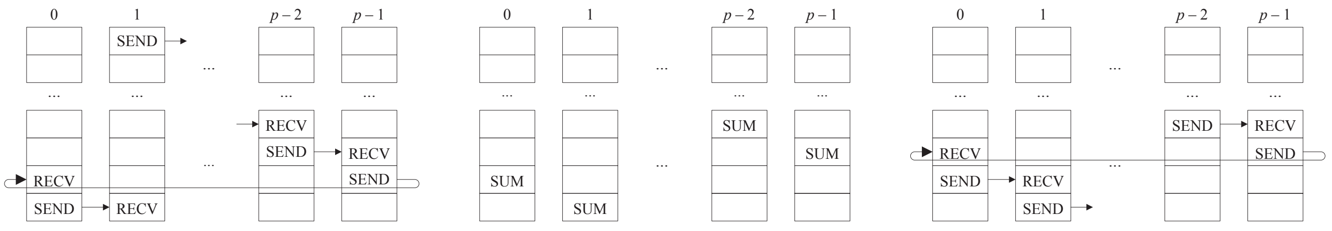
The first approach for porting the thermodynamics application to the SCC was based on RCKMPI [3], an MPI implementation customized for the SCC. This MPI stack is derived of MPICH and uses the SCC's on-chip network as communication channel. While cross-compiling the application and linking it against RCKMPI worked without modififcations to the source code, the RCKMPI version available at that time was known to have some scalability issues. Thus, in a second step, we replaced it by the small and lightweight library RCCE provided by Intel. As RCCE is missing efficient implementations for collective operations, we complemented it with RCCE_comm [4], a library implementing collectives with RCCE primitives. Compared to RCKMPI, this combination roughly doubled the achieved application performance.

At a later stage, RCKMPI was superseded by a new version called RCKMPI2 [5] that fixes the scalability issue and increases the communication efficiency significantly compared to its predecessor. Benchmarks showed that RCKMPI2 outperforms its older version in nearly every respect, but still performs worse than RCCE_comm (see results in Section IV).

As will be shown later, the performance of RCCE_comm can be improved further by using non-blocking *send* and *receive* primitives instead of the blocking ones provided by RCCE. The iRCCE library [6] can be used to provide the required functionality as it adds MPI-like support for issuing non-blocking send and receive requests, accompanied by functions to cancel pending requests or to wait for requests to complete. However, the introduction of unnecessary features in functions acting as primitives may result in significant overhead that limits performance [7]. Thus, we have investigated steps to reduce such overheads in order to obtain fast implementations of collectives [8]. In this paper, we will elaborate on these points and compare their efficiency against RCKMPI2, a representative for full MPI solutions, and against RCCE_comm on top of RCCE as lightweight solution.

III. Optimizations

The considerable performance improvement by switching from RCKMPI to RCCE_comm and RCCE motivated a closer



(a) first round: communication (b) first round: summation (c) second round: communication

Fig. 1: Bucket algorithm

look at the distribution of runtime spent in each of the application's functions. Profiling revealed that the highest amount of communication time falls upon the *Allreduce* collective. *Allreduce* is an operation that combines one operand value of each process into a single result value by means of a binary operator. In the following, we assume this to be the addition operator "+", but in general any associative binary operator can be used. Formally, having p processes supplying operand vectors with n elements each, let $v_k[i]$ denote the i-th element of process k's operand vector ($0 \leq i < n$, $0 \leq k < p$). After collectively calling *Allreduce*, all processes should have a copy of the result vector res, with $\forall i : res[i] = \sum_{c=0}^{p-1} v_c[i]$.

For operand vectors exceeding 512 double precision floating point values, the *Allreduce* implementation of RCCE_comm uses Barnett's *bucket algorithm* [9]. This algorithm splits the operand vectors into p parts ("buckets") and distributes the computation of the individual buckets among the involved cores. It is organized into rounds, where in each round, a core sends its intermediate result of the previous round (its initial bucket in the first round) to its right neighbour (cf. Fig. 1a). Consequently, all cores receive a bucket of their left neighbor, which they combine element-wise with the corresponding elements of their local input operand (Fig. 1b). These actions are repeated in future rounds (Fig. 1c) until each core has computed one bucket of the result vector.

A. Relaxed synchronization

An analysis of the application's call graph profile showed that most of the time spent in *Allreduce* is used by RCCE's *send* and *receive* functions. For both of these functions, the sub-function `RCCE_wait_until`, which performs inter-core synchronization by waiting on flags to be manipulated by remote cores, accounted for roughly 15–40% of their runtime.

The reason why the bucket algorithm synchronizes this much can be found in its structure. The important points to notice here are the circular traffic pattern (cf. Figs. 1a and 1c), and the fact that cores both need to send and to receive a bucket each round. Since the *send* and *receive* primitives of RCCE are blocking, they can both return only after their matching counterpart has been called at the remote side. This requires cores to post the *send* and *receive* in a specific order to avoid deadlocks. In RCCE_comm, this is taken care of by the *odd-even pattern* that lets odd-numbered cores first call *receive*, followed by *send*, while even-numbered cores perform the calls in reverse order (cf. Alg. 1).

In addition to the overhead caused by branching on the process number, this forces the operations to finish in the given

```
Input: $bi$: bucket index; $left$, $right$: IDs of neighbor cores
1  if RCCE_comm_size() mod $2 = 0$ then
2      if RCCE_comm_rank() mod $2 = 0$ then
3          RCCE_send($result[bi], bucket\_size, right$);
4          RCCE_recv($recv\_bucket, bucket\_size, left$);
5      else
6          RCCE_recv($recv\_bucket, bucket\_size, left$);
7          RCCE_send($result[bi], bucket\_size, right$);
8  else
9      if RCCE_comm_rank() mod $2 = 1$ then
10         RCCE_send($result[bi], bucket\_size, right$);
11     else if RCCE_comm_rank() $> 0$ then
12         RCCE_recv($recv\_bucket, bucket\_size, left$);
13     if RCCE_comm_rank() mod $2 = 1$ then
14         RCCE_recv($recv\_bucket, bucket\_size, left$);
15     else
16         RCCE_send($result[bi], bucket\_size, right$);
17     if RCCE_comm_rank() $= 0$ then
18         RCCE_recv($recv\_bucket, bucket\_size, left$);
19 $result[bi] := result[bi] + recv\_bucket$;
```

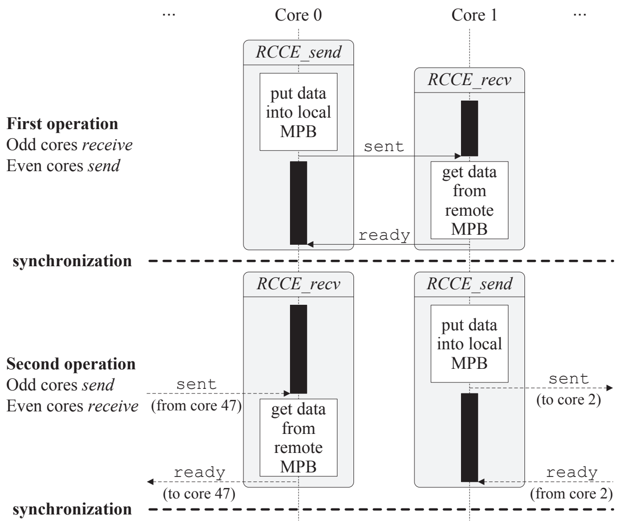
Algorithm 1: *Allreduce* round with blocking primitives

order, as synchronization occurs after the first and after the second call (cf. Fig. 2a). This order can force a core to wait on data, even when the data in question is already available at its left neighbor. For a core ranked p this is e.g. the case when its left neighbor $p-1$ must perform the *receive* first, but $p-1$ itself is waiting for data of its left neighbor $p-2$. Such wasting of time can be avoided by the use of non-blocking operations, which allow the *send* and *receive* to complete in any order. In that case, cores synchronize only once per round, i.e. on the completion of both the *send* and *receive* operations (cf. Fig. 2b).

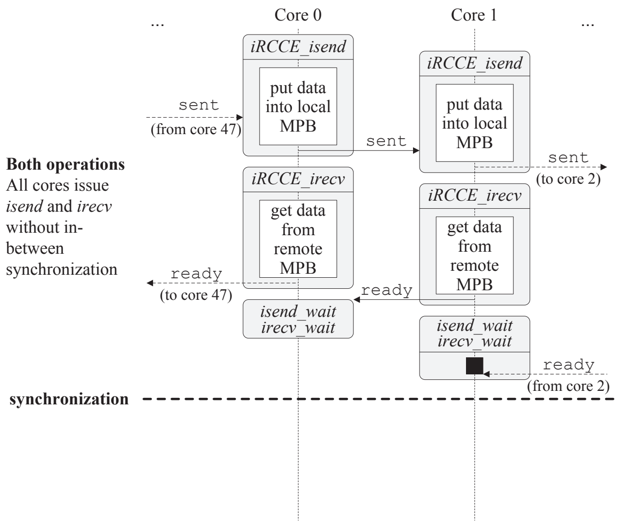
To accelerate *Allreduce*, we included the iRCCE library and replaced the blocking calls within RCCE_comm by iRCCE's non-blocking equivalents. As a side effect, the use of non-blocking primitives allows to issue the start of the *send* and *receive* in an arbitrary order. Consequently, the odd-even

```
Input: $bi$: bucket index; $left$, $right$: IDs of neighbor cores
1 iRCCE_isend($result[bi], bucket\_size, right, \&sndreq$);
2 iRCCE_irecv($recv\_bucket, bucket\_size, left, \&rcvreq$);
3 iRCCE_isend_wait($\&sndreq$);
4 iRCCE_irecv_wait($\&rcvreq$);
5 $result[bi] := result[bi] + recv\_bucket$;
```

Algorithm 2: *Allreduce* round with non-blocking primitives



(a) blocking primitives with odd-even pattern



(b) non-blocking primitives

Fig. 2: Signaling sequence of one *Allreduce* round

pattern is no longer required, allowing a considerably simpler implementation of the message exchange phase (cf. Alg. 2).

B. Overhead minimization

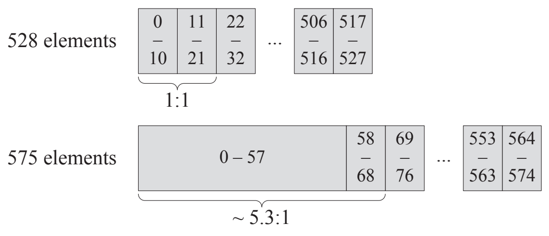
While evaluation of non-blocking based *Allreduce* showed an absolute increase in performance, the percentage of time spent for communication management, like setting up and scheduling a data transfer, actually increased compared to the blocking primitives of RCCE. This is due to the fact that when communication calls do not block until finished, multiple pending send or receive requests can be active at the same time. iRCCE supports this by storing incoming requests inside a linked list and processing them in the order of their arrival. In addition to communication initiation, iRCCE contains functions for waiting on the completion of a particular request or for canceling its execution.

While these features are very comfortable seen from an application developer's perspective, they come at the cost of significant overhead. For operations like *Allreduce* that make heavy use of communication primitives, this overhead can be a limiting factor for performance. Since implementations of many collectives are organized into rounds during which at most one message is being sent and received simultaneously, this overhead is caused by functionality that is not even used. As a consequence, we have implemented our own set of non-blocking primitives as a RCCE extension called *RCCEnb* that limits the maximum number of active transmissions to at most one for each direction (incoming and outgoing messages). This way, list-keeping can be omitted completely, and also the remaining management (e.g. communication setup) is simplified significantly. Measurements have shown that replacing iRCCE with our own set of non-blocking primitives effectively halves the latency of an *Allreduce* call (see Section IV).

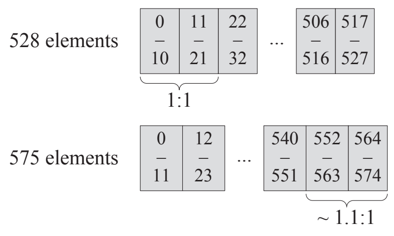
C. Fair load balancing

When two cores are synchronized with each other, this usually means that the core reaching the synchronization point in its code first must wait on the other core catching up. In case of collective operations organized in rounds, cores have to wait for incoming data, which is subsequently forwarded or processed locally. If – like in case of the bucket algorithm – all cores are synchronized with each other, e.g. by a circular traffic pattern (cf. Fig. 1), the amount of work to be done each round should be equal for all cores in order to minimize the waiting times imposed by synchronization.

In the bucket algorithm, the amount of work to be done directly corresponds to the bucket size. Since each core processes a different bucket, all buckets should ideally contain the same number of elements. RCCE_comm determines the bucket size by performing an integer division of the input vector's element count by the number of cores, i.e. $\lfloor n/p \rfloor$. If



(a) Standard bucket sizes (RCCE_comm)



(b) Optimized bucket sizes

Fig. 3: Bucket size ratios for different vector lengths

n is not a multiple of p, the remaining $n \mod p$ elements are added to the first bucket. This can lead to a serious imbalance in bucket sizes, as the remainder of the integer division can be larger than the result. Consider the example of Fig. 3a: For 575 elements, the general bucket size is 11 elements. Until the next multiple of 48 (i.e. 576) is reached, further elements are simply added to the first bucket, thereby linearly increasing the imbalance. At the worst point of 575 elements, the remainder of the integer division is 47, resulting in the first bucket containing $11 + 47 = 58$ elements while all other buckets comprise just 11 elements. The ratio of the first bucket's size to that of other buckets is about $5.3 : 1$ here, meaning that the core processing the first bucket runs about five times as long as the other cores each round, effectively slowing them down by synchronization.

We adapted the bucket sizing method such that the remaining elements of the integer division are distributed among a set of buckets instead of assigning all to the first one, hence providing a more even distibution. Particularly, the first $n \mod p$ buckets are assigned one additional element to take up the non-evenly distributable elements. This can be seen in Fig. 3b: For the worst-case situation described in the previous paragraph, the first 47 buckets are assigned one extra element each such that they have 12 elements in total. Identical to the old sizing method, the last bucket still comprises 11 elements. This distribution has a balance ratio of roughly $1.1 : 1$, which is a considerable improvement.

D. Data flow optimization

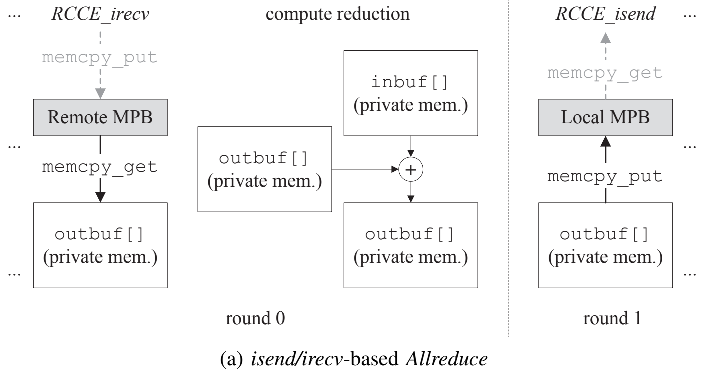
Since the on-die local memory (i.e. the MPB) is very limited in its size, cores usually store working data in their private off-chip memory partition. In order to transport data between cores, the sending core copies the data from this private memory into its local MPB and sends a signal to the receiving core. As soon as the receiving core receives this signal, it copies the data from the sender's MPB into its own private memory ("pull"-style communication). While this procedure works well if received data is to be processed further locally, collective operations often use cores as "relay nodes" that simply have to forward received data to the next core with little or no local processing. In this case, the frequent copying of data into and out of the MPB can be avoided by allowing in-transit data to be kept inside the MPB.

For our example of *Allreduce*, each round of the bucket algorithm moves operand data as shown in Fig. 4a. At the start of a round, a core receives a remote bucket by calling *(i)recv*. Internally, this function uses the `memcpy_get` routine to copy the data out of the sender's MPB into private off-chip memory. In the computation phase, both the received operands and the local ones are read from off-chip RAM and are combined subsequently. The results are written back to private memory. At the start of the next round, the result vector must be sent to the right neighbor, which is done by calling *(i)send*. Similar to the receiving case, this uses `memcpy_put` internally to copy the data from off-chip memory into the local MPB. Since the results of a round have to be transferred to the right neighbor in the next round, but are never used locally again, the overhead of copying can be avoided by keeping them inside the MPB.

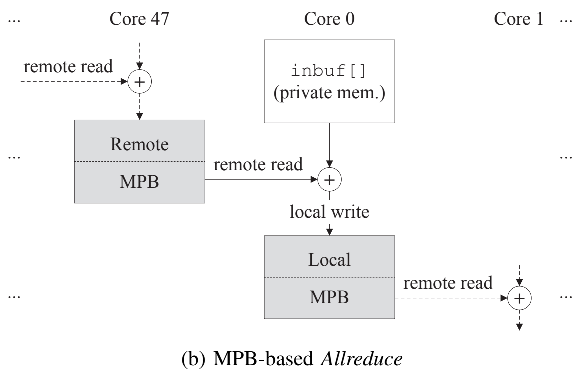
The simplified approach is shown in Fig. 4b for one round. Assume that core 47, i.e. the left neighbor of core 0, has the intermediate result of the previous round (or its local operand vector in case of the first round) stored in its MPB. Then, core 0 can directly read the remote operands from core 47's MPB, combine it with its local operands, and store the result in its local MPB. This procedure omits the two copy operations involved by the *send* and *receive* calls and requires only a single access to off-chip memory. As MPBs are filled by their local core and read by the right neighbors simultaneously, a double buffering approach is used where the MPB is split in half as indicated by the dashed lines in Fig. 4b. While one half contains the results of the previous round, acting as a read buffer for the right neighbor's remote operands, the other half serves as storage for the current round's results. At the end of a round, cores are synchronized with each other, and the roles of the buffers are swapped afterwards.

IV. Experimental results

We applied the proposed optimizations to six collective operations of RCCE_comm, namely *Allgather*, *Allreduce*, *Alltoall*, *Broadcast*, *Reduce*, and *ReduceScatter*. Then we evaluated the effect of the different optimization steps on the latency of individual operations, and on the runtime of a complete application. To this end, we configured the SCC with the standard preset, clocking the cores at 533 MHz, the network and DRAM at 800 MHz. On the software side, we

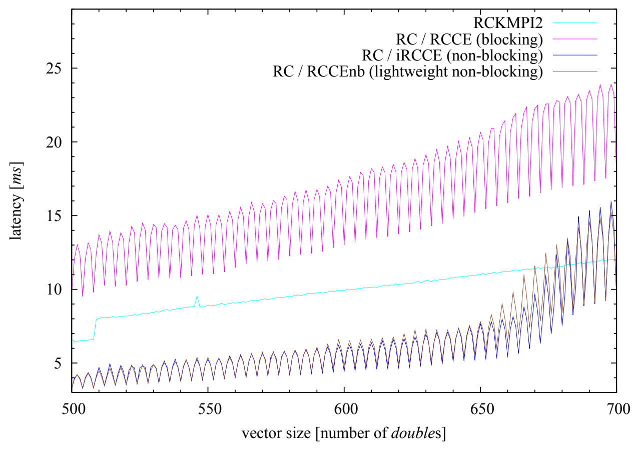


(a) *isend/irecv*-based *Allreduce*

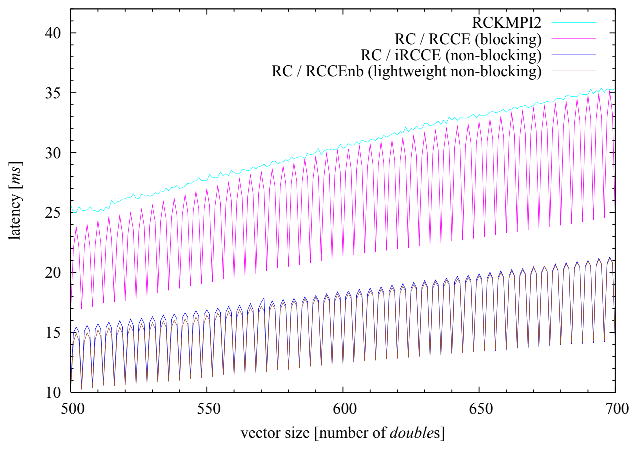


(b) MPB-based *Allreduce*

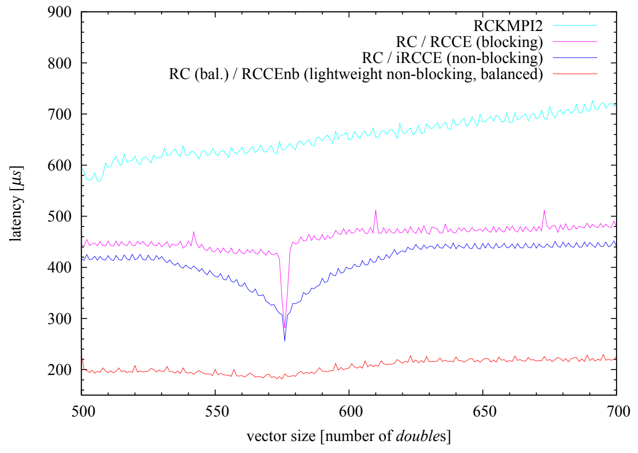
Fig. 4: Data movement inside an *Allreduce* round



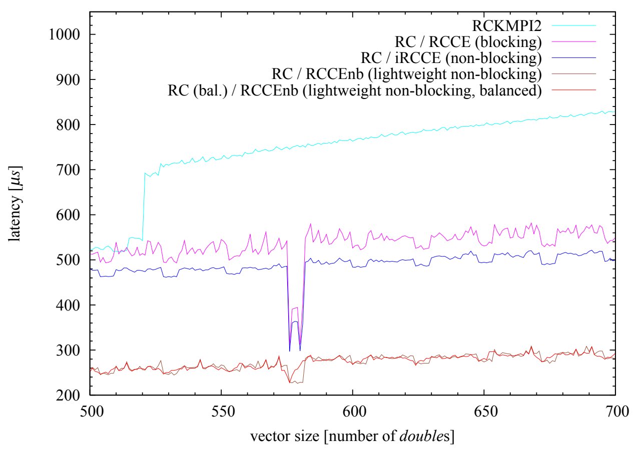
(a) Allgather



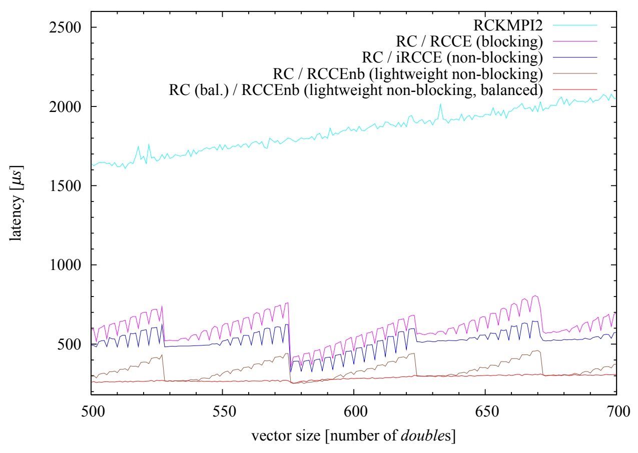
(b) Alltoall



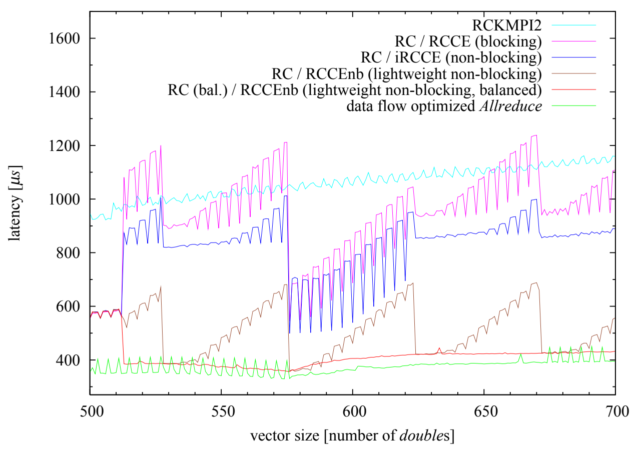
(c) ReduceScatter



(d) Broadcast



(e) Reduce



(f) Allreduce

Fig. 5: Latencies of the optimized collectives ("RC /" reads "RCCE_comm on top of")

used sccKit 1.4.1.3 and the latest available versions of the communication libraries, specifically RCCE 1.1.0, iRCCE 1.2, and both RCKMPI2 as well as RCCE_comm in revision 303 of the public SVN repository [10].

Fig. 5 shows the latency of individual operations. Nearly all graphs of RCCE-based implementations show spikes with a period of four elements, which matches the size of an L1 cacheline. As RCCE performs an extra synchronization for the transmission of bytes that do not fill a whole cacheline, the transfer of messages not being a multiple of the L1's line size takes longer than a transfer of messages consisting only of complete cachelines. As RCKMPI2 uses its own method for data transmission, it does not suffer from this issue. Consequently, its latency graphs display a smoother behavior.

With the exception of *Allgather*, RCCE_comm outperforms RCKMPI2 in every respect even in its original form and based on RCCE's blocking primitives. To provide a fair comparison, we hence chose the combination of RCCE_comm using RCCE primitives as baseline implementation. Relaxing the synchronization by switching to non-blocking primitives gives speedups between 9.5% for *Broadcast* and a factor of 2.6x for *Allgather*. The use of more lightweight primitives has nearly no performance impact on *Allgather* and *Alltoall*, but accelerates the other four collectives by factors between 1.5x (*Reduce*) and 2x (*ReduceScatter*). For *Reduce* and *Allreduce*, the load imbalance issue is clearly visible in form of the ramp-like latency increments between element counts of multiples of 48. The balanced versions avoid this and increase performance by more than 60% (*Reduce*) or 90% (*Allreduce*) at best, or around 20% on average. Finally, the data flow optimized version of *Allreduce* results in an average acceleration of an additional 10%. While we initially expected this to have a much higher impact on performance, the relatively high latency of local MPB accesses, caused by a workaround for a bug in the MPB arbitration logic[1], reduces the effectiveness of this optimization step. In spite of the potential we attribute to a simplified data flow, we hence did not apply this step to further collectives.

To show the impact on the runtime of a complete application, we benchmarked a thermodynamics code performing *Grand Canonical Monte Carlo* simulation [11]. As Fig. 6 shows, low-latency collectives have a notable effect on this application's performance. The combined optimizations achieve a speedup of 42% compared to the baseline of RCCE_comm and RCCE, the main contribution (17.8%) being due to lightweight primitives.

V. Conclusion

The low latencies in on-chip networks enable fine-grained parallelism involving a higher communication rate than traditional cluster architectures. While point-to-point communications directly benefit from this, message-passing applications often use collective operations to distribute and gather data. As a consequence, collectives should also be subjected to latency optimizations in order to turn the lower communication latency into application performance.

[1]see http://communities.intel.com/docs/DOC-5405

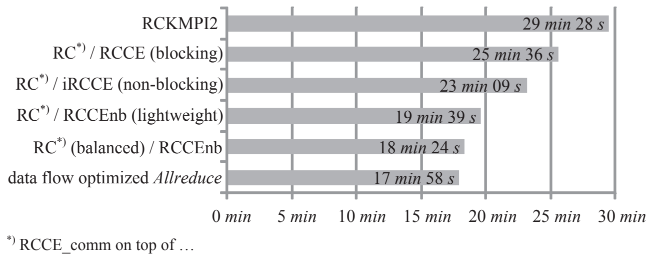


Fig. 6: Application performance

We have investigated optimizations for collectives that have been shown to accelerate individual operations by factors of 1.6x up to 2.5x. For an example application, the proposed optimizations result in a total speedup of 42%, thus demonstrating the importance of low latency collectives for application performance.

Acknowledgment

The authors 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.

References

[1] T. Mattson and R. van der Wijngaart, "RCCE: a Small Library for Many-Core Communication," 2011, (2012, August 30). [Online]. Available: http://communities.intel.com/docs/DOC-5628

[2] "The SCC programmer's guide," 2011, (2012, January 20). [Online]. Available: http://communities.intel.com/docs/DOC-5684

[3] 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.

[4] 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

[5] I. A. Comprés Ureña and M. Gerndt, "Improved RCKMPI's SCCMPB Channel: Scaling and Dynamic Processes Support," in *Proc. 4th Many-core Applications Research Community (MARC) Symposium*, Potsdam, Germany, December 2011, pp. 1–6.

[6] 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

[7] A. Kohler, J. M. Castillo-Sanchez, J. Groß, and M. Radetzki, "Minimal MPI as Programming Interface for Multicore Systems-on-Chips," in *Proc. Forum on Specification and Design Languages (FDL)*, Vienna, Austria, Sept. 2012, pp. 120–127.

[8] A. Kohler, P. Gschwandtner, T. Fahringer, and M. Radetzki, "Low-Latency Collectives for the Intel SCC," in *Proc. Conference on Cluster Computing (CLUSTER)*, Beijing, China, Sept. 2012.

[9] 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.

[10] "MARC SVN repository," (2012, September 11). [Online]. Available: http://marcbug.scc-dc.com/svn/repository/trunk

[11] D. Adams, "Grand canonical ensemble Monte Carlo for a Lennard-Jones fluid," *Molecular Physics*, vol. 29, no. 1, pp. 307–311, 1975.